CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formal Topology in Univalent Foundations

Master's Thesis in Computer Science and Engineering

Ayberk Tosun

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Master's thesis 2020

Formal Topology in Univalent Foundations

Ayberk Tosun

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Logic and Types
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Formal Topology in Univalent Foundations
Ayberk Tosun

Supervisor: Thierry Coquand, Department of Computer Science and Engineering
Examiner: Nils Anders Danielsson, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Division of Logic and Types
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset using LaTeX
Agda code typeset in PragmataPro
Gothenburg, Sweden 2020

Formal Topology in Univalent Foundations
Ayberk Tosun
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Formal topology is a mathematical discipline that aims to interpret topology in type-theoretical terms, that is, constructively *and* predicatively. Type theory has recently undergone a transformation [42] through insights arising from its association with homotopy theory, resulting in the formulation of the notion of a *univalent* type theory, and more broadly, a univalent foundation.

We investigate, in this thesis, the natural continuation of the line of work on formal topology into univalent type theory. We first recapitulate our finding that a naive approach to formal topology in univalent type theory is problematic, and then present a solution to this problem that involves the use of higher inductive types. We hence sketch the beginnings of an approach towards developing formal topology in univalent type theory. As a proof of concept for this approach, we develop the formal topology of the Cantor space and construct a proof that it is compact.

The development that we present has been carried out using the cubical extension of the Agda proof assistant [44]. No postulates have been used and the development typechecks with the `--safe` flag of Agda. The presentation in this thesis amounts to an informalisation of this formal development.

Keywords: topology, formal topology, pointless topology, formal space, locale, locale theory, frame, homotopy type theory, univalence, univalent foundations, agda, cubical agda

Acknowledgements

I would like to start by thanking my supervisor Thierry Coquand for all his efforts that have made this thesis possible. He has not only suggested the topic of this thesis, closely supported all my work, and patiently answered my (at times never-ending) questions, but he has also inspired much of my thought on type theory through various lectures and discussions, some in the context of the *Types for Programs and Proofs* course he co-taught in Fall 2018. In fact, this course has been the most important course I have taken at Chalmers so I would like to thank all of its teaching staff: Peter Dybjer, Andrea Vezzosi, and Fabian Ruch.

I would like to also thank Nils Anders Danielsson for serving as the examiner of this thesis. His suggestions and careful review of the first draft of this thesis have been extremely helpful.

Throughout the two years I have spent at Chalmers, I have had the chance to engage with the lively type theory community of Gothenburg, most importantly through the *Initial Types Club* (ITC). ITC has provided a conducive environment for fun discussions on type theory (every Thursday), from which I have learned a lot. I would like to therefore thank all ITC regulars for these meetings, but especially Andreas Abel, Jesper Cockx, Carlos Tomé Cortiñas, Nachiappan Valiappan, and Sandro Stucki for undertaking the organisational efforts required to keep ITC going.

I would like to thank my friends Joel Sjögren, Warrick Macmillan, Alexander Fuhs, and Robert Krook for various discussions on logic, types, categories, and much more. I would also like to thank Joel Sjögren for accepting to serve as an opponent for this thesis: his careful reading and suggestions have proved quite beneficial. Joomy Korkut deserves special mention for getting me interested in Agda in the first place.

Of course, anything I have done has been possible thanks to the support of my parents to whom I am, and will always be, grateful.

Finally, I would like to thank the Adlerbert Foundation for funding my studies at Chalmers through the Adlerbert Study Scholarship.

Ayberk Tosun, Gothenburg, June 2020

Contents

1
Introduction

This thesis is about topology, the branch of mathematics that studies *continuous* transformations between topological spaces. The raison d'être of topological spaces is to allow the formulation of the notion of continuity that pervades practically all of mathematics, as pointed out by Sylvester [5, pg. 27]: "if I were asked to name, in one word, the pole star round which the mathematical firmament revolves, the central idea which pervades the whole corpus of mathematical doctrine, I should point to Continuity as contained in our notions of space, and say, it is this, it is this!". Let us then start by considering the question of what topology is.

The usual definition of a topological space is the following [29].

Definition 1.1 (Topological space)**.** A topology on a set X is a class $\mathcal{T}$ of subsets of X such that

- The trivial subsets $X, \emptyset \subseteq X$ are in $\mathcal{T}$,
- $\mathcal{T}$ is closed under *finite* intersections, and
- $\mathcal{T}$ is closed under *arbitrary* unions.

A topological space is a set X equipped with a topology $\mathcal{T}$.

When a set X forms a topological space i.e. is equipped with some topology $\mathcal{T}$, its elements are called *points* and the members of $\mathcal{T}$ are called *open sets* of X. A noteworthy thing about this definition is the asymmetry between intersection and union: a topology is required to be closed only under finite intersection whereas it *must* be closed under arbitrary union.

This asymmetry is familiar to computer scientists from computability theory. Let P be a program whose internal structure we do not have access to. One could say it is a "black box" although this is not to say it is a scenario occurring only in theory; black boxes *do* occur in practice. A compiled program whose source code one does not have access to, or even, a program whose source code is not familiar to one, can be seen as examples of black boxes from one's perspective.

One way or another, suppose that we are in a situation in which we have to understand the behaviour of some program P by running it and examining its output. For the sake of example, suppose that P does something simple such as printing a sequence of integers (which we do not know is finite or infinite). We run P and observe that it starts printing a sequence of integers:

(1.2) `7 8 11 2 2 42 ⋯`

We then observe the output and judge whether or not P satisfies certain properties. Among the properties of P, some are special in that we can observe the fact that P satisfies them. Consider, for instance, the property:

(1.3) "P eventually prints `42`".

The knowledge about the behaviour of P we can obtain from (1.2) is sufficient to judge that P satisfies this property. In other words, at the point of time at which we make the observation (1.2), we have gathered complete evidence for the fact that P satisfies this property: there is no need to make any further examination. Whenever a program satisfies Property 1.3, there will be such a *finite* prefix of the output of the program by observing which we will be able to judge for sure that P satisfies this property. We refer to such properties as *observable properties*.

Some properties of P, on the other hand, are not observable. Consider, for instance, the property:

(1.4) "P prints at most two `2`s".

Suppose that P is a program such that, after printing the sequence of integers in (1.2), starts to continually print `0`. Although P then certainly satisfies this property, we will not be able to judge on the basis of a finite observation that it does so. We will gain more and more information about the behaviour of P but no amount of finite information will allow us to *rule out* the possibility that P prints a third `2`.

Let us then make precise what exactly we mean by an "observable property".

Definition 1.5 (Observable property (informal))**.** Let P be a program that prints a possibly infinite sequence σ of integers. We say that an extensional property ϕ of P is observable if and only if:

> if P satisfies the property ϕ, there exists a finite prefix $\sigma|_i$ of the output σ of P such that P is *verified* to satisfy ϕ on the basis of $\sigma|_i$: no extension of $\sigma|_i$ can disrupt the fact that P satisfies ϕ.

Another name for the class of observable properties in this context is "semidecidable" although we will use the term "observable" as we will soon be generalising it to things that are not programs. Let us now consider some properties of the class of observable properties themselves.

Let $\phi_1, \cdots, \phi_n$ be a *finite* number of observable properties and assume that

$$\phi_1 \wedge \cdots \wedge \phi_n$$

holds. There must be *stages*

$$m_1, \cdots, m_n$$

such that ϕ_k is verified at stage m_k. $\phi_1 \wedge \cdots \wedge \phi_n$ must then be verified at stage

$$\max(m_1, \cdots, m_n).$$

This is to say: if $\phi_1, \cdots, \phi_n$ are *observable* then so is $\phi_1 \wedge \cdots \wedge \phi_n$. In other words,

the class of observable properties is closed under finite intersection.

Similarly, let $\{ \psi_i \mid i \in I \}$ be an *arbitrary* number of observable properties such that $\bigvee_i \psi_i$ holds. As the disjunction holds, it must be the case that some ψ_i holds, meaning it must be verified at some stage m as it is observable. $\bigvee_i \psi_i$ is hence verified at stage m. This is to say: if $\{ \psi_i \mid i \in I \}$ are *observable* then so is $\bigvee_i \psi_i$. In other words,

the class of observable properties is closed under arbitrary union.

We have given an informal argument that the class of observable properties of programs is a topology. This characterises the perspective towards topology we will adopt in this thesis: we will view it as a mathematical theory of observable properties. This perspective towards topology has been developed by many researchers. Although it originates in the work of Dana Scott [36], it was first made explicit by Michael Smyth [38]. It was then subsequently developed by, most saliently, Abramsky [1], Vickers [46], Escardó [15], and Taylor [40]. Our presentation in this chapter follows specifically Smyth [38].

When one approaches topology from such a computational perspective, it is natural to consider the question: if a topology is like a system of observable properties, what are then the points? In other words, what are the observations about? The idea is that the points of the topological space are programs whereas the open sets are observable properties of these programs—but there is a subtlety: this *pointful* view is not entirely compatible with our computational view of topology.

We said that we are viewing programs as black boxes meaning we do not have full understanding of their behaviour; we only have partial understanding that we form on the bases of finite examinations. To work with the points directly then is to trivialise the fact that we cannot, in general, fully understand the behaviour of a program by partial examinations.

A view of topology more compatible with this perspective is what is called *pointless* topology [26], originating from the seminal work of Marshall Stone [39]. In pointless topology, points take a back seat to open sets: instead of starting with a set of points, from which the notion of an open set is derived, we start with a primitive set of observable properties (called the "opens"), and we then axiomatise the behaviour of these to reflect that they behave as a system of observable properties. In other words, we axiomatise the behaviour of the lattice of open sets of a topological space directly, that is, as an arbitrary lattice rather than a lattice of subsets. The points can then be defined in terms of the opens but what is interesting is that one can entertain important questions of topology without mentioning the points at all [26].

Why is this view of topology more compatible with our perspective? We previously said that if we are to understand the behaviour of a black-box program, we have to do this by means of finite examinations of its output. This is for the simple reason that finite outputs of a program are the only data about its behaviour that we can computationally access. The pointless vantage point can then be motivated by the fact that we only want to write down things that we can computationally access (i.e. construct).

We have not yet precisely defined what a pointless topology is so let us do that now. As mentioned, the idea is to axiomatise the behaviour of the lattice of open sets. Let $\mathcal{O}$ be a set of opens, that are some unspecified primitive entities.

1. Corresponding to the set-inclusion partial order in the pointful case, we require that there be a partial order $\_ \sqsubseteq \_ \subseteq \mathcal{O} \times \mathcal{O}$.

2. Corresponding to the fact that open sets are closed under finite intersection, we require that there be a binary meet operation $\_ \wedge \_ : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$ and a nullary one $\top : \mathcal{O}$.

3. Corresponding to the fact that open sets are closed under arbitary union, we require that there be a join operation of arbitrary arity: $\bigvee \_ : \mathcal{P}(\mathcal{O}) \to \mathcal{O}$.

In addition to these, we have to require that these operations satisfy an infinite distributivity law, on which we will elaborate in Chapter 3. The official name for such a lattice that embodies a logic of observable or finitely verifiable properties is *frame* [46]. Indeed, frames form a category whose morphisms are frame homomorphisms; objects of the opposite category are called *locales*. Therefore, frames and locales are synonymous *as long as no mentions to morphisms are made* [46, 25]

The precise relationship between the pointful and the pointless approaches is an *adjunction* called *Stone duality*, that is not an equivalence in general. More precisely, this adjunction is between the category of topological spaces and the category of frames, and it comprises the following functor pair: the functor taking a topological space to its frame of open subsets, and the functor taking a frame to the topological space formed by its set of points (which we will not define here). Spaces are equivalent to their pointless representation exactly when they satisfy the property of being sober. We will not go into this topic in detail; The interested reader is referred to the work of Johstone [26, 25].

We have provided some preliminary motivation for pointless topology that applies in the context where we are viewing it as a theory of observable properties. Does this motivation apply also from the perspective of general topology? Topology notoriously relies on classical reasoning in many of its fundamental theorems such as the Tychonoff theorem. A more concrete advantage of pointless topology is that by doing topology pointlessly, we can avoid classical reasoning and hence gain a computational understanding of it—most saliently of its crucial theorems such as the Tychonoff theorem [8, 45]. This point was put eloquently by Johnstone [25, pg. 46]:

> It is here that the real point of pointless topology begins to emerge; the difference between locales and spaces is one that we can (usually) afford to ignore if we are working in a "classical" universe with the axiom of choice available, but when (or if) we work in a context where choice principles are not allowed, then we have to take account of the difference—and usually it is locales, not spaces, which provide the right context in which to do topology. This is the point which, as I mentioned earlier, André Joyal began to hammer home in the early 1970s; I can well remember how, at the time, his insistence that locales were the

> real stuff of topology, and spaces were merely figments of the classical mathematician's imagination, seemed (to me, and I suspect to others) like unmotivated fanaticism. I have learned better since then.

This thesis is concerned with investigating topology in the context of type theory, *without* modifying its logic by the addition of postulates. The first prerequisite for this is that we be able to develop topology constructively, that is, without relying on classical principles. As pointed out by Johnstone, pointless topology can help us here as it allows us to understand topology in constructive terms. The goal of carrying out topology in type theory, however, presents further challenges: we have to understand topology not only constructively but also *predicatively*.

This is the subject matter of the branch of mathematics known as formal topology, first instigated by Martin-Löf and Sambin [35] in the early days of type theory. The idea is that a formal topology recasts the notion of a frame into a form that resembles a formal proof system. In addition to enabling the importation of proof-theoretical ideas, such a formal presentation has the virtue of being *predicative* hence enabling the development of pointless topology in type theory.

Our goal in this thesis is to carry out formal topology in the context of univalent type theory [42, 14]. By now, it has become clear that univalence addresses certain severe shortcomings of type theory, and presents a new way of engaging in mathematical developments in it. The question of what novelties it presents for formal topology is therefore a natural one. In attempting to answer this question, we follow a particular approach to formal topology, implementing an idea of Coquand [9] to define formal topologies as posets endowed with "interaction" structures [32, 21].

In summary, this thesis presents two contributions. The primary contribution is an answer to the question of how formal topology can be done in univalent type theory. In particular, we explain that an as is development of formal topology in univalent type theory is problematic (at least ostensibly), although this issue can be remedied by using higher inductive types (similar to how the Cauchy reals are constructed in the HoTT book [42]). The secondary contribution is the development of Coquand's idea of doing formal topology with interaction systems to a further extent. To the author's knowledge, this thesis presents the first formalised[1] development of this approach to formal topology. All results given as a Definition, Proposition, Lemma, or Theorem are present in the Agda formalisation with the exception of Definition 4.5. See the `Index` module for a detailed list explaining how each result maps to the Agda formalisation.

This thesis is structured as follows. In Chapter 2, we summarise the fundamentals of univalent type theory. In Chapter 3, we present our development of frames and constructs related to them. In Chapter 4, we present our main development of formal topology in univalent type theory. In Chapter 5, we present a prime example of a formal topology: the Cantor space. As a proof of concept for the formal study

[1] The Agda formalisation is given in Appendix A and can also be downloaded at:

`https://ayberkt.gitlab.io/msc-thesis/thesis-agda-formalisation.tar`.

The `Index` module provides a list explaining how each definition and proposition from the thesis maps to the Agda code.

of topological properties in univalent type theory, we prove an important property of the Cantor space, namely, that it is compact.

2
Foundations

We present in this chapter the basics of univalent type theory. We assume that the reader is familiar with standard (intensional) type theory and refrain from presenting the formal calculus from scratch; for this, the reader is referred to Appendix A of the HoTT Book [42]. We instead focus on the fundamental notions and theorems of univalent type theory.

Our presentation in this chapter has been heavily influenced by three resources: (1) the HoTT Book [42], (2) Martín Escardó's introductory exposition of HoTT/UF in Agda [14], and (3) the `cubical` library of Agda [28].

2.1 Formal vs. informal mathematics

We will use type theory in this thesis in an informal way, much like how set theory is frequently used without committing to a specific one of its implementations. The work to be presented is in fact the *informalisation* of a completely formalised development, for which the formal system of choice is Cubical Type Theory[1] (CTT) [7]. CTT is just one instance of a univalent foundation, and its details will therefore not concern us in this thesis. We will instead abstract over the implementation, and use univalent type theory as a *practical* foundation [41, 22] whose formal details are left unspecified. The reader interested in the details of the formal development can find the (cubical) Agda formalisation in Appendix A.

To keep our informal presentation precise, we will closely follow the notational conventions of the HoTT Book [42], as explained in [42, Sec. I.1]. For instance, we will often make use of pattern matching: we will write $(x, \_) : \sum_{(a\,:\,A)} B(a)$ rather than using the projection notation for projecting out the first component. There will be some minor notational deviations and abuses of language, which will be addressed locally, in the relevant sections.

Finally, we note that we will be introducing new definitional equalities in various definitions. The definienda introduced by these definitional equalities are hyperlinked to their definitions, which might be useful to the reader who is reading this thesis electronically. Such hyperlinked definienda are coloured dark green.

[1] The implementation of Cubical Type Theory in the cubical extension of Agda [44], to be specific.

2.2 Univalent foundations: the idea

Starting with the work of Hofmann and Streicher [24], the idea that types (as conceived in intensional type theory) have connections to homotopy theory had been brewing in the type theory community, the key observation being that types and their types of equality proofs can be viewed as spaces (identified up to homotopy) and their spaces of *paths*. The solidification of this idea started around 2006, through the efforts of various researchers [49, 6, 18, 43, 47] exploring this connection from different perspectives. It was around this time that the concept of univalence emerged. We will refrain from addressing the history of the subject in detail.

Among these researchers, Vladimir Voevodsky [48] emphasised the possibility of transforming type theory through insights arising from this line of work. He coined the term "univalent foundations" to refer to his vision of mathematical foundations that support univalent mathematics.

So what is univalence and what are the novelties it presents for mathematical foundations? Foundations have historically suffered problems with the notion of equality. In mathematics, superficial differences between structures are immaterial. For instance, we never want to talk about the difference between isomorphic graphs. It has therefore been common practice in mathematics to work with the "right" notion of equality between structures: homeomorphism for topological spaces, isometry for metric spaces, equivalence for categories, group isomorphism for groups, and so on.

A reasonable expectation from a foundational system for mathematics is that it be able to accommodate the common practices occurring in its high-level use in a natural way. Both set theory and type theory have suffered the problem that they could not accommodate this practice well. In the context of type theory, however, this problem can be seen more clearly: the implementation of this informal practice in type theory gives rise to what is known as "setoid hell" [2].

Homotopy type theory, being the first of univalent type theories, is arguably the first foundational system that validates this practice of working with the "right" equality. As remarked by Escardó in his introductory exposition of HoTT/UF [14], once the type corresponding to a certain algebraic structure has been written down in univalent type theory, the identity type corresponds *automatically* to the type of isomorphisms for that structure. It is quite remarkable that this can be achieved by a minor modification to type theory: first, make sure that the rich structures of types are not trivialised by the likes of Axiom K, and then add a simple axiom—the univalence axiom—allowing one to prove that *equivalent* types are equal.

One immediate application of this axiom is that it allows one to prove function extensionality, that is commonly added as a postulate to type theory. The fact that one has to extend type theory with a postulate for something as natural as function extensionality can be considered a deficiency of type theory, and that univalence renders it into a theorem can therefore be considered an noteworthy improvement.

2.3 Equivalence and univalence

At the heart of univalent type theory lies the notion of *type equivalence*. We build up to its definition (Defn. 2.3) in this section.

Following the HoTT book [42, Sec. I.1.12], we denote by $x =_A y$ the type of identity proofs (or "identifications") between terms x and y. This is given by the constructor

$$\mathsf{refl} : \prod_{(x\,:\,A)} x =_A x,$$

whose elimination counterpart is the usual path induction rule. As mentioned before, we will abstract over such technicalities and engage in a high-level use of univalent type theory.

We start by delineating the class of types that have exactly one inhabitant.

Definition 2.1 (Contractible)**.** A type A is called contractible if it has exactly one inhabitant:

$$\mathsf{isContr}\,(A) \quad :\equiv \quad \sum_{(c\,:\,A)} \prod_{(x\,:\,A)} c =_A x.$$

Definition 2.2 (Fiber)**.** Given types $A : \mathcal{U}_m$, $B : \mathcal{U}_n$, a function $f : A \to B$, and some $y : B$, the fiber of f over y is the type of all inhabitants of A that are mapped by f to y:

$$\mathsf{fiber}\,(f, y) \quad :\equiv \quad \sum_{(x\,:\,A)} f(x) =_B y.$$

The following definition of type equivalence was first formulated by Voevodsky.

Definition 2.3 (Equivalence)**.** Given types $A : \mathcal{U}_m$, $B : \mathcal{U}_n$, a function $f : A \to B$ is an *equivalence* if $\mathsf{fiber}\,(f, y)$ is a contractible type for every $y : B$:

$$\mathsf{isEquiv}\,(f) \quad :\equiv \quad \prod_{(y\,:\,B)} \mathsf{isContr}\,(\mathsf{fiber}\,(f, y))\,.$$

We will denote by $A \simeq B$ the type of equivalences between types A and B.

Definition 2.4 (Equivalence of types)**.** Given $A : \mathcal{U}_m$, $B : \mathcal{U}_n$,

$$A \simeq B \quad :\equiv \quad \sum_{(f\,:\,A\to B)} \mathsf{isEquiv}\,(f)\,.$$

One may wonder why the notion of type equivalence is not the familiar categorical notion of isomorphism, that is, a function between types that is invertible. We will address this point in Section 2.4.1.

Definition 2.5 (Identity equivalence)**.** The identity function on every type A can easily be seen to be an equivalence. Therefore there exists a function:

$$\mathsf{idEqv} : \prod_{(A\,:\,\mathcal{U}_n)} A \simeq A.$$

Given two types $A, B : \mathcal{U}_n$, and a proof $p : A = B$ that they are equal, we can clearly prove that they are equivalent.

Definition 2.6. Given any $A, B : \mathcal{U}_n$ with a proof $p : A = B$, there exists a function:

$$\mathsf{idToEquiv} : \prod_{(A,B : \mathcal{U}_n)} A = B \to A \simeq B.$$

In other words, it is justified intuitively that equality is stronger than equivalence. The famous univalence axiom states, essentially, that equivalence is as strong as equality. Formally, this amounts to saying that idToEquiv is an equivalence.

Axiom 1 (Univalence)**.** The following type has an inhabitant:

$$\prod_{(A,B : \mathcal{U}_n)} \mathsf{isEquiv}\left(\mathsf{idToEquiv}\left(A, B\right)\right).$$

The addition of this simple axiom into type theory has some surprising consequences. Perhaps most importantly, it allows us to *prove* function extensionality as mentioned.

Definition 2.7 (Extensional equality)**.** Given types $A : \mathcal{U}_m$, $B : \mathcal{U}_n$, and functions $f, g : A \to B$, f and g are said to be extensionally equal if the following type is inhabited:

$$f \sim g \quad :\equiv \quad \prod_{(x\ :\ A)} f(x) = g(x).$$

Proposition 2.8 (Function extensionality)**.** *Two functions are equal whenever they are extensionally equal.*

This result was first proven by Voevodsky[2].

2.4 Homotopy levels

One of the fundamental observations arising from the association of type theory with homotopy theory is that we can classify types with respect to the nontrivial homotopy structure they bear. Let A be a type. Given any $x, y : A$, the identity type $x = y$ is the type of equality proofs between x and y. Given, then, some $p, q : x = y$, we can talk about the type $p = q$ of equality proofs between the equality proofs. We can repeat this process *ad infinitum* meaning any type A induces an infinite tower of types of equality proofs. When we use the term *dimension*, we are referring to levels of this tower. A space containing no nontrivial homotopy above dimension n is called a homotopy n-type (or said to be of homotopy level n).

We then recursively define a predicate expressing that a given type has homotopy level n. The idea is that we increase the dimension by one (i.e. go from talking about a type to talking about its type of equality proofs) at each step.

[2]Attributed to Voevodsky in [14].

Definition 2.9 (Homotopy level)**.** We will say that the homotopy level of a type A is $n : \mathbb{N}$ if $\mathsf{isOfHLevel}\,(A, n)$ as inhabited:

$$\begin{aligned} \mathsf{isOfHLevel}\,(A, \mathsf{zero}) &:\equiv \mathsf{isContr}\,(A) \\ \mathsf{isOfHLevel}\,(A, \mathsf{suc}\,(n)) &:\equiv \prod_{(x\ y\ :\ A)} \mathsf{isOfHLevel}\,(x =_A y, n)\,. \end{aligned}$$

The hierarchy of n-types is upwards-closed as expressed in the following proposition.

Proposition 2.10. *Given any type A, the following type has an inhabitant:*

$$\prod_{(n\ :\ \mathbb{N})} \mathsf{isOfHLevel}\,(A, n) \to \mathsf{isOfHLevel}\,(A, \mathsf{suc}\,(n))\,.$$

Furthermore, $\sum$ and $\prod$ types respect h-levels.

Proposition 2.11. *Given an arbitrary type $A : \mathcal{U}_m$ and an A-indexed family of n-types $B : A \to \mathcal{U}_o$, $\prod_{(x\ :\ A)} B(x)$ is an n-type. In formal terms:*

$$\left(\prod_{(x\ :\ A)} \mathsf{isOfHLevel}\,(B(x), n)\right) \to \mathsf{isOfHLevel}\left(\prod_{(x\ :\ A)} B(x), n\right).$$

Proposition 2.12. *Given an n-type $A : \mathcal{U}_m$ and an A-indexed family $B : A \to \mathcal{U}_o$ of n-types, $\sum_{(x\ :\ A)} B(x)$ is an n-type. This is formally expressed by the type below.*

$$\mathsf{isOfHLevel}\,(A, n) \to \left(\prod_{(x\ :\ A)} \mathsf{isOfHLevel}\,(B(x), n)\right) \to \mathsf{isOfHLevel}\left(\sum_{(x\ :\ A)} B(x), n\right)$$

2.4.1 Propositions

The homotopy level of one is of special interest: it is the class of types that behave like logical propositions, in the sense that their proof structures are trivial; they can inhabited by at most one term. This is precisely the property we desire in those types that we view as expressing *properties*: we are interested, not in what they are inhabited by, but in whether they are inhabited or not.

Definition 2.13 (Proposition)**.** A type A is a proposition (sometimes disambiguated as an *h-proposition* if it has a homotopy level of one:

$$\mathsf{isProp}\,(A) :\equiv \mathsf{isOfHLevel}\,(A, 1)\,.$$

An equivalent and much more intuitive way of expressing propositionality is the following.

Definition 2.14 (Proposition (official))**.**

$$\mathsf{isProp}\,(A) :\equiv \prod_{(x\ y\ :\ A)} x =_A y.$$

Proposition 2.15. *Definitions 2.13 and 2.14 are equivalent.*

Due to this equivalence, we will use definitions 2.13 and 2.14 interchangeably. Which definition we are referring to should be clear in context.

We collect propositional types at level n in the type Ω_n.

Definition 2.16 (Ω)**.** Ω_n is the type of all propositional types at universe n:

$$\Omega_n \quad :\equiv \quad \sum_{(A\,:\,\mathcal{U}_n)} \mathsf{isProp}\,(A)\,.$$

When asserting that a proposition $A : \Omega_n$ is inhabited, we have to project out the first component to be completely precise. We will engage in the excusable notational abuse of denoting this projection by A itself. In the Agda formalisation, we use the notation of the `cubical` [28] library in which this is denoted `[ A ]`.

Once we introduce this delineation of the class of propositional types, we have to be careful about the distinction between *property* and *structure*. Ideally, types that are thought of as expressing propositions should always be propositional. For instance, the definition of equivalence we gave (in Defn. 2.3) has the crucial property that it is propositional. This would not always be the case [42] if we were to use the usual notion of equivalence as an invertible function.

Many types that are expected to behave like propositions behave naturally like propositions so it suffices to prove their propositionality; for those that do not, univalent type theory provides a mechanism, called *propositional truncation*, for forcing them to be propositional. A substantial component of the work presented in this thesis pertains to problems related to the use of this mechanism.

Now, let us state some simple facts about the class of propositional types. Observe the two corollaries, of propositions 2.11 and 2.12 respectively.

Proposition 2.17. *Given $A : \mathcal{U}_m$, $B : A \to \mathcal{U}_n$, the type $\prod_{(x\,:\,A)} B(x)$ is a proposition whenever $B(x)$ is proposition every $x : A$.*

Proposition 2.18. *Given types $A : \mathcal{U}_m$ and $B : A \to \mathcal{U}_n$, if A is a proposition and B is a family of propositions, then $\sum_{(x\,:\,A)} B(x)$ is a proposition.*

The type expressing that a given type A is a proposition, is itself a proposition.

Proposition 2.19. *Given $A : \mathcal{U}_n$, the type $\mathsf{isProp}\,(A)$ is propositional.*

Note also the following fact.

Proposition 2.20. *Given some type A, a family of propositions $B : A \to \Omega$, and two inhabitants $(x, p), (y, q) : \sum_{(a\,:\,A)} B(a)$, if $x = y$ then $(x, p) = (y, q)$.*

This can in fact be strengthened to an equivalence but this form will be sufficient for our purposes. Such a type $\sum_{(a\,:\,A)} B(a)$, where B is a family of propositions, can be thought of as a "subtype" of A: the restriction of A to its inhabitants that satisfy the *property* B.

As we view propositional types as embodying properties, it is natural to expect the right notion of equivalence between them to be logical equivalence.

Definition 2.21. Types $A : \mathcal{U}_m$ and $B : \mathcal{U}_n$ are logically equivalent (denoted $A \leftrightarrow B$) if the following type is inhabited:

$$A \leftrightarrow B \quad :\equiv \quad (A \to B) \times (B \to A).$$

We can indeed show that logically equivalent h-propositions are equivalent.

Proposition 2.22. *Given propositions $A : \mathcal{U}_m$, $B : \mathcal{U}_n$, if $A \leftrightarrow B$ then $A \simeq B$.*

As we mentioned before, Voevodsky's notion of type equivalence enjoys the crucial property that it is propositional. Let us consider the familiar notion of equivalence between types.

Definition 2.23 (Type isomorphism)**.** Given types A and B, the type of type isomorphisms between them is denoted $A \cong B$:

$$\begin{aligned} \mathsf{isIso}\,(f) \quad &:\equiv \quad \sum_{(g\,:\,B\to A)} g \circ f \sim \mathsf{id}_A \times f \circ g \sim \mathsf{id}_B \\ A \cong B \quad &:\equiv \quad \sum_{(f\,:\,A\to B)} \mathsf{isIso}\,(f)\,. \end{aligned}$$

This would not be a viable definition of type equivalence in univalent type theory as it suffers the problem that it is not propositional: although the inverse a function is necessarily unique, the invertibility proofs of are not unique in the general case when we are dealing with n-types. Therefore, Definition 2.23 is logically equivalent but *not* equivalent to Definition 2.3.

Proposition 2.24. *Definitions 2.23 and 2.3 are logically equivalent.*

Notice, however, that this logical equivalence is an equivalence in the special case where the domain and the codomain are h-sets.

When we have some $f : A \to B$ such that $\mathsf{isEquiv}\,(f)$, we will speak of the inverse of f and denote this $\mathsf{inv}\,(f)$. This expression refers implicitly to this logical equivalence. The distinction between $A \simeq B$ and $A \cong B$ will be especially important in Section 3.4.

2.4.2 Sets

The other homotopy class of interest is the class of types whose proof structures are not necessarily trivial but the proof structures of their types of equality proofs are trivial. Put more simply, inhabitants of such types are equal to each other in *at most one way*. This is the class of types that are called sets.

Definition 2.25 (Set)**.** A type A is a set (disambiguated as h-set) if its homotopy level is two:

$$\mathsf{isSet}\,(A) \quad :\equiv \quad \mathsf{isOfHLevel}\,(A, 2)\,.$$

The structures we present in chapters 3 and 4 have underlying partially ordered *sets*. When implementing these in univalent type theory, they must be required to be h-sets. A more accurate name for a "poset" with a carrier set bearing higher dimensional homotopy structure would be ∞-*poset*.

Observe the following corollary of Proposition 2.10.

Proposition 2.26. *Every proposition is a set.*

We also note that the type Ω of h-propositions is an h-set.

Proposition 2.27. *Ω is an h-set.*

As we have done for the case of h-propositions, we consider the special cases of propositions 2.11 and 2.12 in the h-set case.

Proposition 2.28. *Given $A : \mathcal{U}_m$, $B : A \to \mathcal{U}_n$, the type $\prod_{(x\ :\ A)} B(x)$ is a set whenever $B(x)$ is a set for every $x : A$.*

Proposition 2.29. *Given types $A : \mathcal{U}_m$ and $B : A \to \mathcal{U}_n$, if A is a set and $B(a)$ is a set for every $a : A$, then $\sum_{(x\ :\ A)} B(x)$ is a set.*

Notice also that the type expressing that a given type is a set is a proposition.

Proposition 2.30. *Given any type A, the type $\mathsf{isSet}\,(A)$ is a proposition.*

We conclude this section by presenting a highly useful sufficient condition for setness. Let us write down the usual definition of a decidable type.

Definition 2.31. A type A is called decidable if it satisfies the law of excluded middle:

$$\mathsf{isDecidable}\,(A) \quad :\equiv \quad A + \neg A,$$

where $+$ denotes the binary sum type as in the HoTT Book [42].

A type is called discrete if its type of equality proofs is decidable.

Definition 2.32. A type A is called discrete if the following type is inhabited:

$$\mathsf{isDiscrete}\,(A) \quad :\equiv \quad \prod_{(x\ y\ :\ A)} \mathsf{isDecidable}\,(x =_A y)\,.$$

The following sufficient condition for setness is due to Michael Hedberg [23] and is known as Hedberg's theorem.

Theorem 2.33 (Hedberg)**.** *Every discrete type is a set.*

2.5 Powersets

Given a type A, we would like to talk about its powerset: the type of *properties* inhabitants of A might have. This is nothing but the type $A \to \Omega$. It is here that the property-structure distinction starts to become visible: one would have to use $A \to \mathcal{U}$ for this purpose in non-univalent type theory. Of course, Ω could be defined in non-univalent type theory as well, but non-univalent type theory is too restrictive for one to engage in sophisticated developments that maintain this distinction between structure and property. Most importantly, there are no methods for turning structure into property in non-univalent type theory meaning one would not be able to express many crucial inhabitants of $A \to \Omega$ as they would involve non-trivial proof structure.

We represent the subsets of inhabitants of a type using the powerset. This is just one such representation; we will present another common approach in Section 2.6.

Definition 2.34. Given a type $A : \mathcal{U}_n$, the *powerset* of A is the type of all predicates on A: $\mathcal{P}(A) :\equiv A \to \Omega_n$. We introduce a bit of syntactic sugar and write membership in a subset as: $x \in U :\equiv U(x)$.

Our use of the word "set" in "powerset" is justified by the following proposition.

Proposition 2.35. *Given* any *type* A, $\mathcal{P}(A)$ *is an h-set.*

Proof. By Proposition 2.28, suffices to show that the codomain, Ω, is a set. This is given by Proposition 2.27. □

We can define the usual inclusion order on the powerset of a given type.

Definition 2.36. Given a type $A : \mathcal{U}_n$ and subsets $U, V : \mathcal{P}(A)$, U is a subset of B if the following type is inhabited:

$$U \subseteq V \quad :\equiv \quad \prod_{(x\ :\ A)} x \in U \to x \in V.$$

The propositionality follows from Prop. 2.17.

Definition 2.37 (Full subset)**.** Given a type $A : \mathcal{U}_n$, the subset containing all inhabitants of A is defined as the constant function:

$$\top_A \quad :\equiv \quad \lambda\_.\ \mathsf{Unit}_n.$$

Notice that the definition of $\mathcal{P}$ requires the involved h-proposition to live in the same universe as A. It is for this reason that we generalise the Unit type to live in the universe level we provide it as an argument; we denote this using a subscript.

Definition 2.38. Given a type $A : \mathcal{U}_n$ and subsets $U, V : \mathcal{P}(A)$, the subset delineating those elements that are in *both* of U and V is defined as:

$$\begin{aligned} \_ \cap \_ \quad &: \quad \mathcal{P}(A) \to \mathcal{P}(A) \to \mathcal{P}(A) \\ U \cap V \quad &:\equiv \quad \lambda x.\ x \in U \times x \in V. \end{aligned}$$

We note that $x \in U \cap V$ is propositional by the propositionality of $x \in U$ and $x \in V$ combined via Proposition 2.18.

2.6 Families

In type theory, we have two common notions of "subset of inhabitants of a type". One of these is the powerset presented in Sec. 2.5. We now briefly summarise the other approach in which we view a subset of a type A as an A-valued function (on some domain).

Definition 2.39 (Family)**.** Given a type A, an I-indexed family of inhabitants of A is simply a function: $I \to A$. We collect the type of all families on A in the following type:

$$\mathsf{Fam}_o(A) \quad :\equiv \quad \sum_{(I\ :\ \mathcal{U}_o)} I \to A.$$

Notice that this is parameterised by a level o, being the level of the index type. Given a family $U \equiv (I, f) : \mathsf{Fam}_o(A)$, the evaluation of U at $i : I$ will be denoted U_i.

Notational clarification. An inhabitant of $\mathsf{Fam}_o\,(A)$ has the form (I, f) as defined in Defn. 2.39. However, we will use some more suggestive notation for pairs of this form. It is quite convenient to be able to talk about a family as though it were a concrete set, so we will use the notation $\{x_i \mid i \in I\}$ as a substitute for "I-indexed family whose ith projection is denoted x_i". Furthermore, we will talk quite often about join operators $\bigvee\_ : \mathsf{Fam}_o\,(A) \to A$. Instead of writing $\bigvee(I, \lambda i.\ e(i))$ for the application of a join operator to a family, we will use the more familiar notation:

$$\bigvee_i e. \tag{2.40}$$

What is meant by such notational sugar is expected to be obvious to the reader.

Definition 2.41 (Family membership)**.** Given a type A, some $x : A$, and a family $U \equiv (I, f)$, we say that x is a member of U if the fiber of f over x is inhabited:

$$x \in U \quad :\equiv \quad \mathsf{fiber}\,(f, x)\,.$$

Let us note at this point that, technically, we would like to work with families whose functions are injective since repeated elements in the image are not desirable. If we were to add this requirement, we could prove $x \in U$ to be propositional. To keep the presentation simple, however, we will not do this; this is not a fact we need for the work presented in this thesis.

Definition 2.42 (Image over a family)**.** Given a type A, some family $U \equiv (I, f)$ on A, and a function $g : A \to B$, the image of g over U is nothing but the family $(I, g \circ f)$. We will denote this $\{g\,(a) \mid a \in U\}$ or $\{g(f(i)) \mid i : I\}$.

Given an inhabitant U of the powerset of some type A, we can represent U as a family as defined below.

Definition 2.43 (Familification of a powerset)**.** Let A be a type and $U : \mathcal{P}\,(A)$. The family associated with U is the pair (I, f) where

$$\begin{aligned} I \quad &:\equiv \quad \sum_{(x\ :\ A)} x \in U \\ f \quad &:\equiv \quad \mathsf{pr}_1. \end{aligned}$$

As the representation we are working with will be clear to the reader from the context, we will engage in a form of notational abuse and will not explicitly denote that we are working with the familification of a powerset. In the Agda formalisation, this is denoted $\langle\!\langle U \rangle\!\rangle$.

2.7 Higher inductive types

In ordinary type theory, the inductive definition of a type amounts to a specification of its points. In univalent type theory, we are working not just with 2-types but with arbitrary n-types. Therefore, univalent type theory generalises the mechanism for the inductive generation of types by generalising points (0-paths) to n-paths. In

other words, it provides the means to define a type not just by specifying its points, but also the paths between the points, paths between those paths, and then paths between those and so on. Such an inductive type is called a *higher inductive type* (HIT for short).

Let us give an example of what an HIT definition might look like. The usual Unit type would be inductively defined by the specification of a single constructor:

$$\frac{}{\star : \mathsf{Unit}}$$

whose only equality is the trivial one: $\mathsf{refl}_\star : \star =_{\mathsf{Unit}} \star$. This means that Unit contains no nontrivial paths.

HITs allow us to insert nontrivial paths into a type. For instance, we can add a new equality of $\star$ to itself, that we will call loop; the resulting type is called the Circle.

$$\frac{}{\mathsf{base} : \mathsf{Circle}} \qquad \frac{}{\mathsf{loop} : \mathsf{base} = \mathsf{base}}$$

Of course, when generalising the machinery for defining inductive types in this way, we have to be careful about the corresponding *introduction* and *elimination* rules. The introduction rules are easy to deal with as they are given by the constructors. The elimination rules, however, are quite subtle and it is technically intricate to completely address the problem of extracting (dependent) eliminators from higher inductive definitions. Even in the HoTT Book [42, Sec. 6.2], this issue is not addressed in a completely technical way. Similarly, we will consider in the following section, a useful instance of an HIT and present its recursion principle. Hopefully, this should provide sufficient intuition for the reader; it certainly suffices for the purposes of our development.

HITs turn out to be immensely useful and in fact we will explain later that they are used in a crucial way in this thesis: the main construction of Chapter 4 would not have been possible without them (i.e. it is not known to us how it would have been possible without them).

2.8 Truncation

A particular HIT of interest will be the *propositional truncation* type. This will allow us to *forget* the proof structure of a type, or in other words, force a non-propositional type to be propositional by collapsing its proof structure.

This turns out to be quite convenient as it gives us a way of turning structure into property. Consider for instance two propositions $P, Q : \Omega$. We would like to be able to express their logical disjunction and we would like this to be a proposition itself. It would clearly not be propositional if we were to define their disjunction as the sum type $P + Q$ since inhabitants of sum types contain the information of which one of the summands they come from. For this to become propositional, we have to somehow *forget* this information; propositional truncation allows us to do precisely this.

We now give the definition of propositional truncation as an HIT.

Definition 2.44 (Propositional truncation)**.** Given a type A, its propositional truncation $\|A\|$ is an HIT given by one point constructor, $|\_|$, and one path constructor, trunc.

$$\frac{x : A}{|x| : \|A\|} \qquad \frac{x : \|A\| \qquad y : \|A\|}{\mathsf{trunc} : x = y}$$

The recursion principle for propositional truncation is the following: given any two types A and B, to show $\|A\| \to B$, it suffices to show $\mathsf{isProp}\,(B)$ and $A \to B$. Formally,

$$\mathsf{isProp}\,(B) \to (A \to B) \to \|A\| \to B.$$

The trunc rule is often called squash but we choose this name to prevent ambiguity as we will later introduce another HIT featuring a rule called squash.

Intuitively, why does the recursion principle have this form? To define a function $\mathbb{N} \to B$ on the natural numbers using their elimination principle, we need to construct two terms: (1) an inhabitant of B for the base case, and (2) a $\mathbb{N}$-indexed family of inhabitants of $B \to B$ for the inductive case; these correspond to the types of the (non-dependent) constructors $\mathsf{zero} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$. In the case of truncation, we have a (dependent) constructor of the form $\mathsf{trunc} : \prod_{(x\ y\ :\ \|A\|)} x = y$. Accordingly, in addition to some term of type $A \to B$ for the $|\_|$ constructor, we now have to require a $(B \times B)$-indexed family of equality proofs: for each $x, y : B$, a proof of $x = y$, which is none other than the requirement of B to be propositional.

Observe that this corresponds to the intuitive distinction between *property* and *structure*: we are allowed to use the inhabitant of a propositionally truncated type only if it will be used in the proof of another property i.e. only in a context where the structure will not become exposed.

When we use propositional truncation, we will not explicitly refer to these two constructors. Instead, we will simply mention that some $\|A\| \to B$ is enabled by the propositionality of B.

This definition of propositional truncation as an HIT brings us to the end of this chapter. The foundational notions we have collected so far suffice to carry out an investigation of formal topology in univalent foundations.

3

Frames

We present in this chapter the lattice-theoretic notion of a *frame*. In Chapter 1, we remarked that a frame is the algebra of a logic of finitely verifiable properties. Recall that a frame consists of the following:

- a set O of *opens*,
- a partial order $\_ \sqsubseteq \_ \subseteq O \times O$, corresponding to the set inclusion order of the open subsets,
- finite meets, and
- arbitrary joins.

In addition to these, we need a law to ensure the correct interplay between meets and joins. Suppose we have a set A and a family of sets $B_0, B_1, \cdots$. Consider the set:

$$A \cap (\bigcup_i B_i).$$

By set-theoretic reasoning, this is the same as:

$$\bigcup_i (A \cap B_i).$$

As we are trying to characterise the behaviour of open sets, without defining them as sets of points, we have to explicitly add this distributivity law into the definition of a frame. Put simply:

binary meets must distribute over arbitrary joins in a frame.

We now start presenting our formal development of frames. We start with partially ordered sets in Section 3.1. In Section 3.2, we present the definition of a frame. In Section 3.4, we discuss our proof that isomorphic frames are equal, which can be considered the hallmark of the fact that our development takes place in a univalent context. In sections 3.5 and 3.6, we prove two important lemmas in preparation for the chapter on formal topology (Chapter 4): (1) the set of downwards-closed subsets of a poset forms a frame and (2) given a nucleus (a technical notion to be introduced) on a frame, its set of fixed points is itself a frame.

3.1 Partially ordered sets

This section corresponds to the `Poset` module in the Agda development.

Definition 3.1 (Poset)**.** Given some $A : \mathcal{U}_m$, let $\mathsf{Order}_n(A) :\equiv A \to A \to \Omega_n$. A poset with carrier level m and relation level n is then defined as:

$$\mathsf{Poset}_{m,n} :\equiv \sum_{(A\,:\,\mathcal{U}_m)} \mathsf{PosetStr}_n(A)\,,$$

where

$$\begin{aligned}
\mathsf{PosetStr}_n(A) &:\equiv \sum_{(R\,:\,\mathsf{Order}_n(A))} \mathsf{PosetAx}(A,R) \\
\mathsf{PosetAx} &: \left(\sum_{(A\,:\,\mathcal{U}_m)} \mathsf{Order}_n(A)\right) \to \Omega \\
\mathsf{PosetAx}(A,R) &:\equiv \prod_{(x\,:\,A)} R(x,x) \\
&\quad \times \prod_{(x\ y\ z\,:\,A)} R(x,y) \to R(y,z) \to R(x,z) \\
&\quad \times \prod_{(x\ y\,:\,A)} R(x,y) \to R(y,x) \to x =_A y \\
&\quad \times \mathsf{isSet}(A)\,.
\end{aligned}$$

Propositionality of PosetAx follows by propositions 2.18, 2.17, and 2.30.

The function name PosetStr is intended to be an abbreviation of "poset structure", that is, an ordered structure subject to the axioms for a partial order. The predicate expressing that a given ordered structure satisfies these axioms is in turn abbreviated by the function name PosetAx.

Let us also note a minor deviation from the Agda formalisation here. The function PosetAx is curried in the Agda code. It is more conventional in traditional ("pen-and-paper") mathematics to work with uncurried functions, whereas the exact opposite is more conventional in formalised mathematics. Instead of importing the conventions of one into the other, we take the correspondence to be self-evident.

Given a poset P, we will refer to its relation as $\_ \sqsubseteq_P \_$ (in cases where there might be ambiguity) and the underlying set of P as $|P|$. Notice that the fourth component of PosetAx requires the carrier set to be an h-set (Defn. 2.25).

We can talk about the *downwards-closed subsets* of a given poset P: sets that include all elements below their elements. One way of reading this is as expressing the property of being closed under refinement or "collection of more information". Take a certain element $x : |P|$ that we view as a stage of information. For some other $y : |P|$, $y \sqsubseteq x$ expresses the idea that y is a *finer* stage of information: it approximates the result better meaning it confines it into a *narrower* range. Let U be a subset of $|P|$. The property that U is downwards-closed is then nothing but:

$$x \in U \to y \sqsubseteq x \to y \in U,$$

the intuitive reading of which is: U is closed under the reception of more information. In other words, regardless of how things unfold after x has been reached, the stage of information we end up at will still be in U. One can therefore say that U is itself like an observable property on $|P|$: given a sequence $x_0 \sqsupseteq x_1 \sqsupseteq \cdots$ of stages of information that eventually "hits" U, no further examination will be needed as soon as we find the first element that hits U.

Let us now formally summarise the notion of downwards-closure.

Definition 3.2 (Downwards-closed subset)**.** We first define a predicate expressing that a given subset $U : \mathcal{P}(|P|)$ of some poset P is downwards-closed:

$$\begin{aligned} \mathsf{isDownwardsClosed} &: \left(\sum_{(P\ :\ \mathsf{Poset})} \mathcal{P}(|P|)\right) \to \Omega \\ \mathsf{isDownwardsClosed}(P, U) &:\equiv \prod_{(x\ y\ :\ |P|)} x \in U \to y \sqsubseteq x \to y \in U. \end{aligned}$$

The propositionality follows by Prop. 2.17. We then define the type of downwards-closed subsets of a poset as:

$$\begin{aligned} \mathsf{DCSubset} &: \mathsf{Poset} \to \mathcal{U} \\ \mathsf{DCSubset}(P) &:\equiv \sum_{(U\ :\ \mathcal{P}(|P|))} \mathsf{isDownwardsClosed}(P, U). \end{aligned}$$

So far we have dealt with two notions of *observable property* throughout the development:

1. elements of a poset which we will view as opens when we get to frames, and
2. the notion of downwards-closed subset which expresses that a property of the poset of opens behaves like an observable property.

We will now start relating these two by showing that the set of downwards-closed subsets of a poset is itself a poset, and indeed, we will prove later (in Sec. 3.5) that it actually forms a frame meaning downwards-closed subsets satisfy our expectations from properties we view as observable.

Let us start by showing that $\mathsf{DCSubset}(P)$ is always a set.

Proposition 3.3. $\mathsf{DCSubset}(P)$ *is a set for every poset* P*.*

Proof. By Proposition 2.29, it suffices to show that $\mathcal{P}(|P|)$ is a set and

$$\mathsf{isDownwardsClosed}(P, U)$$

is a set for every $U : \mathcal{P}(|P|)$. The former holds by Proposition 2.35. For the latter, observe that every $\mathsf{isDownwardsClosed}(P, U)$ is a proposition by definition meaning it is also a set by Proposition 2.26. □

We can now proceed to construct the poset of downwards-closed subsets.

Proposition 3.4. *(Poset of downwards-closed subsets) Let P be a poset. The type* DCSubset (P) *forms a poset under the subset inclusion relation (Defn. 2.36).*

Proof. The fact that DCSubset (P) is a set is given by Proposition 3.3 so it suffices to show that the poset axioms are satisfied. Reflexivity and transitivity are immediate. For antisymmetry, let $(U, \_), (V, \_) : \mathsf{DCSubset}\,(P)$ and assume $U \subseteq V$, $V \subseteq U$. By Proposition 2.20, it suffices to show $U = V$ as downwards-closure is a proposition. By function extensionality (Prop. 2.8), it remains to be shown that for every $x : |P|$, $U(x) = V(x)$. Let $x : |P|$. By univalence (Axiom 1), we are done if can show $U(x) \simeq V(x)$ which must clearly be the case by Prop. 2.22 as we know $U(x) \leftrightarrow V(x)$. □

Observe that we have made use univalence and function extensionality here. If we were working in a non-univalent setting, we would already have to resort to postulates or start using setoids at this point.

3.1.1 Monotonic functions

Let us now write down the type of those functions that preserve the partial order structure of a poset: monotonic maps.

Definition 3.5 (Monotonic function)**.** Let P, Q be posets. A function $f : |P| \to |Q|$ is monotonic if the following type is inhabited:

$$\mathsf{isMonotonic}\,(f) \quad :\equiv \quad \prod_{(x\ y\ :\ |P|)} x \sqsubseteq_P y \to f(x) \sqsubseteq_Q f(y).$$

We collect the type of monotonic functions between posets P and Q in the following type:

$$P \to_{\mathrm{m}} Q \quad :\equiv \quad \sum_{(f\ :\ |P| \to |Q|)} \mathsf{isMonotonic}\,(f)\,.$$

For reasons that will be explained in Section 3.4, the Agda formalisation follows a slightly different approach to this definition: it starts by defining the notion of a morphism between ordered structures in general (i.e. Order) and then defines monotonic maps as a special case of that.

Definition 3.6 (Poset isomorphism)**.** An isomorphism between two posets is a monotonic function with a monotonic inverse. We will denote the type of isomorphisms between two posets P and Q by $P \cong_{\mathrm{p}} Q$.

3.2 Definition of a frame

We now proceed to define frames as discussed. The constructions presented here can be found in the `Frame` module in the Agda formalisation.

Definition 3.7 (Frame)**.** A frame structure on some type A consists of the following data: (1) a poset structure, (2) a top element, (3) a binary meet operation, and (4)

a join operation of arbitrary arity, which we define using families. We then define a raw frame structure with relation level n and index level o as:

$$\mathsf{RawFrameStr}_{n,o}\,(A) \quad :\equiv \quad \mathsf{PosetStr}_n\,(A) \times A \times (A \to A \to A) \times (\mathsf{Fam}_o\,(A) \to A).$$

The prefix "raw" is used to signify that it is the *structural* component of the definition: it is merely some data that may or may not satisfy the laws imposed upon it. Such a raw structure that obeys the following axioms is called a *frame*.

$$\begin{aligned}\mathsf{FrameAx}\left(\sqsubseteq, \top, \wedge, \bigvee\right) \quad :\equiv \quad & \mathsf{isTop}(\top) \times \mathsf{isGLB}(\wedge) \times \mathsf{isLUB}\left(\bigvee\right) \\ \times \quad & \mathsf{isDistr}\left(\wedge, \bigvee\right)\end{aligned}$$

where

$$\begin{aligned}\mathsf{isTop}(\top) \quad :\equiv \quad & \prod_{(x\ :\ A)} x \sqsubseteq \top \\ \mathsf{isGLB}(\wedge) \quad :\equiv \quad & \prod_{(x\ y\ :\ A)} (x \wedge y \sqsubseteq x) \times (x \wedge y \sqsubseteq y) \\ \times \quad & \prod_{(x\ y\ z\ :\ A)} (z \sqsubseteq x) \times (z \sqsubseteq y) \to z \sqsubseteq x \wedge y \\ \mathsf{isLUB}\left(\bigvee\right) \quad :\equiv \quad & \prod_{(U\ :\ \mathsf{Fam}_o(A))} \prod_{(x\ :\ A)} x \in U \to x \sqsubseteq \bigvee_i U_i \\ \times \quad & \prod_{(U\ :\ \mathsf{Fam}_o(A))} \prod_{(x\ :\ A)} \left(\prod_{(y\ :\ A)} y \in U \to y \sqsubseteq x\right) \to \bigvee_i U_i \sqsubseteq x \\ \mathsf{isDistr}(\wedge, \bigvee) \quad :\equiv \quad & \prod_{(U\ :\ \mathsf{Fam}_o(A))} \prod_{(x\ :\ A)} x \wedge \bigvee_i U_i =_A \bigvee_i (x \wedge U_i)\end{aligned}$$

A minor notational clarification: the scope of the $\prod$ types do not extend to the new line in the cases where the lines are separated by $\times$. We then define the type of frames as:

$$\mathsf{Frame}_{m,n,o} \quad :\equiv \quad \sum_{(A\ :\ \mathcal{U}_m)} \mathsf{FrameStr}\,(A)\,,$$

where

$$\mathsf{FrameStr}\,(A) \quad :\equiv \quad \sum_{(s\ :\ \mathsf{RawFrameStr}_{n,o}(A))} \mathsf{FrameAx}\,(s)\,.$$

We will use the notation $|F|$ for referring to the underlying set of a frame, as we do for posets. Similarly, we will refer to the underlying partial order as $\_ \sqsubseteq_F \_$, the join operator as $\bigvee^F$, and the meet operator as $\wedge_F$. We will colloquially refer to the underlying poset of a given frame F which we will denote $\mathsf{pos}\,(F)$.

Notice at this point the source of the predicativity problems: in the definition of isLUB, we are quantifying over *subsets*. We have a type that quantifies over other types, which in type theory, is a "large" type. This quickly gives rise to problems: if we were to work with this, we would be able to have only joins of families whose

index types live in $\mathcal{U}_0$ (see [33, 11] for details). This would be too restricted for developing (pointless) topology in type theory. For instance, in Chapter 5, we will present an important example of a pointless topology: the Cantor space. It would not be possible to define this directly as a frame as we could not prove that it is *complete* (i.e. has all joins).

Let us now note that FrameAx is propositional as one would expect.

Proposition 3.8. *For every raw frame structure* $(\sqsubseteq, \top, \wedge, \bigvee)$, $\mathsf{FrameAx}\,(\sqsubseteq, \top, \wedge, \bigvee)$ *is a proposition.*

Proof sketch. By Proposition 2.18, it suffices to show that each component is an h-prop. For isTop, isGLB, and isLUB this can be concluded by using Proposition 2.18 and Proposition 2.17. For isDistr, we use Proposition 2.17 followed by the fact that the underlying set of a poset is an h-set (by the definition of PosetAx from Definition 3.1). □

3.2.1 Frame homomorphisms

As we have just introduced a new structure, we shall define what it means for a function to respect it.

Definition 3.9 (Frame homomorphism)**.** Let F and G be frames with the same index level. A frame homomorphism from F to G is a monotonic map (Defn. 4.2) from the underlying poset of F to the underlying poset of G that preserves the top element, the meets, and the joins. Formally,

$$\begin{aligned}
\mathsf{isFrameHomo}\,(f) \quad :\equiv \quad & f(\top_F) = \top_G \\
\times \quad & \prod_{(x\ y\ :\ |F|)} f(x \wedge_F y) = f(x) \wedge_G f(y) \\
\times \quad & \prod_{(U\ :\ \mathsf{Fam}_o(|F|))} f(\bigvee^F U) = \bigvee^G \{f(x) \mid x \in U\}\,.
\end{aligned}$$

Observe that this is propositional by propositions 2.18, 2.17, and the fact that the carrier of G is an h-set. The type of frame homomorphisms between F and G is then just:

$$F \to_{\mathrm{f}} G \quad :\equiv \quad \sum_{(f\ :\ \mathsf{pos}(F) \to_{\mathrm{m}} \mathsf{pos}(G))} \mathsf{isFrameHomo}\,(f)\,.$$

Definition 3.10. A frame isomorphism is a frame homomorphism with an inverse that is also a frame homomorphism.

We could strengthen this definition so that it requires only the preservation of the underlying order. This would be equivalent since all poset isomorphisms preserve meets and joins.

Definition 3.11. An isomorphism between frames F and G is simply an isomorphism of their underlying posets.

We will consider notions of frame isomorphisms in detail in Section 3.4.

3.3 Some properties of frames

Proposition 3.12 (Meet commutativity)**.** *In any frame F, $x \wedge_F y = y \wedge_F x$ for all $x, y : |F|$.*

Proof. By antisymmetry, it suffices to show $x \wedge_F y \sqsubseteq_F y \wedge_F x$ and $y \wedge_F x \sqsubseteq x \wedge_F y$. This is direct since both of $x \wedge_F y$ and $y \wedge_F x$ are lower bounds of x and y. □

We will need the following important lemma when we get to the universal property.

Lemma 3.13 (Flattening lemma)**.** *Let $F : \mathsf{Frame}_{m,n,o}$ and let $f : \prod_{(a\ :\ A)} B(a) \to |F|$ for some $A : \mathcal{U}_o$ and $B : A \to \mathcal{U}_o$. The following equality holds:*

$$\bigvee_F \left\{ \bigvee_F \{ f(a,b) \mid b \in B(a) \} \mid a : A \right\} = \bigvee_F \left\{ f(a,b) \mid (a,b) : \sum_{(x\ :\ A)} B(x) \right\}.$$

We omit the proof of Lemma 3.13 as its content is not particularly interesting and involves a non-trivial amount of bureaucracy. It can be found in the Agda formalisation, in the `Frame` module; the constructed inhabitant is named `flatten`.

The following proposition will be useful when proving the equality of joins of two "logically equivalent" families. Suppose that we have two families U and V and we know that $x \in U \leftrightarrow x \in V$. We cannot infer from this that $U = V$, in the way we have defined families. We will, however, not need this result and it will be satisfactory for our purposes to prove merely that $\bigvee_i U_i = \bigvee_i V_i$.

Proposition 3.14. *Let F be a frame and U, V be two families over $|F|$. The following type is then inhabited:*

$$\left(\prod_{(x\ :\ |F|)} x \in U \leftrightarrow x \in V \right) \to \bigvee_i U_i = \bigvee_i V_i.$$

Proof sketch. Follows directly from the LUB property. □

The distributivity law we required in the definition of a frame applies only in cases where we have the join as the right-hand conjunct of a meet. Therefore we cannot simultaneously distribute the meet of two joins to get a join of meets. Of course, this does not matter thanks to Prop. 3.12, but it is a bit inconvenient to have to explicitly apply Prop. 3.12 for this purpose. We therefore record the following fact for simultaneously distributing a meet of two joins.

Proposition 3.15. *Let $F : \mathsf{Frame}_{m,n,o}$ and $U, V : \mathsf{Fam}_o(|F|)$. Call the index types of U and V, I and J, respectively. The following equality holds:*

$$\left(\bigvee_i U_i \right) \wedge \left(\bigvee_i V_i \right) = \bigvee \{ U_i \wedge V_j \mid (i,j) : I \times J \}.$$

Proof sketch. Using Prop. 3.14, apply the distributivity law, use Prop. 3.12 (commutativity), apply the distributivity law again, and then flatten (using Lemma 3.13). □

3.4 Univalence for frames

It is common practice in mathematics to regard two isomorphic structures as equal. However, this requires isomorphisms to be viewed as the appropriate notion of equality on structures, as neither set-theoretical nor (non-univalent) type-theoretical foundations support this informal practice. This introduces an unnecessary bifurcation of the notion of equality and has to be remedied by ad hoc methods such as quotienting. One of the remarkable features of univalent type theory is that it addresses this problem: in a univalent setting, we can actually prove that two isomorphic structures are equal.

As frames are the central objects of study of this thesis, we prove that isomorphic frames are equal. In fact, we prove that there is an equivalence between the type of isomorphisms between two frames and the type of equality proofs between them. For this, we make use of a *structure identity principle* (SIP for short): a description of the identity type between two structures in terms of (well-behaved) equivalences of the carrier types (as explained by Escardó [14]). The first published SIP was formulated by Coquand and Danielsson [10] and another one (due to Peter Aczel) is given in the HoTT Book [42].

We use the SIP due to Martín Escardó [14], as implemented in the `cubical` library [28] of Agda. Instead of going into the proof details, we provide a high-level overview of what this SIP requires and what exact results we have obtained by making use of it in the Agda formalisation.

We have specified the *structure* of a poset and a frame (in definitions 3.1 and 3.7) in two steps: (1) writing down a function of type $\mathcal{U}_m \to \mathcal{U}_n$, expressing what it means for a type to have a certain structure, and (2) describing the isomorphisms of such structures (by defining what it means for a function to preserve the structure defined in (1)).

The result of Escardó [14] applies to a general notion of structure, as specified by (1) and (2). This requires us to describe the conditions under which a notion of isomorphism of structures is well-behaved. Such a notion of structure that is equipped with a well-behaved notion of isomorphism is called a *standard notion of structure* by Escardó [14].

When is a notion of isomorphism well-behaved? Let $S : \mathcal{U}_m \to \mathcal{U}_n$ be a structure, and let ι be a function expressing that a given equivalence between types A and B, equipped with S-structures s_A and s_B, preserves these structures. The type of ι looks like the following:

$$\iota : \left(\sum_{((A,s_A) \,:\, \sum_{(X \,:\, \mathcal{U}_m)} S(X))} \; \sum_{((B,s_B) \,:\, \sum_{(X \,:\, \mathcal{U}_m)} S(X))} A \simeq B \right) \to \mathcal{U}_o$$

In the case of posets, for instance, we pick $S :\equiv \mathsf{PosetStr}_k$ which has type:

$$\mathsf{PosetStr}_k : \mathcal{U}_m \to \mathcal{U}_{\max(m,k+1)},$$

and define ι as:

$$\iota(P, Q, (f, \_)) \quad :\equiv \quad \mathsf{isMonotonic}\,(f) \times \mathsf{isMonotonic}\,(\mathsf{inv}\,(f))\,.$$

This delineates those type equivalences between $|P|$ and $|Q|$ that have the virtue of preserving the poset structures. We call such equivalences *monotonic equivalences* and denote the type of monotonic equivalences between $|P|$ and $|Q|$ by $P \simeq_{\mathrm{p}} Q$.

Definition 3.16 (Monotonic equivalence)**.**

$$P \simeq_{\mathrm{p}} Q \quad :\equiv \sum_{(e \,:\, |P| \simeq |Q|)} \iota(P, Q, e)$$

In the general case, we require ι to satisfy two conditions to make sure it is a well-behaved characterisation of S-homomorphic equivalences.

1. The identity equivalence meets the ι criterion: for every

$$(A, s) : \sum_{(X \,:\, \mathcal{U})} S(X),$$

   $\iota((A, s), (A, s), \mathsf{idEqv}\,(A))$ is inhabited.

2. The map of type

$$s = t \to \iota((A, s), (A, t), \mathsf{idEqv}\,(A)),$$

   that one can define for any type A, S-structures s and t, and ι satisfying (1), is an equivalence.

Given some S, we say that it forms a *standard notion of structure* and write $\mathsf{SNS}\,(S)$ if there exists an ι satisfying (1) and (2).

Let us now state exactly what result is given by the SIP [14, 28].

Definition 3.17 (Structure identity principle)**.** Let $S : \mathcal{U}_m \to \mathcal{U}_n$ be a structure and let $A, B : \mathcal{U}_m$ be two types equipped with S-structures s and t. If S is standard in the sense that there is some ι witnessing that it forms an SNS (i.e. satisfies (1) and (2)), the type of ι-equivalences between (A, s) and (B, t) is equivalent to the identity type between them. Formally,

$$\prod_{(M\ N \,:\, \sum_{(A \,:\, \mathcal{U}_m)} S(A))} (M = N) \simeq (M \simeq_{\iota} N),$$

where

$$M \simeq_{\iota} N \quad :\equiv \sum_{(e \,:\, |M| \simeq |N|)} \iota(M, N, e).$$

How does univalence allow us to prove this equivalence? Although the technical details are not relevant for the rest of the development, let us briefly explain. Let A be a type, $B : A \to \mathcal{U}$, a family of types, and let

$$(x, p), (y, q) : \sum_{(x \,:\, A)} B(x).$$

We can then prove that the identity type $(x, p) = (y, q)$ is equivalent to the product of $x = y$ and $p = q$. Of course, we need to transport p or q by a given proof $r : x = y$

of equality to even be able to write this down, but the key idea is that the type of identity proofs between two inhabitants of a $\sum$ type is equivalent to the product of the identity types of the multiplicands.

Let S be a standard notion of structure, witnessed by some ι, and let

$$(A, s), (B, t) : \sum_{(A\ :\ \mathcal{U})} S(A).$$

Suppose that $(A, s) \simeq_\iota (B, t)$. We already have an equivalence between $A \simeq B$ meaning we have an inhabitant p of the identity type $A = B$ by univalence. What is then needed is an inhabitant of the identity type between $s = t$ (modulo transportation by p). This comes from the ι-equivalence between s and t which is ensured by SNS condition (2) to be strong enough to give us this result. At this point, it might not be clear how a notion of structure is proven to satisfy (2). We will explain how frames are proven to form an SNS soon which we hope will clarify this.

Let us now explain how we use the SIP for frames before explaining how they form an SNS. The situation is quite similar to the case of posets; frames form an SNS with $S :\equiv \mathsf{FrameStr}_{n,o}$ and some ι expressing the notion of an equivalence being homomorphic by requiring both of its directions to be frame homomorphisms as defined in Defn. 3.9. We will call such equivalences *frame-homomorphic equivalences* and will denote the type of frame-homomorphic equivalences between frames F and G by $F \simeq_{\mathrm{f}} G$

Definition 3.18 (Frame-homomorphic equivalence)**.** Let $(f, p) : A \simeq B$ be an equivalence between types A and B that are equipped with frame structures s_A and s_B. (f, p) is called frame-homomorphic if both f and $\mathsf{inv}\,(f)$ are frame homomorphisms. We denote this by $\mathsf{isFrameHomoEqv}\,((A, s_A), (B, s_B), (f, p))$, and collect all frame-homomorphic equivalences in the type:

$$F \simeq_{\mathrm{f}} G \quad :\equiv \quad \sum_{(e\ :\ |F| \simeq |G|)} \mathsf{isFrameHomoEqv}\,(F, G, e)\,.$$

We noted in Chapter 2 that the type of type isomorphisms is equivalent to the type of type equivalences in the special case when the types are h-sets. As we are working with algebraic structures that have carrier h-sets, the notions of frame-homomorphic and monotonic equivalences are not precisely what we want; we should instead use the familiar notion of categorical isomorphism, as we gave in definitions 3.6 and 3.10.

In the case of posets, we prove the equivalence of $P \simeq_{\mathrm{p}} Q$ with $P \cong_{\mathrm{p}} Q$ for any two posets P and Q. The final result then follows from this combined with the equivalence given by the SIP:

$$(P \cong_{\mathrm{p}} Q) \simeq (P \simeq_{\mathrm{p}} Q) \simeq (P = Q).$$

In the case of frames, the situation is similar: we have that $F \simeq_{\mathrm{f}} G$ is equivalent to $F \cong_{\mathrm{f}} G$ for any two frames F and G. However, this result could be strengthened as the notion of frame isomorphism does not even need to require the preservation of frame data (as remarked in Defn. 3.11); simply preserving the partial order suffices.

It is, however, quite convenient to carry out the SNS proof with a notion of frame homomorphism that explicitly requires the preservation of frame data. It is for this reason that, in the Agda formalisation, we first obtain the result (using the SIP)

$$(F \simeq_{\mathrm{f}} G) \simeq (F \equiv G), \tag{3.19}$$

and after this:

$$(\mathsf{pos}\,(F) \cong_{\mathrm{p}} \mathsf{pos}\,(G)) \simeq (F \simeq_{\mathrm{f}} G). \tag{3.20}$$

The final result,

$$(\mathsf{pos}\,(F) \cong_{\mathrm{p}} \mathsf{pos}\,(G)) \simeq (F \equiv G), \tag{3.21}$$

is then just a combination of (3.19) and (3.20)

As this result requires a high amount of bureaucracy (such as multiple notions of isomorphism), the reader might want to look at its completely formal form as constructed in the Agda formalisation. All of these results that involve frames live in the `Frame` module. The term witnessing the truth of (3.19) is called `≃f≃≡` whilst the one for (3.20) is called `≃f≃≅p`. The witness of the main result (3.21) is called `≅p≃≡`.

The result given in (3.21) is remarkable from the perspective of formal topology. In Chapter 4, we will present a method for generating a frame from a formal topology, and then prove that it is correct in the sense that the frame generated following this method satisfies a certain universal property. One could likely prove (although we have not done this) that this universal property characterises the generated frame uniquely *up to isomorphism*. A direct corollary of (3.21) is that *isomorphic frames are equal* so we can now drop the "up to isomorphism" and talk about the universal property characterising the generated frame *uniquely*.

We conclude this section by briefly explaining how it is that frames form an SNS. The first requirement that the identity equivalence be a frame-homomorphism is direct. We then have to prove the second requirement: let A be a type with two frame structures s and t; we have to show that the map

$$s = t \to \mathsf{isFrameHomoEqv}\,((A, s), (A, t), \mathsf{idEqv})$$

is an equivalence. This is achieved by defining the inverse

$$\mathsf{isFrameHomoEqv}\,((A, s), (A, t), \mathsf{idEqv}) \to s = t$$

using the fact that the identity function is a frame-homomorphic equivalence. This fact gives us, for instance, directly that $\top_s = \top_t$, and that

$$\prod_{(x\ y\ :\ A)} x \wedge_s y = x \wedge_t y$$

from which one can conclude, by function extensionality, that $\wedge_s = \wedge_t$. In other words, the fact that the identity equivalence is frame-homomorphic gives us exactly that the structures s and t can be mapped to each other without loss of information.

Once this map has been defined, we prove that it is a section and a retraction, which follows from various h-level theorems that we have proven in the Agda formalisation but not here. These include the fact that `RawFrameStr` is an h-set, being a frame homomorphism is an h-prop and so on. These are not particularly interesting except for this result and we hence omit them.

3.5 Frames of downwards-closed subsets

We have shown, in Proposition 3.4, how to construct the poset of downwards-closed subsets of a given poset. We will now show that this poset forms a *frame* with subset intersection as the meet and subset union as the join. The latter has not been defined yet.

Theorem 3.22. *Given a poset P, the poset of downwards-closed subsets of P (i.e., $\mathsf{DCSubset}\,(|P|)$) (as constructed in Prop. 3.4), is a frame.*

Proof. The top element and the meet operation are just those of the subset inclusion order: $\top_{|P|}$ and $\cap$ (definitions 2.37 and 2.38). The join operation is defined as follows: let U be a family of downwards-closed subsets with index set I;

$$\bigvee_i U_i \quad :\equiv \quad \lambda x. \left\| \sum_{(i\,:\,I)} x \in U_i \right\| \qquad \text{(using truncation as given in Defn. 2.44).}$$

$\top_{|P|}$ and $\cap$ are propositional by construction whereas $\bigvee$ is not and is hence forced to be propositional using truncation. Downwards-closure and the LUB/GLB properties are easy to verify. For the distributivity law, let U be a downwards-closed subset and $\{V_i \mid i : I\}$, a family of downward-closed subsets. We must show

$$\begin{aligned} U \cap \bigvee_i V_i \quad &\equiv \quad U \cap \left(\lambda x. \left\| \sum_{(i\,:\,I)} x \epsilon V_i \right\| \right) \\ &= \quad \bigvee_i (U \cap V_i) \\ &\equiv \quad \lambda x. \left\| \sum_{(i\,:\,I)} x \in (U \cap V_i) \right\|. \end{aligned}$$

This follows easily by antisymmetry combined with the induction principle of truncation (Defn. 2.44), the use of which is made possible by the propositionality of both sides of the equality. □

We will denote the frame of downwards-closed subsets of a poset P by $P\!\downarrow$.

3.6 Nuclei and their fixed points

To prepare for formal topology, we will now define a technical notion called a *nucleus*. Simply put, a nucleus on a frame is a meet-preserving, idempotent monad on the frame when it is viewed as a category. Nuclei describe quotient frames of a frame, which one views as subspaces of the space corresponding to that frame. They are presented in standard treatments of locale theory such as Johnstone [25, Sec. II.2].

As a brief digression, let us note that locales can be viewed as special kinds of toposes, called *localic* toposes. Nuclei are special cases of what are known as Lawvere-Tierney topologies [27, 30] in the general case of toposes.

The reason we are interested in nuclei is that in Chapter 4 we will be focusing on a particular nucleus on the frame of downward-closed subsets: the covering relation. It is this nucleus that will allow us to describe a topology by letting us specify "laws" that are expected to hold in the resulting frame. This is a form of the familiar method of *presenting by generators and relations* from algebra. From a given formal topology, we will *freely* generate a frame and then impose on it the relation given by this relation of covering.

The development presented in this section corresponds to the `Nucleus` module in the Agda formalisation.

Definition 3.23 (Nucleus)**.** Let $F : \mathsf{Frame}$ be a frame and $j : |F| \to |F|$, an endofunction on it. We say that j is *nuclear* if the following condition holds:

$$\begin{aligned}
\mathsf{isNuclear} &: (|F| \to |F|) \to \Omega \\
\mathsf{isNuclear}\,(j) &:\equiv \prod_{(x\ y\ :\ |F|)} j(x \wedge y) = j(x) \wedge j(y) \\
&\times \prod_{(x\ :\ |F|)} x \sqsubseteq j(x) \\
&\times \prod_{(x\ :\ |F|)} j(j(x)) \sqsubseteq j(x).
\end{aligned}$$

The propositionality follows by propositions 2.18 and 2.17, and the fact that the carrier type is an h-set (by the definition of PosetAx from Defn. 3.1). The type of nuclei is then just the $\sum$ type collecting all nuclear endofunctions on a frame:

$$\mathsf{Nucleus} :\equiv \sum_{(j\ :\ |F| \to |F|)} \mathsf{isNuclear}\,(j)\,.$$

In a context involving a nucleus, we will simply refer to these three nuclearity properties as N_0, N_1, and N_2.

Notice that every nucleus preserves the top element.

Proposition 3.24. *Let F be a frame and $j : |F| \to |F|$ a nucleus on it. $j(\top_F) = \top_F$.*

Proof. Clearly, $j(\top_F) \sqsubseteq \top_F$ and $\top_F \sqsubseteq j(\top_F)$ by N_1. We are done by antisymmetry. □

Observe also that every nucleus is monotonic (given in Prop. 3.26).

Lemma 3.25. *$x \sqsubseteq y$ if and only if $x = x \wedge y$ in any frame.*

Proof. Let F be a frame and let $x, y : |F|$. Suppose that $x \sqsubseteq y$. We have $x \wedge y \sqsubseteq x$ so it suffices by antisymmetry to show $x \sqsubseteq x \wedge y$. This holds as well since $x \wedge y$ is greater than any other lower bound and x is a lower bound of x and y: $x \sqsubseteq x$ and $x \sqsubseteq y$. For the other direction, suppose that $x = x \wedge y$. We have $x \sqsubseteq x \wedge y \sqsubseteq y$. □

Proposition 3.26. *Every nucleus is monotonic.*

Proof. Let F be a frame and $j : |F| \to |F|$, a nucleus on it. Let $x, y : |F|$ and suppose $x \sqsubseteq y$. It can be observed that $j(x) \sqsubseteq j(y)$ is the case as follows:

$$\begin{array}{lcll} j(x) & = & j(x \wedge y) & [\text{Lemma } 3.25] \\ & = & j(x) \wedge j(y) & [N_0] \\ & \sqsubseteq & j(y) & [j(x) \wedge j(y) \text{ is a lower bound}]. \end{array}$$

□

Given a nucleus $j : |F| \to |F|$ on a frame F, we will be interested in those inhabitants of $|F|$ that are j-closed, that is, fixed points of j. Starting with a frame F, its subset consisting of j-closed elements (for some nucleus j) is itself a frame.

Let us start working towards a proof of this by constructing the underlying poset of this frame.

Proposition 3.27. *Given a frame F and a nucleus $j : |F| \to |F|$, the type*

$$\sum_{(x \,:\, |F|)} j(x) = x$$

of j-closed elements forms a poset under the ordering $\sqsubseteq_F$.

Proof sketch. The proof amounts to forgetting the information of being a fixed point. For antisymmetry, we use Proposition 2.20 along with the fact that the carrier set is an h-set (by the definition of PosetAx from Defn. 3.1).

Notice that $|F|$ is an h-set by the definition of a poset meaning $j(x) = x$ is a proposition for every $x : |F|$. Since every proposition is a set (Prop. 2.26), it follows that $\sum_{(x \,:\, |F|)} j(x) = x$ is a set by Prop. 2.18. □

The Agda construction corresponding to Prop. 3.27 can be found in the `Nucleus` module under the name `fix-pos`.

Now, we are ready to prove the main theorem of this section: this poset of j-closed elements of a frame is itself a frame. The proof we present is an adaptation of Johnstone's proof [25, II.2.2, pg. 49] to the type-theoretical setting. The corresponding Agda proof can be found in the `Nucleus` module under the name `fix`.

Theorem 3.28. *The set of fixed points for a nucleus j on some frame F forms a frame, with the meet operator $(\_ \wedge \_)$ and the top element $(\top)$ taken directly from F, and the join defined as:*

$$\bigvee_i x_i \quad :\equiv \quad j\left(\bigvee_i^F x_i\right).$$

Proof. We start by showing that the meets and the top element respect the fixed point property. $\top$ is a fixed point by Prop. 3.24. For the meet operation, let $x, y : |F|$ such that $j(x) = x$ and $j(y) = y$. We have:

$$\begin{array}{ll} j(x \wedge y) = j(x) \wedge j(y) & [N_0] \\ \quad = x \wedge y & [j(x) = x \text{ and } j(y) = y]. \end{array}$$

We now show that the join operator, defined by applying j on the join of F, actually yields a LUB. Let $\{x_i \mid i \in I\}$ be a family of j-closed elements and let $i : I$. We have

$$x_i \sqsubseteq \bigvee_k^F x_k \sqsubseteq j\left(\bigvee_k^F x_k\right)$$

by N_1 meaning it is an upper bound. To see that it is the least such, let u be some other upper bound of $\{x_i \mid i \in I\}$ such that $j(u) = u$. We need to show that $j\left(\bigvee_k^F x_k\right) \sqsubseteq u$. Since $u = j(u)$ it suffices by the monotonicity of j (Prop. 3.26) to show $\bigvee_i^F x_i \sqsubseteq u$. We are done since u is an upper bound of $\{x_i \mid i \in I\}$.

It remains to be shown that the infinite distributivity law is satisfied. Let $x : |F|$ such that $j(x) = x$ and let $\{y_i \mid i \in I\}$ be a family. By Prop. 2.20, the proofs fixed-point-ness are irrelevant as the carrier is an h-set. The result then follows as:

$$\begin{aligned}
x \wedge \bigvee_i y_i &\equiv x \wedge j\left(\bigvee_i^F y_i\right) && [x = j(x)] \\
&= j(x) \wedge j\left(\bigvee_i^F y_i\right) && [N_0] \\
&= j\left(x \wedge \bigvee_i^F y_i\right) && [\text{distributivity of } F] \\
&= j\left(\bigvee_i^F x \wedge y_i\right) \\
&\equiv \bigvee_i x \wedge y_i.
\end{aligned}$$

□

Given a frame F, and a nucleus j on it, we will refer to the frame of j-closed elements as $\mathsf{fix}\,(F, j)$.

Remark on formalisation. The corresponding function in the Agda formalisation is called `fix`.

In Chapter 4, we will make use of nuclei to present a frame from a certain nucleus on the frame of downwards-closed subsets of a poset.

4

Formal topology

We remarked that the motivation for pointless topology is to attain a constructive understanding of topology, and that this is a prerequisite for being able to express topology in type theory in a natural way, that is, without postulating classical axioms. The task of making type-theoretical sense of topology presents another challenge: we must be able to develop our results in a completely predicative way as well. To address this, we will use formal topologies which give us a way of *presenting* frames in a predicative way. Instead of working with frames directly, we will work with formal topologies from which the frames are generated. As the motivation for formal topology arises naturally when attempting to carry out topology in type theory, it is fair to say that formal topology is *topology, construed type-theoretically*. This was aptly pointed out by Giovanni Sambin [34]:

> What is formal topology? A good approximation to the correct answer is: formal topology is topology as developed in (Martin-Löf's) type theory.

Our definition of a formal topology will make use of the notion of an interaction system, first formulated by Petersson and Synek [32]. In formulating this notion, the motivation of Petersson and Synek was to generalise Martin-Löf's W types to be able to accommodate mutual induction. Other names for interaction systems include *tree set constructors* (which is the name used by Petersson and Synek [32]), and *indexed containers* [4]. The idea of doing formal topology with interaction systems is due to Coquand [9] who was inspired by Dragalin [13].

This chapter corresponds to several modules in the Agda formalisation. Sections 4.1 and 4.2 correspond, respectively, to modules `FormalTopology` and `Cover`. Sections 4.3 and 4.4 correspond to the `CoverFormsNucleus` module. Section 4.5 corresponds to the `UniversalProperty` module.

4.1 Interaction systems

The fundamental idea of an interaction system is simple. Consider the progression of a two-player game from the perspective of one of the two players. First, there is a type of *game states*; call it A:

$$A : \mathcal{U}.$$

At each state $a : A$ of the game, there are certain moves the player can take. Formally, this is a type parameterised by the type of game states:

$$B : A \to \mathcal{U}.$$

Table 4.1: Peter Hancock's [21] table (with notational modifications) providing a list of examples of interaction systems.

$a : A$	$b : B(a)$	$c : C(a,b)$	$d(a,b,c) : A$
sort	constructor	selector	component sort
statement	inference	premise	premise statement
neighbourhood	partition	part	new neighbourhood
game	attack	defence	new state
interrogation	question	answer	new state
interface	call	return	new state
universe	observation	reading	new state
knowledge	experience	result	new state
dialogue	thesis	antithesis	synthesis

Furthermore, for every move b the player may take at state a, we can speak of the type of counter-moves that the opponent may take in response. Formally, this is expressed as a function:

$$C : \prod_{(a\ :\ A)} B(a) \to \mathcal{U}.$$

Finally, given a counter-move c in response to a certain move b at some state a, we proceed to a new game state. This is formally expressed by a function:

$$d : \prod_{(a\ :\ A)} \prod_{(b\ :\ B(a))} C(a,b) \to A.$$

In four pieces, namely (A, B, C, d), we characterise structures that involve some kind of *interaction*. Even though the game analogy is useful, interaction systems are more general than games: they express anything that can be viewed as a *dialogue* i.e. involving two parties interacting with each other. Hancock [21] provides a table of various structures that can be viewed as interaction systems. We reproduce this (with minor notational modifications) in Table 4.1.

To define the notion of a formal topology, we will require the type A of states to be not just a set, but a poset. The idea is the same as in the case of frames: we would like to view these states as stages of information ranked with respect to how refined they are; in the case of a formal topology, however, these stages of information are *basic* in the sense of basis for a topological space. In the standard terminology of formal topology, this is called a set of *basic opens* [11, 31, 17]. In addition, we will expect such a poset equipped with an interaction system to satisfy the following two properties:

1. the **monotonicity** property: for every state $a : A$, move $b : B(a)$ at state a, and counter-move $c : C(a,b)$ in response to b, $d(a,b,c) \sqsubseteq a$.

2. the **simulation** property which states that at any state, we can simulate the previous states (in a sense that we will explain in detail later).

Once we have imposed the requirement that the set of stages be a poset, it makes sense to use some more suggestive terminology for referring to the components of

an interaction system due to Per Martin-Löf[1,2] (though similar notions are common in the literature; see, for instance, [20]):

- A: a type of stages of *knowledge*,
- B: a type of *experiments* that one can perform at a certain stage of knowledge a,
- C: a type of *outcomes* of an experiment $b : B(a)$ at some stage of knowledge a, and
- d: a function that expresses the act of *revising* one's knowledge state based on the outcome of an experiment performed at a stage of knowledge.

Once we adopt this view, what the monotonicity property says becomes much clearer: if we perform an experiment b while we have knowledge a and revise our knowledge based on some outcome c of b, the new state of knowledge we arrive at must contain at least as much information as does a.

Now, to proceed towards the definition of a formal topology, let us first formally define interaction systems.

Definition 4.1 (Interaction system)**.** For simplicity, we will require all types to be at the same level and omit the level in the notation.

$$\mathsf{IntrStr}\,(A) :\equiv \sum_{(B\,:\,A\to\mathcal{U})}\ \sum_{(C\,:\,\prod_{(a\,:\,A)} B(a)\to\mathcal{U})}\ \prod_{(a\,:\,A)}\ \prod_{(b\,:\,B(a))} C(a,b)\to A$$

$$\mathsf{IntrSys} :\equiv \sum_{(A\,:\,\mathcal{U})} \mathsf{IntrStr}\,(A)$$

Given an interaction system $\mathcal{J}$, we will refer to its components as $A_{\mathcal{J}}$, $B_{\mathcal{J}}$, $C_{\mathcal{J}}$, and $d_{\mathcal{J}}$ in contexts where the possibility of ambiguity is present.

Remark on formalisation. In the Agda formalisation, A, B, C, and d are called respectively `state`, `action`, `reaction`, and `δ`. Furthermore, arguments that can be inferred have been made implicit. For `δ`, for instance, the stage and the experiment can be inferred from the outcome so, given an outcome $c : C(a, b)$, we write `δ c` in Agda for what we express here as $d(a, b, c)$.

Given an interaction system $\mathcal{J}$, the monotonicity property is then formally expressed as given in the following definition.

[1] Attributed to Martin-Löf by Coquand [9, pg. 2].

[2] Although this terminology is due to Martin-Löf, it is the present author's opinion that it is particularly sensible to use this terminology in the context where we are imposing an ordering on the states that satisfies the monotonicity property. This terminology is freely used for *all* interaction systems in the literature (as exemplified by Table 4.1 as well). Our justification is that it makes no sense to call B a type of experiments if the experiments might be *taking away* existing knowledge i.e. not satisfying the monotonicity property.

Definition 4.2 (Monotonicity property of an interaction system)**.** An interaction system $\mathcal{I}$, whose type $A_{\mathcal{I}}$ of stages is equipped with a partial order, is said to have the monotonicity property if the following type is inhabited:

$$\begin{aligned} \mathsf{hasMono} &: \left(\sum_{(P\,:\,\mathsf{Poset})} \mathsf{IntrStr}\,(|P|)\right) \to \mathcal{U} \\ \mathsf{hasMono}\,(P,\mathcal{I}) &:\equiv \prod_{(a\,:\,A_{\mathcal{I}})} \prod_{(b\,:\,B_{\mathcal{I}}(a))} \prod_{(c\,:\,C_{\mathcal{I}}(a,b))} d(a,b,c) \sqsubseteq_P a. \end{aligned}$$

We have not yet fully explained the simulation property so let us now address that. We first provide its formal definition.

Definition 4.3 (Simulation property)**.** Given an interaction system $\mathcal{I}$, whose type $A_{\mathcal{I}}$ of stages is a poset, we will say that it satisfies the simulation property if the following type is inhabited:

$$\begin{aligned} &\mathsf{hasSim}\,(\mathcal{I},\sqsubseteq) :\equiv \\ &\quad \prod_{(a'\ a\,:\,A)} a' \sqsubseteq a \to \\ &\qquad \prod_{(b\,:\,B(a))} \sum_{(b'\,:\,B(a'))} \prod_{(c'\,:\,C(a',b'))} \sum_{(c\,:\,C(a,b))} d(a',b',c') \sqsubseteq d(a,b,c). \end{aligned}$$

What does this say intuitively? Let us use the terminology of "below" and "above" for referring to finer and coarser stages of information. The simulation property then says that at any stage, we can always find a counterpart to any experiment from a stage above—"counterpart" in the sense that *any* outcome of that experiment can be mapped to *some* outcome of the above experiment and that this experiment will lead to a stage below the one given by the outcome of the experiment above. In other words, as we choose to perform certain experiments and proceed to more refined stages, we do not lose the ability to perform the experiments we previously chose not to perform: there is always a corresponding experiment (b') that will get us to a stage that is at least as refined as the one we can reach from the less refined stage.

We are now ready to formally define the notion of a formal topology.

Definition 4.4 (Formal topology)**.** A *formal topology* is simply an interaction system on a poset P of stages that satisfies the monotonicity and simulation properties:

$$\mathsf{FT}_{m,n} :\equiv \sum_{(P\,:\,\mathsf{Poset}_{m,n})} \sum_{(\mathcal{I}\,:\,\mathsf{IntrStr}(|P|))} \mathsf{hasMono}\,(\mathcal{I},\sqsubseteq_P) \times \mathsf{hasSim}\,(\mathcal{I},\sqsubseteq_P)\,.$$

4.2 Cover relation

The real reason we are interested in formal topologies is that the structure they contain, along with the monotonicity and simulation properties, induces a well-behaved *cover relation*. This is a method going back to Johnstone's [25, pg. 57, II.2.11]

adaptation of the notion of a Grothendieck topology [19] to the context of locale theory, that was subsequently developed by Martin-Löf and Sambin [35]. The idea of defining the cover as an inductive type originated in the joint work of Coquand, Sambin, et al. [11].

The idea is as follows: given a set A that we view like a set of *basic opens*, we require a relation:

$$\_ \triangleleft \_ : A \to \mathcal{P}(A) \to \Omega$$

that is expected to pointlessly express the relation of being an *open cover* i.e., $x \triangleleft U$ iff U is an open cover of x: $x = \bigvee_i U_i$. In other words, we are specifying which basic opens can be expressed as the join of which others. The information contained by this relation is sufficient to generate the topology (i.e. the frame).

The original formulation of formal topology by Sambin suffers a particular problem: it is not known how to define, predicatively, the coproduct of two frames using it. This problem was solved by Coquand, Sambin et al. [11] by defining the cover relation inductively. We follow this approach in our development. In fact, as we will explain later, we will define this relation not just as an inductive type but as a *higher* inductive one.

We now proceed to define the cover relation on a given formal topology. Our presentation will recapitulate the progression of our development: we first explain our naive attempt, which we found out not to work in univalent type theory, and after that its remedied form (thanks to a (privately communicated) suggestion by Thierry Coquand) that circumvents this problem through the use of HITs.

Definition 4.5 (Naive cover relation)**.** Given a formal topology $\mathcal{F}$ on type A, and given $a : A$, $U : \mathcal{P}(A)$, the type $a \triangleleft U$ is inductively defined with the following two constructors.

$$\frac{a \in U}{a \triangleleft U}\ \textsf{dir} \qquad \frac{b : B(a) \qquad \prod_{(c\ :\ C(a,b))} d(a,b,c) \triangleleft U}{a \triangleleft U}\ \textsf{branch}$$

To emphasise the reading of these constructors as proof rules, we write them as rule names and refer to them as rules.

One way of reading $a \triangleleft U$ is as a relaxation of $a \in U$ to "it is eventually the case that $a \in U$" if one views the refining of a as an evolution over time. In other words, it might not be that $a \in U$ but once stage a has been reached, all paths that do not lead to U have been ruled out: regardless of what experiments are run, they will eventually lead to a stage in U.

Let us now turn our attention to the question of how to present a frame using the cover relation of a formal topology. The cover relation has the following type:

$$\_ \triangleleft \_ : A \to \mathcal{P}(A) \to \mathcal{U},$$

which we can flip to get

$$\_ \triangleright \_ : \mathcal{P}(A) \to A \to \mathcal{U}.$$

If only this had the result type Ω rather than $\mathcal{U}$, we would have been able to write it like:

$$\_ \triangleright \_ : \mathcal{P}(A) \to \mathcal{P}(A), \tag{4.6}$$

but this is not the case as the relation has been defined using more than one constructor: it can have multiple, distinct proofs. If (4.6) were the case, we could at least try restricting it to get an endofunction on the set of downwards-closed subsets:

$$\_\triangleright\_ : \mathsf{DCSubset}\,(A) \to \mathsf{DCSubset}\,(A)$$

which (we believe) could then be shown to be a nucleus. (4.6), however, is not the case so we first have to deal with this problem.

It is tempting to try to achieve this by truncating $\_\triangleleft\_$. When we do this, however, it becomes impossible (ostensibly) to prove the idempotence law for the purported nucleus $\triangleright : \mathsf{DCSubset}\,(A) \to \mathsf{DCSubset}\,(A)$. Why exactly is that? If we were to work with the truncated form of $\triangleleft$ (as defined in Defn. 4.5), the idempotence law would look like:

$$\|a \triangleleft \|\lambda x.\ x \triangleleft U\|\| \to \|a \triangleleft U\|\,.$$

In our attempt to prove this, we had to use a lemma saying given any a, and subsets U, V,

$$\text{if } \|a \triangleleft U\| \text{ and } \|a' \triangleleft V\| \text{ for every } a' \in U, \text{ then } \|a \triangleleft V\|.$$

It is perhaps possible to avoid using such a lemma, but we have not been able to achieve this; we therefore believe that such a lemma would be necessarily needed.

The dir case of this lemma is easily verified. The branch case, however, results in a situation where we are trying to show

$$\left\| \prod_{(c\ :\ C(a,b))} d(a,b,c) \triangleleft V \right\|$$

whereas all we get from the inductive hypothesis is

$$\prod_{(c\ :\ C(a,b))} \|d(a,b,c) \triangleleft V\|\,.$$

An inference of the former from the latter would most likely require the use of a classical reasoning principle looking a lot like the axiom of choice [42, pg. 119, Lemma 3.8.2]:

$$\left(\prod_{(x\ :\ A)} \|B(x)\| \right) \to \left\| \prod_{(x\ :\ A)} B(x) \right\|,$$

with the difference that neither A nor B are required to be h-sets. Notice at this point that this is not just not provable in type theory, but is in fact *provably false* [42, pg. 120, Lemma 3.8.5]!

We would like to clarify at this point that we have not proven the fact that the idempotence law is not provable without this (generalised) choice axiom: we have merely *failed* to prove it. We conjecture, however, that it is not provable. At the time of writing, no answer to this question has been found.

To remedy this, we will refrain from truncating the naive form of the cover relation, and we will instead add a path constructor that *squashes* the difference

between the dir and branch constructors. The observation that the choice principle is no longer needed when the covering relation is defined this way is due to Coquand. We now provide the revised form, repeating the rules from Defn. 4.5 for the sake of self-containment.

Definition 4.7 (Cover relation)**.** Given a formal topology $\mathcal{F}$ on type A, and given $a : A$, $U : \mathcal{P}(A)$, the type $a \lhd U$ is inductively defined with the following two constructors:

$$\frac{a \in U}{a \lhd U}\ \mathsf{dir} \qquad \frac{b : B(a) \qquad \prod_{(c\ :\ C(a,b))} d(a,b,c) \lhd U}{a \lhd U}\ \mathsf{branch}.$$

In addition to the constructors, the type $a \lhd U$ contains the following family of paths:

$$\frac{p : a \lhd U \qquad q : a \lhd U}{p = q}\ \mathsf{squash}.$$

Remark on formalisation. The definition of this cover relation can be found in the `Cover` module in the Agda formalisation and is denoted `◁`. Notice also that Defn. 4.5 is not present in the Agda code.

We note that a similar idea is employed in the construction of the Cauchy reals as a higher inductive-inductive type in the HoTT Book [42, Defn. 11.3.2]. There, a path constructor is used to avoid using the axiom of countable choice. Given that the topology of the real numbers is a prime example of a topology, it is perhaps not a coincidence that the need to use a similar trick arises when one sets out to investigate topology in univalent type theory. It is an interesting direction for further work to investigate the relation between these, for instance by defining the formal topology of the real numbers.

When trying to prove the idempotence law with this, we get from the inductive hypothesis a product $\prod_{(c\ :\ C(a,b))} d(a,b,c) \lhd V$ where $d(a,b,c) \lhd V$ is still "squashed", but this squashing is an integral part of $\_ \lhd \_$ rather than a truncation that is imposed extrinsically upon it. This is sufficient for circumventing the problem that would have required a form of choice, and allows us to successfully complete the idempotence proof. The type of $\rhd$ can be seen *directly* to be $\mathcal{P}(A) \to \mathcal{P}(A)$ by the existence of the squash constructor.

We can now restrict $\mathcal{P}(A)$ to $\mathsf{DCSubset}(A)$ since, given a downwards-closed subset $U : \mathcal{P}(A)$, the subset of elements that are covered by U (i.e. $\_ \lhd U$) is itself downwards-closed.

We now prove some crucial properties of the cover relation, starting with downwards-closure.

Proposition 4.8. *Given a formal topology $\mathcal{F}$, $a, a' : A_{\mathcal{F}}$ such that $a' \sqsubseteq a$, and a downwards-closed subset U of $A_{\mathcal{F}}$, if $a \lhd U$ then $a' \lhd U$.*

Proof. Let $a, a' : A$ such that $a' \sqsubseteq a$ and $a \lhd U$. We need to show $a' \lhd U$. Our proof proceeds by (higher) induction on the proof of $a \lhd U$.

- Case dir. It must be that $a \in U$ and hence, by the downwards-closure of U, $a' \in U$ meaning $a' \lhd U$ by dir.

- Case branch. We have some experiment b on a and a function

$$f : \prod_{(c \,:\, C(a,b))} d(a,b,c) \triangleleft U.$$

  Using the branch rule, we are done if we can exhibit some experiment $b' : B(a')$ along with a function

$$g : \prod_{(c' \,:\, C(a',b'))} d(a',b',c') \triangleleft U.$$

  Pick b' to be the experiment obtained by appealing to the simulation property (Defn. 4.3) with b. Let $c' : C(a',b')$. It remains to be shown that $d(a',b',c') \triangleleft U$. The simulation property gives us some outcome $c : C(a,b)$ such that $d(a',b',c') \sqsubseteq d(a,b,c)$. The result follows by appealing to the inductive hypothesis with proof $f(c) : d(a,b,c) \triangleleft U$ along with the fact that $d(a',b',c') \sqsubseteq d(a,b,c)$.

- Case squash. We are done by the squash rule.

□

Proposition 4.9. *Let $\mathcal{F}$ be a formal topology with cover relation $\triangleleft$, and let $U, V : \mathcal{P}(A_{\mathcal{F}})$ be subsets of $A_{\mathcal{F}}$ such that V is downwards-closed. For any $a : A_{\mathcal{F}}$, if $a \triangleleft U$ and $a \in V$, then $a \triangleleft U \cap V$.*

Proof. Let $a : A_{\mathcal{F}}$ such that $a \triangleleft U$ and $a \in V$. We proceed by induction on the proof of $a \triangleleft U$.

- Case dir. We have $a \in U$ and $a \in V$ so we are done by the dir rule.

- Case branch. We have some experiment $b : a$ such that $d(a,b,c) \triangleleft U$ for every outcome $c : C(a,b)$. We are done by the branch rule on b if we can show that $d(a,b,c) \triangleleft U \cap V$ for every $c : C(a,b)$. Let $c : C(a,b)$. Notice that $d(a,b,c) \sqsubseteq a$ by the monotonicity property (Defn. 4.2) and hence by the downwards-closure of V, we have $d(a,b,c) \in V$. This means that we have $d(a,b,c) \triangleleft U$ and $d(a,b,c) \in V$ so we are done by the inductive hypothesis.

- Case squash. Immediate by the squash rule.

□

Proposition 4.10. *Given a formal topology $\mathcal{F}$, $a, a' : A_{\mathcal{F}}$ such that $a' \sqsubseteq a$, and downwards-closed subsets U and V of $A_{\mathcal{F}}$, if $a \triangleleft U$ and $a' \triangleleft V$ then $a' \triangleleft V \cap U$.*

Proof. Let $a, a' : A_{\mathcal{F}}$ such that $a' \sqsubseteq a$ and assume $a \triangleleft U$ and $a' \triangleleft V$. We need to show that $a' \triangleleft V \cap U$. We proceed by induction on the proof of $a' \triangleleft U$.

- Case dir. We have $a \in U$ and hence $a' \in U$ by the downwards-closure of U. The result then follows directly by Prop. 4.9.

- Case branch. We have some experiment $b : a$ such that $d(a, b, c) \triangleleft U$ for every outcome $c : C(a, b)$. By the branch rule, we are done if can exhibit some experiment $b' : B(a')$ such that $d(a', b', c') \triangleleft V \cap U$ for every $c' : C(a', b')$. We choose the b' given by the simulation property. Let $c' : C(a', b')$. Notice that $d(a', b', c') \sqsubseteq a'$ by monotonicity. This means, by the downwards-closure of $\_ \triangleleft V$ (Prop. 4.8), that $d(a', b', c') \triangleleft V$. It suffices, then, by the inductive hypothesis on $d(a, b, c)$, $d(a', b', c')$, and the proof of $d(a, b, c) \triangleleft U$ to show $d(a', b', c') \sqsubseteq d(a, b, c)$. This is precisely what the simulation property gives us.

- Case squash. We are done by the squash rule.

□

Proposition 4.11. *Let $\mathcal{F}$ be a formal topology and U, V be subsets of its underlying poset. If $U \subseteq \_ \triangleleft V$ then $\_ \triangleleft U \subseteq \_ \triangleleft V$.*

Proof sketch. By induction on the covering proof. The dir case is direct by assumption. The branch case follows by the branch rule with a direct appeal to the inductive hypothesis. The squash case follows by the squash rule. □

Let us now summarise our method for generating a frame out of a formal topology which comprises four steps.

1. Start with a formal topology $\mathcal{F}$.
2. $\mathcal{F}$ has an underlying poset P; construct its frame of downwards-closed subsets $P\downarrow$ (as given in Theorem 3.22).
3. Note: $\triangleright$ is a nucleus on $P\downarrow$.
4. We have shown (in Theorem 3.28) that the set of fixed points for every nucleus is a frame. The generated frame is $\mathsf{fix}\,(P\downarrow, \triangleright)$: the frame of elements of $P\downarrow$ that are fixed points for $\triangleright$.

The only missing step is (3): we have not yet shown that the covering relation is a nucleus. We will do exactly this in Section 4.3.

4.3 Covers are nuclei

$\_ \triangleright \_ : \mathcal{P}\,(A) \to \mathcal{P}\,(A)$ is an endofunction on the powerset of A. By restricting our attention to subsets that are downwards-closed, we can view this as an endofunction on the frame of downwards-closed subsets:

$$\_ \triangleright \_ : \mathsf{DCSubset}\,(A) \to \mathsf{DCSubset}\,(A)\,.$$

The natural question to be asked is then: is this a nucleus (Defn. 3.23) on the frame of downwards-closed subsets of A? Let us recapitulate what this means before we answer the question positively. Let U and V be downwards-closed subsets of a formal topology;

- N_0: $\prod_{(a\ :\ A)} a \triangleleft (U \cap V) = (a \triangleleft U) \cap (a \triangleleft V)$,
- N_1: $U \subseteq \_ \triangleleft U$, and
- N_2: $\_ \triangleleft (\_ \triangleleft U) \subseteq \_ \triangleleft U$.

Theorem 4.12. *Let $\mathcal{F}$ be a formal topology with an underlying poset P. The covering relation of $\mathcal{F}$ (defined in Defn. 4.7) is a nucleus on $P\downarrow$.*

Proof. N_1 is direct using the dir rule. N_2 is a direct application of Prop. 4.11.

For N_0, let $U, V : \mathsf{DCSubset}\,(|P|)$ be downwards-closed subsets. We will show

$$\_ \triangleleft (U \cap V) \quad = \quad (\_ \triangleleft U) \cap (\_ \triangleleft V)$$

by antisymmetry. The $(\_ \triangleleft U) \cap (\_ \triangleleft V) \subseteq \_ \triangleleft (U \cap V)$ direction follows directly by Proposition 4.10. For the other direction, let $a : A$ such that $a \triangleleft U \cap V$. We proceed by induction on this proof.

- Case dir. $a \in U \cap V$ meaning $a \in U$ and $a \in V$ hence we are done by two applications of dir.
- Case branch. Follows by two uses of the branch rule on two conjuncts of the inductive hypothesis.
- Case squash. We get two proofs of $a \triangleleft U \cap V$ and we use the inductive hypothesis on both. We combine the results' left conjuncts and right conjuncts by using the squash rule twice.

□

4.4 Lifting into the generated frame

Now that we have shown the nuclearity of the covering relation $\_ \triangleleft \_$, we have everything we need for the procedure of generating a frame from a formal topology.

Let $\mathcal{F}$ be a formal topology with underlying poset P. We know by Theorem 3.22 that the set of downwards-closed subsets of the underlying poset of $\mathcal{F}$ is a frame; denote this by $P\downarrow$. As we know that $\triangleright$ is a nucleus on this frame (by Theorem 4.12), we know that the set of fixed points for $\triangleright$ is a frame as well; we denote this $\mathsf{fix}\,(P\downarrow, \triangleright)$. Now, notice that we can define a lifting map $\eta : |P| \to \mathsf{fix}\,(P\downarrow, \triangleright)$ as follows:

$$\begin{aligned} \eta &: |P| \to \mathcal{P}\,(A_{\mathcal{F}}) \\ \eta(a) &:\equiv \_ \triangleleft a\downarrow \end{aligned}$$

where $a\downarrow$ denotes the *downwards-closure* of a: $\{a' \mid a' \sqsubseteq a\}$. So $a' \in \eta(a) \equiv a' \triangleleft a\downarrow$ is to say "a' eventually evolves into a stage that a evolves into" if we take the reading of stages of knowledge being refined over time. As we have proven in Prop. 4.8, $\eta(x)$ is downwards-closed. It is also a fixed point for $\triangleright$, as we will show, so its type can be refined to $\eta : |P| \to \mathsf{fix}\,(P\downarrow, \triangleright)$.

Definition 4.13 (η)**.** Let $\mathcal{F}$ be a formal topology with underlying poset P (note: $|P| \equiv A_{\mathcal{F}}$) and cover relation $\_ \lhd \_$. There exists a monotonic map from P to the underlying poset of $\mathsf{fix}\,(P\downarrow, \rhd)$, defined as:

$$\begin{aligned} \eta &: P \to_{\mathrm{m}} \mathsf{fix}\,(|P|\downarrow, \rhd) \\ \eta(a) &:\equiv \_ \lhd a\downarrow. \end{aligned}$$

The fact that η is monotonic follows from Proposition 4.8. It remains to be shown that $\eta(a)$ is a fixed point of $\rhd$ for every $a : A_{\mathcal{F}}$. Let $a : A_{\mathcal{F}}$. We need to show $\_ \lhd (\_ \lhd a\downarrow) = \_ \lhd a\downarrow$. We proceed by antisymmetry. $\_ \lhd a\downarrow \subseteq \_ \lhd (\_ \lhd a\downarrow)$ follows by N_1. The other direction is a direct application of Proposition 4.11.

4.5 Formal topologies present

We are now ready to shift our focus on what can be called main theorem of this thesis: formal topologies, as we have defined them, present frames as expected. Let $\mathcal{F}$ be a formal topology and F a frame. Consider a monotonic function $f : A_{\mathcal{F}} \to |\mathsf{fix}\,(A_{\mathcal{F}}\downarrow, \rhd)|$ on the underlying posets. We will define a notion of f representing $\mathcal{F}$ in F.

Definition 4.14 (Representation)**.** Given a formal topology $\mathcal{F} \equiv (A, B, C, d)$, a frame R, and a function $f : A \to |R|$, we say that f represents $\mathcal{F}$ in R if the following type is inhabited:

$$\mathsf{represents}\,(\mathcal{F}, R, f) \quad :\equiv \quad \prod_{(a\,:\,A)} \prod_{(b\,:\,B(a))} f(a) \sqsubseteq \bigvee_{c:C(a,b)}^{R} f(d(a,b,c)).$$

It can be shown that the opposite direction of the inequality expressed in the definition of $\mathsf{represents}$ always holds. To assert $\mathsf{represents}\,(\mathcal{F}, R, f)$ is then to assert:

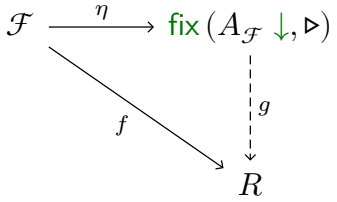
$$\prod_{(a\,:\,A)} \prod_{(b\,:\,B(a))} f(a) = \bigvee_{c:C(a,b)}^{R} f(d(a,b,c)).$$

This property therefore expresses that R behaves like a model of the covering relation of the formal topology $\mathcal{F}$, that is, that the join of R reflects the covering relation induced by the formal-topological structure of $\mathcal{F}$. This is the idea underlying the name "representation".

To state the universal property, we will work with *flat* monotonic maps. This is the special case of the notion of a *flat functor* [16] in the case where the categories we are working with are posets. Consider a monotonic map $f : P \to Q$, where Q has finite meets but P possibly does not. We would like to assert somehow that f preserves these finite meets but we cannot mention the meets of P as they do not exist. So we instead require f to preserve those meets that *do not exist yet* which we can do by requiring the following conditions:

1. $\top_Q = \bigvee \{f(a) \mid a : |P|\}$, and

Figure 4.1: The universal property.



2. $\prod_{(x\ y\ :\ P)} f(x) \wedge f(y) = \bigvee \{f(z) \mid z \sqsubseteq x \text{ and } z \sqsubseteq y\}$.

In our case, we will be considering those monotonic maps whose codomains are frames.

Definition 4.15 (Flat monotonic map)**.** Let P be a poset and F a frame. Denote by I the type $\sum_{(z\ :\ |F|)} z \sqsubseteq x \times z \sqsubseteq y$. A monotonic map $f : P \to F$ from P to the underlying poset of the frame is called *flat* if the following type is inhabited:

$$\begin{aligned} \mathsf{isFlat}\,(f) \quad :\equiv \quad & \top_F = \bigvee \{f(a) \mid a : |P|\} \\ \times \quad & \prod_{(a\ b\ :\ |P|)} f(a) \wedge f(b) = \bigvee_{(i,\_):I} f(i). \end{aligned}$$

Using flat monotonic maps, the universal property can now be stated.

Theorem 4.16 (Universal property for formal topologies)**.** *Given any formal topology $\mathcal{F}$, frame R, flat monotonic map $f : A_{\mathcal{F}} \to R$ from the underlying poset of $\mathcal{F}$ to the underlying poset of R, that represents $\mathcal{F}$ in R, there exists a unique* ***frame homomorphism*** *$g : \mathsf{fix}\,(A_{\mathcal{F}}\downarrow, \triangleright) \to_{\mathrm{f}} R$ such that $f = g \circ \eta$, as summarised in Fig. 4.1*

We express this fully formally as follows. Let $L :\equiv \mathsf{fix}\,(A_{\mathcal{F}}\downarrow, \triangleright)$;

$$\prod_{(\mathcal{F}\ :\ \mathsf{FT}_{n,n})} \prod_{(R\ :\ \mathsf{Frame}_{n+1,n,n})} \prod_{(f\ :\ P \to_{\mathrm{m}} R)} \mathsf{isFlat}\,(f) \to \mathsf{represents}\,(\mathcal{F}, F, f) \to \mathsf{isContr}\left(\sum_{(g\ :\ L \to_f R)} f = g \circ \eta\right).$$

First, observe how isContr (Defn. 2.1) expresses "there exists a unique" in the statement of this theorem. Given some type A and a family of propositions $B : A \to \Omega$ on A, to assert

$$\mathsf{isContr}\left(\sum_{(x\ :\ A)} B(x)\right),$$

is to assert precisely the existence of some $c : A$ satisfying B such that every other $x : A$ satisfying B is equal to c. This is the case due to the propositionality of B. Furthermore, notice also that this statement is talking only about formal topologies whose carrier levels and relation levels are the same. We have preferred this to keep the level bureaucracy manageable. For instance, we would have to generalise the

levels of the powerset operator (Defn. 2.34) to generalise these levels. We believe that such generalisations are not essential for our purposes in this thesis.

Before proceeding to the proof, let us first prove a lemma of key importance.

Lemma 4.17. *Let $\mathcal{F}$ be a formal topology (with its cover denoted by $\triangleleft$) and let $L :\equiv \mathsf{fix}\,(A_{\mathcal{F}} \downarrow, \triangleright)$. For any $U : L$, it is the case that:*

$$U = \bigvee^{L} \{\eta(u) \mid u \in U\}\,.$$

Proof. Let $U : L$. We proceed by antisymmetry. The $U \sqsubseteq \bigvee^{L} \{\eta(u) \mid u \in U\}$ direction is immediate by an application of the dir rule. For the other direction, let $a : A_{\mathcal{F}}$ and suppose that $a \in \bigvee^{L} \{\eta(u) \mid u \in U\}$. Recall that the join operation of the frame of fixed points for a nucleus (Theorem 3.28) on some frame F is defined by applying the nucleus on the join operator of F. This is to say that we have $a \triangleleft \bigcup \{\eta(u) \mid u \in U\}$. The result follows by induction on the proof of this fact, in the dir case of which Prop. 4.11 and the induction principle of truncation are used. □

We will also need the following lemma when showing the existence of a frame homomorphism making the diagram commute—specifically, when showing that it respects joins.

Lemma 4.18. *Let $\mathcal{F}$ be a formal topology, R, a frame and let*

$$L \quad :\equiv \quad \mathsf{fix}\,(A_{\mathcal{F}} \downarrow, \triangleright)\,.$$

Let $f : P \to_{\mathrm{m}} R$ be a flat *monotonic map that represents $\mathcal{F}$ in R (as defined in Defn. 4.14). Given any $a : A_{\mathcal{F}}$, and $U : \mathsf{Fam}\,(|L|)$ with index type I, if $a \in \bigvee^{L} U$ then*

$$\bigvee^{R} \left\{ f(a) \mid a \in \bigvee^{L} U \right\} = \bigvee^{R} \left\{ f(a) \mid (\_, (a, \_)) : \sum_{(i\,:\,I)} \sum_{(a\,:\,A_{\mathcal{F}})} a \in U_i \right\}$$

Proof. We proceed by antisymmetry. The $(\sqsupseteq)$ direction easily follows by the LUB property. For the $(\sqsubseteq)$ direction, it suffices by the LUB property to show:

$$\prod_{(a\,:\,A_{\mathcal{F}})} a \in \bigvee^{L} U \to f(a) \sqsubseteq \bigvee^{R} \left\{ f(a) \mid (\_, (a, \_)) : \sum_{(i\,:\,I)} \sum_{(a\,:\,A_{\mathcal{F}})} a \in U_i \right\}.$$

Let $a : A_{\mathcal{F}}$ such that $a \in \bigvee^{L} U$. We proceed by induction on the proof of $a \in \bigvee^{L} U$.

- Case dir. Direct using the recursion principle of truncation and the fact that $\bigvee^{R}$ is an upper bound.

- Case branch. We have some experiment $b : B(a)$ such that for every $c : C(a,b)$, $d(a,b,c) \triangleleft U$.

$$\begin{array}{lll} & f(a) & \\ \sqsubseteq & \bigvee^{R} \{f(d(a,b,c)) \mid c : C(a,b)\} & [f \text{ represents}] \\ \sqsubseteq & \bigvee^{R} \left\{ f(a) \mid (\_,(a,\_)) : \sum_{(i\,:\,I)} \sum_{(a\,:\,A_{\mathcal{F}})} a \in U_i \right\} & [\text{LUB and IH}]. \end{array}$$

- Case squash. Follows by the propositionality of $\sqsubseteq_R$.

□

Proof of Theorem 4.16. Let $\mathcal{F}$ be a formal topology and R a frame and let L denote $\mathsf{fix}\,(A_{\mathcal{F}} \downarrow, \triangleright)$. Let $f : P \to_{\mathrm{m}} R$ be a *flat* monotonic map that represents $\mathcal{F}$ in R (as defined in Defn. 4.14). We will show the *unique existence* of a *frame homomorphism* $g : L \to_{\mathrm{f}} R$ such that $f = g \circ \eta$.

We choose

$$\begin{array}{rcl} g & : & |L| \to |R| \\ g(U) & :\equiv & \bigvee^{R} \{f(u) \mid u \in U\}. \end{array}$$

To show that the diagram commutes, it suffices by Proposition 2.8 to show $f(a) = g(\eta(a))$ for every $a : A_{\mathcal{F}}$ (also by Proposition 2.20 to be precise). Let $a : A_{\mathcal{F}}$; we proceed by antisymmetry. The $f(a) \sqsubseteq g(\eta(a))$ direction is easy: $g(\eta(a)) \equiv \bigvee^{R} f(\eta(a))$ and $\bigvee^{R}$ is an upper bound of $f(\eta(a))$ so it suffices to show $f(a) \in f(\eta(a))$. This is immediate since $a \in \eta(a)$.

The other direction is the interesting one. First, we prove a lemma in preparation: *given any* $a, a' : A_{\mathcal{F}}$ *such that* $a' \triangleleft a\!\downarrow$, $f(a') \sqsubseteq f(a)$. Let $a, a' : A_{\mathcal{F}}$ and assume $a' \triangleleft a\!\downarrow$. We proceed by induction on the proof of $a' \triangleleft a\!\downarrow$.

- Case: dir. It must be that $a' \in a\!\downarrow$ which is to say $a' \sqsubseteq a$. We are then done by the monotonicity of f.
- Case: branch. There must be some $b' : B_{\mathcal{F}}(a')$ and a function h such that $h(c) : d(a', b', c') \triangleleft a\!\downarrow$ for any $c : C(a', b')$. Notice that

$$f(a') \sqsubseteq \bigvee_{c' : C(a',b')} f(d(a',b',c'))$$

by the assumption that f *represents* (Defn. 4.14) so we would be done if we could show

$$\bigvee_{c' : C(a',b')} f(d(a',b',c')) \sqsubseteq f(a).$$

As $\bigvee_{c' : C(a',b')} f(d(a',b',c'))$ is a LUB, it suffices to show that $f(a)$ is an upper bound of the subset of $A_{\mathcal{F}}$ delineated by $f(d(a',b',\_))$. Consider $f(d(a',b',c'))$ for some c'. We have that $h(c') : d(a',b',c') \triangleleft a\!\downarrow$ meaning $f(d(a',b',c')) \sqsubseteq f(a)$ by the inductive hypothesis.

- Case squash. We are done by the fact that the result type $f(a') \sqsubseteq f(a)$ is propositional.

Now, we want to show that $\bigvee^R f(\eta(a)) \sqsubseteq f(a)$. Since $\bigvee^R$ is a LUB it suffices to show that $f(a)$ is an upper bound of $\{f(a') \mid a' \in \eta(a)\}$. Let

$$f(a') \in \{f(a') \mid a' \in \eta(a)\}\,.$$

We need to show that $f(a') \sqsubseteq f(a)$. By the lemma we have just proven, it suffices to show that $a' \triangleleft a\downarrow$ and this holds directly because $f(a') \in \{f(a') \mid a' \in \eta(a)\}$.

There are two more things that have to be shown: (1) g is a frame homomorphism and (2) g is unique.

Let us first address (1). The fact that g is monotonic follows by the LUB property. The fact that g preserves the top element of L is given by the flatness assumption (Defn. 4.15): $g(\top) = \bigvee^R \{f\,(a) \mid a \in A_{\mathcal{F}}\} = \top_R$. To see that g also preserves binary meets, let $U, V : L$. First, we show:

$$\bigvee^R \{f(a) \mid a \in U \cap V\} = \bigvee^R \left\{ \bigvee^R \{f(w) \mid w \sqsubseteq u \times w \sqsubseteq v\} \mid (u,v) \in U \times V \right\}. \tag{4.19}$$

This follows by antisymmetry. For the ($\sqsubseteq$) direction, it suffices by the fact that $\bigvee^R$ is a LUB, to show that

$$\mathsf{RHS} \quad :\equiv \quad \bigvee^R \left\{ \bigvee^R \{f(w) \mid w \sqsubseteq u \times w \sqsubseteq v\} \mid (u,v) \in U \times V \right\}$$

is an upper bound of $\{f(a) \mid a \in U \cap V\}$. Let $a : A_{\mathcal{F}}$ such that $a \in U$ and $a \in V$. The fact that $f(a) \sqsubseteq \mathsf{RHS}$ can be seen to hold by the LUB property of $\bigvee^R$. For the ($\sqsupseteq$) direction, it is sufficient, since $\bigvee^R$ is less than any other upper bound, to show that

$$\bigvee^R \{f(a) \mid a \in U \cap V\}$$

is an upper bound of

$$\left\{ \bigvee^R \{f(w) \mid w \sqsubseteq u \times w \sqsubseteq v\} \mid (u,v) \in U \times V \right\}.$$

This follows straightforwardly by the LUB property combined with the downwards-closure of U and V. The result that g preserves binary meets then follows as:

$$\begin{aligned}
g(U \cap V) &\equiv \bigvee^R \{f(a) \mid a \in U \cap V\} && [(4.19)] \\
&= \bigvee^R \left\{ \bigvee^R \{f(w) \mid w \sqsubseteq u \times w \sqsubseteq v\} \mid (u,v) \in U \times V \right\} && [\text{flatness}] \\
&= \bigvee^R \{f(u) \wedge f(v) \mid (u,v) \in U \times V\} && [\text{Prop. } 3.15] \\
&= \left(\bigvee^R \{f(u) \mid u \in U\} \right) \wedge \left(\bigvee^R \{f(v) \mid v \in V\} \right) \\
&\equiv g(U) \wedge g(V).
\end{aligned}$$

Let us now show that g preserves joins. Let U be a family of inhabitants of $|L|$.

$$\begin{aligned}
g(\bigvee^{L} U) &\equiv \bigvee^{R} \left\{ f(a) \mid a \in \bigvee_{i}^{L} U_i \right\} && \text{[Lemma 4.18]} \\
&= \bigvee^{R} \left\{ f(a) \mid (_, (a, _)) : \sum_{(i \,:\, I)} \sum_{(a' \,:\, A_{\mathcal{F}})} a' \in U_i \right\} && \text{[Lemma 3.13]} \\
&= \bigvee^{R} \left\{ \bigvee^{R} \{ f\,(a) \mid a \in U_i \} \mid i : I \right\} \\
&\equiv \bigvee^{R} \{ g(U_i) \mid U_i \in U \}\,.
\end{aligned}$$

To be precise about the use of Lemma 3.13, we note that it used on the function:

$$\lambda\_\ (a, \_).\ f(a).$$

Finally, we conclude the proof by showing uniqueness of g: let g' be a frame homomorphism from L to R that makes the diagram commute. We need to show that $g = g'$. Let $U : |L|$. $g(U) = g'(U)$ by the following equational proof:

$$\begin{aligned}
g(U) &\equiv \bigvee^{R} \{ f(u) \mid u \in U \} && [f = g' \circ \eta] \\
&= \bigvee^{R} \{ g'(\eta(u)) \mid u \in U \} && [g' \text{ is a frame homomorphism}] \\
&= g' \left(\bigvee^{L} \{ \eta(u) \mid u \in U \} \right) && \text{[Lemma 4.17]} \\
&= g'(U).
\end{aligned}$$

Note that we are making implicit use of Prop. 2.20 as we are concluding $(g, p) = (g', q)$ where p and q are the respective homomorphism proofs. □

5

The Cantor space

Now that we have pinned down a notion of formal topology, let us make things a bit more concrete by looking at a prime example of a formal topology: the Cantor space. This chapter corresponds to the `CantorSpace` module in the Agda formalisation.

For our purposes, the Cantor space is the type of boolean sequences: $\mathbb{N} \to \mathsf{Bool}$. If we were following a pointful road to the topology of the Cantor space, we would start by saying the points of this space are sequences $f : \mathbb{N} \to \mathsf{Bool}$. As we are taking a *pointless* road instead, we will start by stating what the opens are: (finite) lists of booleans. Recalling the discussion about finite observations from Chapter 1, it is clear why this is the case: a function $f : \mathbb{N} \to \mathsf{Bool}$ is an infinite object, meaning we will never be able to fully write it down (i.e. its extension). If we are to somehow understand its behaviour, this has to be by *experimenting with it*, namely, examining its output up to some $n : \mathbb{N}$. In classical mathematics, the ability to examine such a function up to infinity is provided by the law of excluded middle. Although this may be useful for some purposes, it is clear that it does not fit well with the computational view of topology we would like to adopt: if we write a program to analyse a bitstream, and expect to get any sort of useful output from this program, it has to analyse the given bitstream up to some finite n. This is precisely the point of pointless topology: infinite things make computational sense only by finitary examinations, which is what the opens are. Accordingly, we now start writing down the underlying interaction system of the Cantor topology.

5.1 The Cantor interaction system

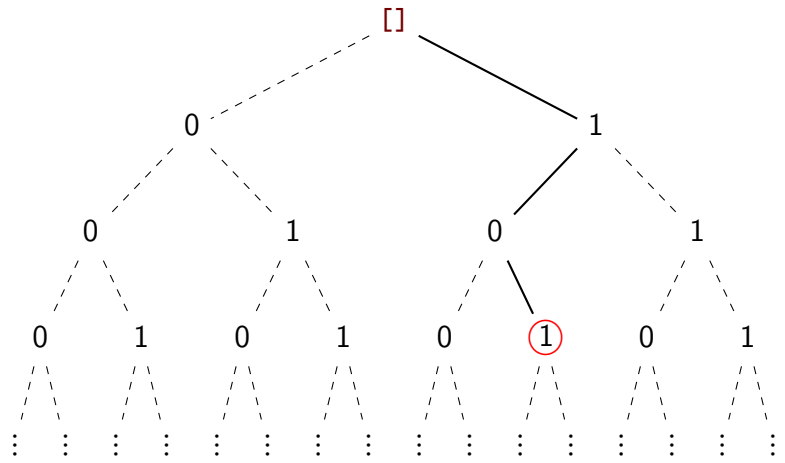
As we have explained in Section 4.1, we need to start defining the Cantor space by giving its interaction structure. A visualisation of this can be found in Fig. 5.1.

Stages: finite lists of booleans. A list of length n is the stage of information at which first n bits of the sequence have been found out.

Experiments: at any stage, there is only one experiment that can be performed: asking for the next bit. Each sequence can be thought of as a black-box computing device with a button on it; whenever the button is pressed, it emits a new bit.

Outcomes: the outcome of the experiment of asking for a new bit is simply the emitted bit itself

Figure 5.1: A visualisation of the Cantor space. Nodes correspond to stages of information (i.e. observed prefixes) about a sequence as there is a unique path from each node to the root. Dashed lines denote possible outcomes whereas plain lines denote actual ones. The encircled node is the stage of information at which the prefix $[] \frown 1 \frown 0 \frown 1$ has been observed.



Revision: when we receive some bit $b : \mathsf{Bool}$ at stage bs, we proceed to the next stage by appending b to bs.

To write this down formally, we first need to define the inductive type of (snoc) lists.

Definition 5.1 (Boolean snoc lists)**.** The type $\mathbb{B}$ of snoc lists of booleans is inductively defined by two constructors:

$$\frac{}{[] : \mathbb{B}} \qquad \frac{b : \mathsf{Bool} \qquad bs : \mathbb{B}}{bs \frown b : \mathbb{B}} \ .$$

We denote the singleton snoc list consisting of b, $[\ b\ ] :\equiv [] \frown b$.

Definition 5.2 (Prepend for snoc lists)**.** Given two inhabitants $bs_0, bs_1 : \mathbb{B}$, the operation of prepending of bs_0 to bs_1, denoted $+\!\!+$, is defined by recursion as follows:

$$\begin{aligned} \_ +\!\!+ \_ &: \mathbb{B} \to \mathbb{B} \to \mathbb{B} \\ bs_0 +\!\!+ [] &:\equiv bs_0 \\ bs_0 +\!\!+ (bs_1' \frown b_1) &:\equiv (bs_0 +\!\!+ bs_1') \frown b_1. \end{aligned}$$

The notation "$\mathbb{B}$" is mnemonic for *basic open* in that the inhabitants of $\mathbb{B}$ are basic opens of the Cantor space.

Proposition 5.3. $\mathbb{B}$ *is an h-set.*

Proof sketch. It is easy to show that $\mathbb{B}$ is discrete (Defn. 2.32) which is the case precisely because the type of booleans is discrete. It is therefore a set by Hedberg's theorem (Theorem 2.33). □

Definition 5.4 (The Cantor poset)**.** The set $\mathbb{B}$ forms a poset under the following order. Let $bs_0, bs_1 : \mathbb{B}$;

$$bs_1 \leq bs_0 \quad :\equiv \quad \sum_{(bs_2 \,:\, \mathbb{B})} bs_1 = bs_0 +\!\!+ bs_2.$$

We will refer to such a bs_2, that is the part of bs_1 after bs_0, as the *difference* of bs_1 from bs_0. The fact that $\mathbb{B}$ is a set is given in Proposition 5.3. Reflexivity is immediate by picking $bs_2 :\equiv$ `[]`. Transitivity boils down to to the associativity of $+\!\!+$. For antisymmetry, let $bs, bs' : \mathbb{B}$ such that $bs \leq bs'$ and $bs' \leq bs$. The result is immediate in the cases where either difference is `[]`, and the case where it is not is impossible. This latter fact involves a non-trivial amount of bureaucracy to prove in a completely precise way but is intuitively apparent.

Finally, note that the relation $\leq$ is h-propositional (without the need for truncation!). This follows from the injectivity of $bs_0 +\!\!+ \_$.

Now let us write down the interaction system that underlies the Cantor space.

Definition 5.5 (The Cantor interaction system)**.** We will call the Cantor interaction system $\mathcal{C}$. It is defined as $\mathcal{C} :\equiv (A, B, C, d)$, where

$$\begin{aligned} A &:\equiv \mathbb{B} \\ B(bs) &:\equiv \mathsf{Unit} \\ C(bs, \star) &:\equiv \mathsf{Bool} \\ d(bs, \star, b) &:\equiv bs \frown b. \end{aligned}$$

Definition 5.5 provides the *data* of the topology. Let us now prove this actually forms a formal topology.

Theorem 5.6 (The Cantor formal topology)**.** *The Cantor interaction system on the Cantor poset forms a formal topology, that is, it satisfies the monotonicity (Defn. 4.2) and the simulation (Defn. 4.3) properties.*

Proof. Monotonicity is immediate since, given a boolean b, the operation $\_ \frown b$ of snocing it onto a list of booleans clearly gets us in a more refined stage, as witnessed by the difference of `[ b ]`.

For the simulation property (Defn. 4.3), let $bs_0, bs_1 : \mathbb{B}$ and assume that $bs_1 \leq bs_0$. Note that the experiment type is trivial so it suffices to show that for any outcome of $b_1 : C(bs_1, \star)$, there exists some $b_0 : C(bs_0, \star)$ such that

$$d(bs_1, \star, b_1) \sqsubseteq d(bs_0, \star, b_0).$$

Unfolding the definitional equalities in this statement, we can restate our goal as follows: for any boolean b_1 there exists some boolean b_0 such that

$$bs_1 \frown b_1 \leq bs_0 \frown b_0.$$

Let b_1 be an arbitrary boolean. We shall now construct such a b_0. We proceed by case analysis on the difference of bs_1 from bs_0.

- Case: `[]`. This means that $bs_0 = bs_1$. We choose $b_0 :\equiv b_1$. It is clear that $bs_1 \frown b_1 \leq bs_0 \frown b_1$ by reflexivity.
- Case: $bs \frown b$. In this case, $bs_1 = bs_0 +\!\!+ (bs \frown b)$ and we know that $bs \frown b$ is nonempty. Call the first element (counting from left to right) of $bs \frown b$, b_h, and the rest of it, bs_t (h for "head" and t for "tail"). We choose $b_0 :\equiv b_h$. It remains to be shown that $bs_1 \frown b_1 \leq bs_0 \frown b_h$. Observe the following equality:

$$\begin{aligned} bs_1 \frown b_1 &= (bs_0 +\!\!+ (bs \frown b)) \frown b_1 \\ &= (bs_0 +\!\!+ (\texttt{[}\ b_h\ \texttt{]} +\!\!+ bs_t)) \frown b_1 \\ &= ((bs_0 +\!\!+ \texttt{[}\ b_h\ \texttt{]}) +\!\!+ bs_t) \frown b_1 \\ &\equiv (bs_0 +\!\!+ \texttt{[}\ b_h\ \texttt{]}) +\!\!+ (bs_t \frown b_1). \end{aligned}$$

  This means that $bs_1 \frown b_1 \leq bs_0 \frown b_h$ is witnessed by the difference $bs_t \frown b_1$.

□

5.2 The Cantor space is compact

An important property of topological spaces in general topology is that of *compactness*. A space X is called compact if

every open cover of X has a finite subcover.

We remarked that an open cover of space X is a collection $\{V_i \mid i \in I\}$ of open sets of X such that $X = \bigcup_i V_i$. As we are viewing open sets as observable properties, this means that an open cover $\{V_i \mid i \in I\}$ is a way of decomposing X into $|I|$-many observable properties. If X is compact, every time we can decompose X into such a collection of open subsets, we can find a **finite subset** of this collection that remains a decomposition of X into observable properties. In other words, even though X might not be finite, its behaviour can somehow be *reduced to finitely many observable properties*.

As we have been representing topologies directly in terms of the notion of an open cover, it will not be too hard for us to state this property for a formal topology. As we will now proceed to do this, let us first clarify some notation. We will assume the usual list type and denote the type of lists containing inhabitants of some type A, $\mathsf{List}\,(A)$. The cons operator and the empty list are denoted $\_ :: \_$ and `[]` respectively. Notice that we are denoting both empty lists of type $\mathbb{B}$ and $\mathsf{List}\,(A)$ by `[]`, as its type should be obvious in context. Similarly, we will denote the singleton list and the append operation of both types using the same notations: `[ _ ]`, $+\!\!+$, respectively. Again, the type should be clear from the context. However, in the coloured version of this thesis, the $+\!\!+$ operation on $\mathbb{B}$ will be coloured *dark green* as it is a definiendum we have introduced by means of a definitional equality (Defn. 5.2).

One might wonder at this point why we are using two kinds of lists. We could certainly modify $\mathbb{B}$ to accommodate polymorphic lists, and use this instead of List. However, we believe that the presentation is clearer when snoc lists are reserved for the type $\mathbb{B}$ to foreground its downwards-growing nature.

Let us now define an auxiliary function that we will need when stating compactness.

Definition 5.7. Given a poset $P : \mathsf{Poset}_{n,n}$ and $xs : \mathsf{List}\,(|P|)$, we define the following function expressing the subset of inhabitants of $|P|$ that are below at least one element of xs.

$$\begin{aligned} \mathsf{down} &: \mathsf{List}\,(A) \to \mathcal{P}\,(A) \\ \mathsf{down}\,([]) &:\equiv \lambda\_.\ \bot_n \\ \mathsf{down}\,(x :: xs') &:\equiv \lambda y.\ \|y \sqsubseteq_P x + y \in \mathsf{down}\,(xs')\| \,. \end{aligned}$$

We can now state[1] the compactness of a formal topology.

Definition 5.8 (Compactness)**.** A given formal topology $\mathcal{F}$ is compact if the following type is inhabited:

$$\begin{aligned} &\mathsf{isCompact}\,(\mathcal{F}) \quad :\equiv \\ &\prod_{(a\ :\ A)} \prod_{((U,\_)\ :\ \mathsf{DCSubset}(A))} \\ &\qquad a \triangleleft U \to \left\| \sum_{(as\ :\ \mathsf{List}(A))} a \triangleleft \mathsf{down}\,(as) \times \mathsf{down}\,(as) \in U \right\| . \end{aligned}$$

Notice that the result type must be truncated due to the existence of the squash constructor: otherwise it would be hard to define proofs inhabiting this type due to the induction principle of covering. One could argue that it would have to be truncated *anyway*, since compactness is a property and should therefore be propositional. However, one could consider refining this property so that it uniquely characterises the finite subcover. This would render the property naturally propositional, hence freeing us from the obligation to truncate it.

We will now prove that the Cantor space is compact. For this, we first prove three small lemmas for the sake of clarity.

Lemma 5.9. *Given downwards-closed subsets U and V of the Cantor poset, if $U \subseteq V$ then $\_ \triangleleft U \subseteq \_ \triangleleft V$.*

Proof. Corollary of Proposition 4.11. □

Lemma 5.10. *For any two lists $bss_0, bss_1 : \mathsf{List}\,(\mathbb{B})$,*

$$\mathsf{down}\,(bss_0) \subseteq \mathsf{down}\,(bss_0 +\!\!+ bss_1) \quad \textit{and} \quad \mathsf{down}\,(bss_1) \subseteq \mathsf{down}\,(bss_0 +\!\!+ bss_1)\,.$$

Proof sketch. Straightforward induction. Notice, however that the recursion principle of propositional truncation has to be invoked. □

Lemma 5.11. *Given any $bs : \mathbb{B}$ and $bss_0, bss_1 : \mathsf{List}\,(\mathbb{B})$, if $bs \in \mathsf{down}\,(bss_0 +\!\!+ bss_1)$ then either $bs \in \mathsf{down}\,(bss_0)$ or $bs \in \mathsf{down}\,(bss_1)$.*

[1] This specific form of the compactness statement for a formal topology was communicated (privately) to the author by Thierry Coquand, the supervisor.

Proof sketch. Result follows by induction on bss_0. □

It should be possible to generalise these lemmas to arbitrary formal topologies but we have no motivation to do so in this context.

We are now ready to prove the compactness of the Cantor space. Note that we denote the two inhabitants of Bool by 1 and 0.

Theorem 5.12. *$\mathcal{C}$ is compact.*

Proof. Let $bs : \mathbb{B}$, and U, a downwards-closed subset of the Cantor poset such that $bs \triangleleft U$. We need to show that U has a finite subcover i.e. there exists some $bss : \mathsf{List}(\mathbb{B})$ such that $bs \triangleleft \mathsf{down}(bss)$ and $\mathsf{down}(bss) \subseteq U$. We proceed by induction on the proof of $bs \triangleleft U$.

Case: dir. Choose $bss :\equiv [\ bs\ ]$, namely, the singleton list consisting of bs. The first property follows by the dir rule and reflexivity whilst the second one holds by the downwards-closure of U.

Case: branch. We know that $bs \frown b \triangleleft U$ for *any* b, by the right premise of the branch rule. Now, notice that we can appeal to the inductive hypothesis with both of $bs \frown 0$ and $bs \frown 1$, as we know $bs \frown 1 \triangleleft U$ and $bs \frown 0 \triangleleft U$. This gives us some bss_0 and bss_1 such that not only

$$(\dagger) \qquad bs \frown 0 \triangleleft \mathsf{down}(bss_0) \quad \text{and} \quad bs \frown 1 \triangleleft \mathsf{down}(bss_1)\,,$$

but also

$$(\ddagger) \qquad \mathsf{down}(bss_0) \subseteq U \quad \text{and} \quad \mathsf{down}(bss_1) \subseteq U.$$

We now pick $bss :\equiv bss_0 +\!\!+ bss_1$. It remains to be shown that

$$bs \triangleleft \mathsf{down}(bss_0 +\!\!+ bss_1) \quad \text{and} \quad \mathsf{down}(bss_0 +\!\!+ bss_1) \subseteq U.$$

The right conjunct is easy to verify. Let $bss' \in \mathsf{down}(bss_0 +\!\!+ bss_1)$. By Lemma 5.11, we have two cases: either $bss' \in \mathsf{down}(bss_0)$ or $bss' \in \mathsf{down}(bss_1)$. In either case, we have that $bss' \in U$ by $(\ddagger)$.

For the left conjunct, notice that it suffices, by the branch rule, to show

$$bs \frown b \triangleleft \mathsf{down}(bss_0 +\!\!+ bss_1)$$

for any $b : \mathsf{Bool}$. Let b be an arbitrary boolean. There are two cases for b.

- Case $b \equiv 0$. By Lemma 5.9, and the fact that

  $$\mathsf{down}(bss_0) \subseteq \mathsf{down}(bss_0 +\!\!+ bss_1) \qquad \text{(by Lemma 5.10)},$$

  it suffices to show $bs \frown 0 \triangleleft \mathsf{down}(bss_0)$ which we know by $(\dagger)$.
- Case $b \equiv 1$. Exactly the same reasoning as in the $b \equiv 0$ case but with bss_1 instead bss_o.

Case squash. We are done by the trunc constructor. □

6

Conclusion

The aim of formal topology is to make type-theoretical sense out of topology. The recent formulation of *univalent* type theories present interesting novelties for the future of type-theoretical mathematics. The meaning of these novelties for existing bodies of mathematical knowledge is an active area of research and it is fair to say that they have not yet been fully explored. Considering this, we have set out to investigate a natural question: what will formal topology look like in the context of univalent type theory?

The first outcome of this investigation was a negative result: we have come to the conclusion that an *as is* development of formal topology in univalent type theory is problematic as a consequence of the distinction between *property* and *structure*, rendered visible through the homotopical view of types. In particular, our development led to a situation in which the proof structure of the covering relation had to be collapsed for it to behave as a property, and at the same time, kept intact for it to be used in succeeding proofs.

We have then found, somewhat unexpectedly, that this problem can be circumvented by using HITs. Using HITs, we were able to reconcile, in our specific case, the imbalance between property and structure. This allowed us to successfully carry out a rudimentary development of formal topology in a univalent setting: we have been able to formulate a notion of formal topology and then prove its universal property. We hence conclude that formal topology is not only possible in univalent type theory but is possible in a way that makes non-trivial use of its conceptual novelties.

HITs have been previously used for similar purposes, and have allowed the circumvention of choice principles. The relevant work known to the author includes the higher inductive-inductive definition of the Cauchy reals in the HoTT book [42, Defn. 11.3.2] and the reconstruction of the partiality monad by Altenkirch, Danielsson, and Kraus [3]. The latter seems to be especially relevant to our use of HITs, as the view of topology as a theory of observable properties is closely related to partial computations (see, for instance, [15, 37]).

In addition to providing a preliminary answer to the question of doing formal topology in univalent type theory, we have carried out Coquand's idea of doing topology with interaction systems on posets [9] to a further extent. We have found this to work well in the context of univalent type theory. Apart from addressing issues of univalence, this has been the secondary contribution of our development. To the author's knowledge, this thesis presents the first formalised development of formal topologies as interaction systems.

As a proof of concept of our approach to formal topology, we have implemented

a fundamental example: the Cantor space. Furthermore, we have proven that the formal Cantor topology is compact. This exemplifies what we mean by "making type-theoretical sense of topology": we have stated and then proven a non-trivial theorem of topology in a way that is completely constructive and predicative.

After all, our development remains but an *esquisse*. There remains much more work to be done and it will be the undertaking of this that will put this approach to the test. We now discuss some of these.

First and foremost, there are many more topological and locale-theoretic results to be reconstructed in the setting of our development. For instance, one of the fundamental theorems of topology is the Tychonoff theorem, formal versions of which have been constructed [8, 45]. It is an interesting question if these will fit well within our approach. Furthermore, a crucial class of spaces in pointless topology is the *Stone spaces* [25]; indeed, the notion of a Stone space is what brought the field into existence in the first place. Developing the theory of Stone spaces in the context of our development is an important direction for further work.

Another limitation of our development is that we have not been able to explore the notion of a *formal point*, which can be introduced as a notion derived from the opens in formal topology [11, pg. 94]. A prime example of this is the space of real numbers, defined as the formal points of the formal topology of the real numbers. It would be interesting to explore such a definition of real numbers in univalent type theory.

Furthermore, formal topology has important connections to the theory of domains. As pointed out by Sambin, domain theory can be viewed as a "branch of formal topology" [34]. It was within our plans to provide a formal-topological reconstruction of domain theory in univalent type theory, as an application of this development. We have not been able to accomplish this due to time constraints. This would be a very interesting example as it would likely make non-trivial use of the constructive properties of our development. Domain theory in a univalent setting has recently been investigated by de Jong [12]. The question of how this compares to our development should be investigated.

Let us also discuss some of the lower-level shortcomings of our development that could be more readily addressed.

As remarked in Chapter 4, our method for generating a frame from a formal topology works only on formal topologies whose carrier set level and order level are equal—this is a restriction we have imposed for the sake of presentational simplicity. To reach full generality, these levels have to be generalised.

Furthermore, our statement of the universal property has not been modular. We have proven that a frame generated from a formal topology using our method is a presented frame. The statement of this theorem, however, does not express what it means for an arbitrary formal topology to present an arbitrary frame. Instead, we have stated this in an ad hoc way, for the specific case of our generated frame. Ideally, this should be stated for *any* frame with a lifting map, our current statement then being one instance of it. Most importantly, this would allow us to attempt to prove (which we *believe* would be a successful attempt) that two frames satisfying the universal property are isomorphic. As we proved that isomorphic frames are equal, this would mean two such frames are *equal*.

A further direction to take for the modularisation of the universal property would be to state it using categorical terms. Conceptually, the universal property ought to amount to the existence of an adjoint pair of a free and a forgetful functor. However, it is not clear what the category of formal topologies is, as the notion of a morphism between two interaction systems is not clear. Therefore, the resolution of this issue is likely not straightforward.

We believe that the type-theoretical distillation of topology enables a conceptual clarification if it. Thanks to the invention of univalent type theory, type-theoretical mathematics has been invigorated, and is now rapidly progressing. We hope the discipline of formal topology can benefit from this progress, ideally through a renewed interest by the type theory community.

Bibliography

[1] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1):1–77, 1991-03-14. ISSN: 0168-0072. DOI: 10.1016/0168-0072(91)90065-T.

[2] T. Altenkirch. From setoid hell to homotopy heaven? 2017-06-01. URL: http://www.cs.nott.ac.uk/~psztxa/talks/types-17-hell.pdf (visited on 2020-05-05).

[3] T. Altenkirch, N. A. Danielsson, and N. Kraus. Partiality, Revisited. In J. Esparza and A. S. Murawski, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 534–549, Berlin, Heidelberg. Springer, 2017. ISBN: 978-3-662-54458-7. DOI: 10.1007/978-3-662-54458-7_31.

[4] T. Altenkirch, N. Ghani, P. Hancock, C. Mcbride, and P. Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S095679681500009X.

[5] M. A. Armstrong. *Basic Topology*. Springer, New York; London, 2011. ISBN: 978-1-4419-2819-1.

[6] S. Awodey. Type theory of higher-dimensional categories. 2006-11-13. URL: http://www2.math.uu.se/~palmgren/itt/ (visited on 2020-05-05).

[7] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2016-11-07. arXiv: 1611.02108 [cs, math]. URL: http://arxiv.org/abs/1611.02108 (visited on 2019-12-18).

[8] T. Coquand. An intuitionistic proof of Tychonoff's theorem. *The Journal of Symbolic Logic*, 57(1):28–32, 1992-03. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2275174.

[9] T. Coquand. Formal Topology with Posets, 1996-10.

[10] T. Coquand and N. A. Danielsson. Isomorphism is equality. *Indagationes Mathematicae*. In Memory of N.G. (Dick) de Bruijn (1918–2012), 24(4):1105–1120, 2013-11-15. ISSN: 0019-3577. DOI: 10.1016/j.indag.2013.09.002.

[11] T. Coquand, G. Sambin, J. Smith, and S. Valentini. Inductively generated formal topologies. *Annals of Pure and Applied Logic*, 124(1-3):71–106, 2003. ISSN: 0168-0072. DOI: 10.1016/S0168-0072(03)00052-6.

[12] T. de Jong. The Scott model of PCF in Univalent Type Theory. In TYPES 2019, Oslo, Norway, 2019-06. arXiv: 1904.09810.

[13] A. G. Dragalin. Explicit algebraic models for constructive and classical theories with non-standard elements. *Studia Logica*, 55(1):33–61, 1995-02-01. ISSN: 1572-8730. DOI: 10.1007/BF01053031.

[14] M. Escardó. Introduction to Homotopy Type Theory and Univalent Foundations (HoTT/UF) with Agda. 2019-03-04. URL: https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes (visited on 2019-12-03).

[15] M. Escardó. Synthetic Topology: of Data Types and Classical Spaces. *Electronic Notes in Theoretical Computer Science*. Proceedings of the Workshop on Domain Theoretic Methods for Probabilistic Processes, 87:21–156, 2004-11-12. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.09.017.

[16] Flat functor in nLab. URL: https://ncatlab.org/nlab/show/flat+functor (visited on 2020-04-21).

[17] Formal topology in nLab. URL: https://ncatlab.org/nlab/show/formal+topology (visited on 2020-05-09).

[18] R. Garner. Factorisation systems for type theory. 2006-11-13. URL: http://www2.math.uu.se/~palmgren/itt/ (visited on 2020-05-05).

[19] A. Grothendieck and J.-L. Verdier. *Theorie Des Topos et Cohomologie Etale Des Schemas. Seminaire de Geometrie Algebrique Du Bois-Marie 1963-1964 (SGA 4): Tome 1*. Lecture Notes in Mathematics. Springer-Verlag, Berlin Heidelberg, 1972. ISBN: 978-3-540-37549-4. DOI: 10.1007/BFb0081551.

[20] A. Grzegorczyk. A philosophically plausible formal interpretation of intuitionistic logic. *Indagationes Mathematicae*, 26:596–601, 1964. DOI: 10.2307/2270284.

[21] P. Hancock. Interaction Systems. 2001-07-30. URL: http://www.dcs.ed.ac.uk/home/pgh/interactive_systems.html (visited on 2019-11-26).

[22] R. Harper. *Practical Foundations for Programming Languages*. In collaboration with Cambridge University Press. Cambridge University Press, New York NY, second edition. Edition, 2016. ISBN: 978-1-316-57689-2.

[23] M. Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998-07. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796898003153.

[24] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory (Venice, 1995)*. Volume 36, Oxford Logic Guides, pages 83–111. Oxford Univ. Press, New York, 1998.

[25] P. T. Johnstone. *Stone Spaces*. Cambridge Univ. Press, Cambridge, 2002. ISBN: 978-0-521-33779-3.

[26] P. T. Johnstone. The point of pointless topology. *Bulletin of the American Mathematical Society*, 8(1):41–53, 1983. ISSN: 0273-0979, 1088-9485. DOI: 10.1090/S0273-0979-1983-15080-2.

[27] F. W. Lawvere. Quantifiers and Sheaves. In Actes Du Congrès International Des Mathématiciens, volume 1, pages 329–334, Nice, France, 1970.

[28] A. Mörtberg, A. Vezzosi, and GitHub contributors. Cubical: an experimental library for Cubical Agda. URL: https://github.com/agda/cubical (visited on 2020-04-28).

[29] J. R. Munkres. *Topology*. Prentice Hall, Inc., 2000. ISBN: 0-13-181629-2.

[30] Nucleus in nLab. URL: https://ncatlab.org/nlab/show/nucleus (visited on 2020-04-26).

[31] E. Palmgren. From Intuitionistic to Point-Free Topology: On the Foundation of Homotopy Theory. In S. Lindström, E. Palmgren, K. Segerberg, and V. Stoltenberg-Hansen, editors, *Logicism, Intuitionism, and Formalism: What Has Become of Them?*, Synthese Library, pages 237–253. Springer Netherlands, Dordrecht, 2009. ISBN: 978-1-4020-8926-8. DOI: 10.1007/978-1-4020-8926-8_12.

[32] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, Lecture Notes in Computer Science, pages 128–140, Berlin, Heidelberg. Springer, 1989. ISBN: 978-3-540-46740-3. DOI: 10.1007/BFb0018349.

[33] G. Sambin. *A Tutorial on Formal Topology and the Basic Picture*. 2007.

[34] G. Sambin. Formal Topology and Domains. *Electronic Notes in Theoretical Computer Science*. Workshop on Domains IV, 35:177–190, 2000-01-01. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80742-X.

[35] G. Sambin. Intuitionistic Formal Spaces — A First Communication. In D. G. Skordev, editor, *Mathematical Logic and Its Applications*, pages 187–204. Springer US, Boston, MA, 1987. ISBN: 978-1-4613-0897-3. DOI: 10.1007/978-1-4613-0897-3_12.

[36] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411–440, 1993-12-06. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90095-B.

[37] M. Shulman. Homotopy type theory: the logic of space, 2017-03-08. arXiv: 1703.03007 [math]. URL: http://arxiv.org/abs/1703.03007 (visited on 2020-05-30).

[38] M. B. Smyth. Topology. In *Handbook of Logic in Computer Science (Vol. 1)*, pages 641–761. Oxford University Press, Inc., USA, 1993. ISBN: 0-19-853735-2.

[39] M. H. Stone. The Theory of Representation for Boolean Algebras. *Transactions of the American Mathematical Society*, 40(1):37–111, 1936. ISSN: 0002-9947. DOI: 10.2307/1989664.

[40] P. Taylor. Geometric and higher order logic in terms of abstract Stone duality. *Theory and Applications of Categories*, 7(15):284–338, 2000.

[41] P. Taylor. *Practical Foundations of Mathematics*. Cambridge University Press, Cambridge, 2003. ISBN: 978-0-521-07043-0.

[42] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: https://homotopytypetheory.org/book.

[43] B. van den Berg. Types as weak omega-categories. 2016-11-14. URL: http://www2.math.uu.se/~palmgren/itt/ (visited on 2020-05-05).

[44] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.*, 3:87:1–87:29, ICFP, 2019-07. ISSN: 2475-1421. DOI: 10.1145/3341691.

[45] S. Vickers. Some constructive roads to tychonoff. In L. Crosilla and P. Schuster, editors. Redacted by D. M. Gabbay, A. J. MacIntyre, and D. S. Scott, *From Sets and Types to Topology and Analysis: Towards Practicable Foundations for Constructive Mathematics*, number 48 in Oxford Logic Guides, pages 223–237. Oxford Science Publications, 2005-12. ISBN: 978-0-19-856651-9.

[46] S. Vickers. *Topology via Logic*. Cambridge University Press, Cambridge, 1989.

[47] V. Voevodsky. A very short note on homotopy lambda calculus. *Unpublished note*, 2006-10-27.

[48] V. Voevodsky. Foundations of mathematics - their past, present and future. 2014-09-09/2014-09-10.

[49] M. A. Warren. Homotopy models of intensional type theory. 2006-06.

A

Agda formalisation

A.1 Discussion of some syntax declarations

We first discuss some of the ways in which the notation of the Agda formalisation differs than the notation we use in the thesis. This is not intended to be a comprehensive list of how each name maps to the Agda formalisation. We discuss only salient differences that we think might be confusing to the reader.

In Defn. 2.39, we mentioned that we denote application of join operators by $\bigvee_i U_i$. We use a very similar syntactic sugaring in the Agda formalisation. The syntax declaration for this can be found in the `JoinSyntax` submodule of the `Frame` module.

```
module JoinSyntax (A : Type ℓ₀) {ℓ₂ : Level} (join : Fam ℓ₂ A → A) where

  join-of : {I : Type ℓ₂} → (I → A) → A
  join-of {I = I} f = join (I , f)

  syntax join-of (λ i → e) = ⋁⟨ i ⟩ e
```

The bracketed index $\langle i \rangle$ here is intended to be a plain text approximation of subscripting.

Similarly, we use syntactic sugaring for the family comprehension notation ($\{\cdots\}$) in the Agda formalisation as well. As the curly bracket symbols, `{` and `}`, are not available in Agda, we approximate these as `[` and `]`. One instance of such a syntax declaration can be found in the `Family` module.

```
  img : {X : Type ℓ₀} {Y : Type ℓ₁} → (g : X → Y) → (U : Fam ℓ₂ X) → Fam ℓ₂ Y
  img g (I , f) = I , g ∘ f

  syntax img (λ x → e) U = [ e | x ε U ]
```

A.2 The `Basis` module

```
{-# OPTIONS --cubical --safe #-}

module Basis where
```

```
open import Level public


import Cubical.Core.Everything               as CE
import Cubical.Data.Sigma                    as DΣ
import Cubical.Data.Sum                      as DS
import Cubical.Foundations.Prelude           as FP
import Cubical.Foundations.Equiv             as FE
import Cubical.Foundations.Logic             as FL
import Cubical.Foundations.HLevels           as FH
import Cubical.Foundations.Isomorphism       as FI
import Cubical.Foundations.Equiv.HalfAdjoint as HAE
import Cubical.Functions.FunExtEquiv         as FEE
import Cubical.Foundations.Function          as FF

open import Cubical.Foundations.Univalence public using (ua)

open CE public using    (_≡_; Type; Σ; Σ-syntax; _,_; _≃_; equivFun; isEquiv)
open DΣ public using    (ΣProp≡; sigmaPath→pathSigma; pathSigma→sigmaPath; _×_; _,_)
              renaming (fst to π₀; snd to π₁)
open DS public using    (inl; inr; _⊎_)
open FP public using    (funExt; subst; isContr; isProp; isPropIsProp; isSet;
                         isProp→isSet; cong; refl; sym; _≡⟨_⟩_; _∎; transport;
                         transportRefl; J; JRefl)
open FE public using    (idEquiv; invEquiv; secEq; retEq; fiber; equivToIso;
                         isPropIsEquiv)
open FL public using    ( _⇔_ ; _→_ ; ⇔toPath ; _⊓_ ; [_] )
open FH public using    (hProp; isSetHProp; isPropIsSet; isPropΣ; isOfHLevel;
                         isOfHLevelΠ; isOfHLevelΣ; isOfHLevelSuc; isSetΣ;
                         isSetΠ; isSetΠ2; isPropΠ; isPropΠ2; isPropΠ3)
open FI public using    (isoToPath; isoToEquiv; iso; section; retract; Iso)
open FF public using    (_∘_) renaming (idfun to id)
open FEE public using   (funExtEquiv; funExt₂; funExt₂Equiv; funExt₂Path)
open HAE public using   (isHAEquiv; equiv→HAEquiv)

variable
  ℓ ℓ′ ℓ₀ ℓ₁ ℓ₂ ℓ₃ ℓ₀′ ℓ₁′ ℓ₂′ ℓ₀′′ ℓ₁′′ ℓ₂′′ : Level

variable
  A  : Type ℓ₀
  B  : A → Type ℓ₀
  A₀ : Type ℓ₁

_↔_ : (A : Type ℓ) (B : Type ℓ′) → Type (ℓ ⊔ ℓ′)
A ↔ B = (A → B) × (B → A)

↔-to : {A : Type ℓ} {B : Type ℓ′} → A ↔ B → A → B
↔-to (to , _) = to

↔-from : {A : Type ℓ} {B : Type ℓ′} → A ↔ B → B → A
↔-from (_ , from) = from
```

The unit type

```
data Unit (ℓ : Level) : Type ℓ where
  tt : Unit ℓ

Unit-prop : {ℓ : Level} → isProp (Unit ℓ)
Unit-prop tt tt = refl
```

Bottom

```
data 𝟘 (ℓ : Level) : Type ℓ where

bot : (ℓ : Level) → hProp ℓ
bot ℓ = 𝟘 ℓ , λ ()
```

Propositions

```
is-true-prop : (P : hProp ℓ) → isProp [ P ]
is-true-prop (P , P-prop) = P-prop

∃_ : {A : Type ℓ₀} → (A → hProp ℓ₁) → Type (ℓ₀ ⊔ ℓ₁)
∃_ {A = A} P = Σ[ x ∈ A ] [ P x ]
```

Extensional equality

```
_~_ : (f g : (x : A) → B x) → Type _
_~_ {A = A} f g = (x : A) → f x ≡ g x
```

Powerset

```
𝒫 : Type ℓ → Type (suc ℓ)
𝒫 {ℓ} A = A → hProp ℓ

_∈_ : A → 𝒫 A → hProp _
x ∈ U = U x

∈-prop : {A : Type ℓ} {x : A} → (U : 𝒫 A) → isProp [ x ∈ U ]
```

```
∈-prop {x = x} U = is-true-prop (x ∈ U)

𝒫-set : (A : Type ℓ) → isSet (𝒫 A)
𝒫-set A = isSetΠ λ _ → isSetHProp

variable
  U V : 𝒫 A

_⊆⊆_ : {A : Type ℓ} → (A → Type ℓ₀) → (A → Type ℓ₁) → Type (ℓ ⊔ ℓ₀ ⊔ ℓ₁)
_⊆⊆_ {A = A} U V = (x : A) → U x → V x

_⊆_ : {A : Type ℓ} → 𝒫 A → 𝒫 A → hProp ℓ
_⊆_ {A = A} U V = ((λ - → [ U - ]) ⊆⊆ (λ - → [ V - ])) , prop
  where
    prop : isProp ((x : A) → [ U x ] → [ V x ])
    prop = isPropΠ λ x → isPropΠ λ _ → is-true-prop (V x)

⊆-antisym : [ U ⊆ V ] → [ V ⊆ U ] → U ≡ V
⊆-antisym {U = U} {V} U⊆V V⊆V = funExt (λ x → ⇔toPath (U⊆V x) (V⊆V x))

entire : {A : Type ℓ} → 𝒫 A
entire {ℓ = ℓ} _ = Unit ℓ , Unit-prop

_∩_ : 𝒫 A → 𝒫 A → 𝒫 A
_∩_ {A = A} U V = λ x → ([ U x ] × [ V x ]) , prop x
  where
    prop : (x : A) → isProp ([ U x ] × [ V x ])
    prop x = isPropΣ (is-true-prop (x ∈ U)) λ _ → is-true-prop (V x)
```

Family

```
Fam : (ℓ₀ : Level) → Type ℓ₁ → Type (suc ℓ₀ ⊔ ℓ₁)
Fam ℓ₀ A = Σ (Set ℓ₀) (λ I → I → A)

index : Fam ℓ₁ A → Type ℓ₁
index (I , _) = I

-- Application of a family over X to an index.
_$_ : (ℱ : Fam ℓ₀ A) → index ℱ → A
_$_ (_ , f) = f

infixr 7 _$_

-- Membership for families.
_ε_ : A → Fam ℓ₁ A → Type _
x ε (_ , f) = fiber f x

-- Composition of a family with a function.
_<$>_ : {X : Type ℓ₀} {Y : Type ℓ₁} → (g : X → Y) → (ℱ : Fam ℓ₂ X) → Fam ℓ₂ Y
g <$> ℱ = (index ℱ) , g ∘ (_$_ ℱ)

fmap : {X : Type ℓ₀} {Y : Type ℓ₁} → (g : X → Y) → (ℱ : Fam ℓ₂ X) → Fam ℓ₂ Y
fmap = _<$>_
```

```
syntax fmap (λ x → e) ℱ = [ e | x ε ℱ ]

compr-:-syntax : {X : Type ℓ₀} → (I : Type ℓ₂) → (I → X) → Fam ℓ₂ X
compr-:-syntax I f = (I , f)

syntax compr-:-syntax I (λ i → e) = [ e | i : I ]

-- Forall quantification for families.
fam-forall : {X : Type ℓ₀} (ℱ : Fam ℓ₂ X) → (X → hProp ℓ₁) → hProp (ℓ₀ ⊔ ℓ₁ ⊔ ℓ₂)
fam-forall {X = X} ℱ P = ((x : X) → x ε ℱ → [ P x ]) , prop
  where
    prop : isProp ((x : X) → x ε ℱ → [ P x ])
    prop = isPropΠ λ x → isPropΠ λ _ → is-true-prop (P x)

syntax fam-forall ℱ (λ x → P) = ∀[ x ε ℱ ] P

-- Familification of a given powerset.
《_》 : {A : Type ℓ₀} → (A → hProp ℓ₁) → Fam (ℓ₀ ⊔ ℓ₁) A
《_》 {A = A} U = (Σ[ x ∈ A ] [ U x ]) , π₀
```

Truncation

```
data ‖_‖ (A : Type ℓ) : Type ℓ where
  |_| : A → ‖ A ‖
  squash : (x y : ‖ A ‖) → x ≡ y

‖‖-prop : (A : Type ℓ) → isProp ‖ A ‖
‖‖-prop _ = squash

‖‖-rec : {X Y : Type ℓ} → isProp Y → (X → Y) → ‖ X ‖ → Y
‖‖-rec Y-prop f | x | = f x
‖‖-rec Y-prop f (squash |x|₀ |x|₁ i) =
  Y-prop (‖‖-rec Y-prop f |x|₀) (‖‖-rec Y-prop f |x|₁) i
```

A.3 The Poset module

```
{-# OPTIONS --without-K --cubical --safe #-}

module Poset where

open import Basis
open import Cubical.Foundations.SIP renaming (SNS-≡ to SNS)
open import Cubical.Foundations.Equiv using (_≃⟨_⟩_) renaming (_■ to _QED)
```

Definition of poset

```
Order : (ℓ₁ : Level) → Type ℓ → Type (ℓ ⊔ suc ℓ₁)
Order ℓ₁ A = A → A → hProp ℓ₁

Order-set : (ℓ₁ : Level) (A : Type ℓ₀) → isSet (Order ℓ₁ A)
Order-set ℓ₁ A = isSetΠ2 λ _ _ → isSetHProp

isReflexive : {A : Type ℓ₀} → Order ℓ₁ A → hProp (ℓ₀ ⊔ ℓ₁)
isReflexive {A = X} _⊑_ =
  ((x : X) → [ x ⊑ x ]) , isPropΠ (λ x → is-true-prop (x ⊑ x))

isTransitive : {A : Type ℓ₀} → Order ℓ₁ A → hProp (ℓ₀ ⊔ ℓ₁)
isTransitive {ℓ₀ = ℓ₀} {ℓ₁ = ℓ₁} {A = X} _⊑_ = ⊑-trans , ⊑-trans-prop
  where
    ⊑-trans : Type (ℓ₀ ⊔ ℓ₁)
    ⊑-trans = ((x y z : X) → [ x ⊑ y ⇒ y ⊑ z ⇒ x ⊑ z ])

    ⊑-trans-prop : isProp ⊑-trans
    ⊑-trans-prop = isPropΠ3 λ x y z → is-true-prop (x ⊑ y ⇒ y ⊑ z ⇒ x ⊑ z)

isAntisym : {A : Type ℓ₀} → isSet A → Order ℓ₁ A → hProp (ℓ₀ ⊔ ℓ₁)
isAntisym {A = A} A-set _⊑_ = ⊑-antisym , ⊑-antisym-prop
  where
    ⊑-antisym = (x y : A) → [ x ⊑ y ] → [ y ⊑ x ] → x ≡ y

    ⊑-antisym-prop : isProp ⊑-antisym
    ⊑-antisym-prop = isPropΠ2 λ x y → isPropΠ2 λ _ _ → A-set x y

PosetAx : (A : Type ℓ₀) → Order ℓ₁ A → hProp (ℓ₀ ⊔ ℓ₁)
PosetAx {ℓ₀ = ℓ₀} {ℓ₁ = ℓ₁} A _⊑_ = isAPartialSet , isAPartialSet-prop
  where
    isAPartialSet =
      Σ[ A-set ∈ isSet A ] [ isReflexive _⊑_ ⊓ isTransitive _⊑_ ⊓ isAntisym A-set _⊑_ ]

    isAPartialSet-prop =
      isPropΣ isPropIsSet λ A-set →
        is-true-prop (isReflexive _⊑_ ⊓ isTransitive _⊑_ ⊓ isAntisym A-set _⊑_)

-- A poset structure with level `ℓ₁`.

PosetStr : (ℓ₁ : Level) → Type ℓ → Type (ℓ ⊔ suc ℓ₁)
PosetStr ℓ₁ A = Σ[ ⊑ ∈ Order ℓ₁ A ] [ PosetAx A ⊑ ]

PosetStr-set : (ℓ₁ : Level) (A : Type ℓ₀) → isSet (PosetStr ℓ₁ A)
PosetStr-set ℓ₁ A =
  isSetΣ (isSetΠ λ _ → isSetΠ λ _ → isSetHProp) λ _⊑_ →
  isSetΣ (isProp→isSet isPropIsSet) λ A-set →
    isProp→isSet (is-true-prop (isReflexive _⊑_ ⊓ isTransitive _⊑_ ⊓ isAntisym A-set _⊑_))

-- A poset with carrier level `ℓ₀` and relation level `ℓ₁`.

Poset : (ℓ₀ ℓ₁ : Level) → Type (suc ℓ₀ ⊔ suc ℓ₁)
Poset ℓ₀ ℓ₁ = Σ (Type ℓ₀) (PosetStr ℓ₁)
```

Projections

```
-- Given a poset `P`, `| P |ₚ` denotes its carrier set and `strₚ P` its order structure.

|_|ₚ : Poset ℓ₀ ℓ₁ → Type ℓ₀
| X , _ |ₚ = X

strₚ : (P : Poset ℓ₀ ℓ₁) → PosetStr ℓ₁ | P |ₚ
strₚ (_ , s) = s

-- We refer to to the order of `P` as `_⊑[ P ]_`.

rel : (P : Poset ℓ₀ ℓ₁) → | P |ₚ → | P |ₚ → hProp ℓ₁
rel (_ , _⊑_ , _) = _⊑_

infix 9 rel

syntax rel P x y = x ⊑[ P ] y

rel°ᵖ : (P : Poset ℓ₀ ℓ₁) → | P |ₚ → | P |ₚ → hProp ℓ₁
rel°ᵖ (_ , _⊑_ , _) x y = y ⊑ x

syntax rel°ᵖ P x y = x ⊒[ P ] y

-- Similarly, we define projections for the poset properties.

⊑[_]-refl : (P : Poset ℓ₀ ℓ₁) → (x : | P |ₚ) → [ x ⊑[ P ] x ]
⊑[_]-refl (_ , _ , _ , ⊑-refl , _) = ⊑-refl

⊑[_]-trans : (P : Poset ℓ₀ ℓ₁) (x y z : | P |ₚ)
           → [ x ⊑[ P ] y ] → [ y ⊑[ P ] z ] → [ x ⊑[ P ] z ]
⊑[_]-trans (_ , _ , _ , _ , ⊑-trans , _) = ⊑-trans

⊑[_]-antisym : (P : Poset ℓ₀ ℓ₁) (x y : | P |ₚ)
             → [ x ⊑[ P ] y ] → [ y ⊑[ P ] x ] → x ≡ y
⊑[_]-antisym (_ , _ , _ , _ , _ , ⊑-antisym) = ⊑-antisym

carrier-is-set : (P : Poset ℓ₀ ℓ₁) → isSet | P |ₚ
carrier-is-set (_ , _ , is-set , _) = is-set
```

Partial order reasoning

```
-- Some convenient notation for carrying out inequality reasoning.

module PosetReasoning (P : Poset ℓ₀ ℓ₁) where

  _⊑⟨_⟩_ : (x : | P |ₚ) {y z : | P |ₚ}
         → [ x ⊑[ P ] y ] → [ y ⊑[ P ] z ] → [ x ⊑[ P ] z ]
  _ ⊑⟨ p ⟩ q = ⊑[ P ]-trans _ _ _ p q

  _■ : (x : | P |ₚ) → [ x ⊑[ P ] x ]
```

```
  _■ = ⊑[ P ]-refl

  infixr 0 _⊑⟨_⟩_
  infix  1 _■

-- It is not convenient to have to keep applying `subst` for the show that two equal things
-- are below each other so let us make note of the following trivial fact.

≡⇒⊑ : (P : Poset ℓ₀ ℓ₁) → {x y : ∣ P ∣ₚ} → x ≡ y → [ x ⊑[ P ] y ]
≡⇒⊑ P {x = x} p = subst (λ z → [ x ⊑[ P ] z ]) p (⊑[ P ]-refl x)
```

Monotonic functions

```
-- We can define the notion preserving the order of a order structure for all types with
-- orders.

isOrderPreserving : (M : Σ (Type ℓ₀) (Order ℓ₁)) (N : Σ (Type ℓ₀′) (Order ℓ₁′))
                  → (π₀ M → π₀ N) → Type (ℓ₀ ⊔ ℓ₁ ⊔ ℓ₁′)
isOrderPreserving (A , _⊑₀_) (B , _⊑₁_) f = (x y : A) → [ x ⊑₀ y ] → [ f x ⊑₁ f y ]

-- Technically, this is called "monotonic" as well but we will reserve that term for posets.

isMonotonic : (P : Poset ℓ₀ ℓ₁) (Q : Poset ℓ₀′ ℓ₁′)
            → (∣ P ∣ₚ → ∣ Q ∣ₚ) → Type (ℓ₀ ⊔ ℓ₁ ⊔ ℓ₁′)
isMonotonic (A , (_⊑₀_ , _)) (B , (_⊑₁_ , _)) = isOrderPreserving (A , _⊑₀_) (B , _⊑₁_)

-- Both of these notions are propositional.

isOrderPreserving-prop : (M : Σ (Type ℓ₀) (Order ℓ₁)) (N : Σ (Type ℓ₀′) (Order ℓ₁′))
                         (f : π₀ M → π₀ N)
                       → isProp (isOrderPreserving M N f)
isOrderPreserving-prop M (_ , _⊑₁_) f = isPropΠ3 λ x y p → is-true-prop ((f x) ⊑₁ (f y))

isMonotonic-prop : (P : Poset ℓ₀ ℓ₁) (Q : Poset ℓ₀′ ℓ₁′) (f : ∣ P ∣ₚ → ∣ Q ∣ₚ)
                 → isProp (isMonotonic P Q f)
isMonotonic-prop (A , (_⊑₀_ , _)) (B , (_⊑₁_ , _)) f =
  isOrderPreserving-prop (A , _⊑₀_) (B , _⊑₁_) f

-- We then collect monotonic functions in the following type.

_-m→_ : Poset ℓ₀ ℓ₁ → Poset ℓ₀′ ℓ₁′ → Type (ℓ₀ ⊔ ℓ₁ ⊔ ℓ₀′ ⊔ ℓ₁′)
_-m→_ P Q = Σ (∣ P ∣ₚ → ∣ Q ∣ₚ) (isMonotonic P Q)

-- Projection for the underlying function of a monotonic map.

_$ₘ_ = π₀

-- The identity monotonic map and composition of monotonic maps.

1m : (P : Poset ℓ₀ ℓ₁) → P -m→ P
1m P = id ∣ P ∣ₚ , (λ x y x⊑y → x⊑y)

_∘m_ : {P : Poset ℓ₀ ℓ₁} {Q : Poset ℓ₀′ ℓ₁′} {R : Poset ℓ₀′′ ℓ₁′′}
```

```
    → (Q —m→ R) → (P —m→ Q) → (P —m→ R)
(g , pg) ∘m (f , pf) = g ∘ f , λ x y p → pg (f x) (f y) (pf x y p)


-- We will often deal with the task of showing the equality of two monotonic functions. As
-- being monotonic is propositional, we can quickly reduce this to showing the equality of
-- the underlying functions using `ΣProp≡` but it is more convenient to record this fact in
-- advance.

forget-mono : (P : Poset ℓ₀ ℓ₁) (Q : Poset ℓ₀′ ℓ₁′) ((f , f-mono) (g , g-mono) : P —m→ Q)
            → f ≡ g
            → (f , f-mono) ≡ (g , g-mono)
forget-mono P Q (f , f-mono) (g , g-mono) =
  ΣProp≡ (λ f → isPropΠ3 λ x y x⊑y → is-true-prop (f x ⊑[ Q ] f y))
```

Downward-closure

```
-- We denote by `↓[ P ] x` the type of everything in `P` that is below `x`.

↓[_]_ : (P : Poset ℓ₀ ℓ₁) → ∣ P ∣ₚ → Type (ℓ₀ ⊔ ℓ₁)
↓[ P ] a = Σ[ b ∈ ∣ P ∣ₚ ] [ b ⊑[ P ] a ]

isDownwardsClosed : (P : Poset ℓ₀ ℓ₁) → 𝒫 ∣ P ∣ₚ → hProp (ℓ₀ ⊔ ℓ₁)
isDownwardsClosed P U =
  ((x y : ∣ P ∣ₚ) → [ x ∈ U ] → [ y ⊑[ P ] x ] → [ y ∈ U ]) , prop
  where
    prop : isProp ((x y : ∣ P ∣ₚ) → [ U x ] → [ y ⊑[ P ] x ] → [ U y ])
    prop = isPropΠ λ _ → isPropΠ λ x → isPropΠ λ _ → isPropΠ λ _ → is-true-prop (x ∈ U)

DCSubset : (P : Poset ℓ₀ ℓ₁) → Type (suc ℓ₀ ⊔ ℓ₁)
DCSubset P = Σ[ U ∈ 𝒫 ∣ P ∣ₚ ] [ isDownwardsClosed P U ]

DCSubset-set : (P : Poset ℓ₀ ℓ₁) → isSet (DCSubset P)
DCSubset-set P =
  isSetΣ (𝒫-set ∣ P ∣ₚ) λ U → isProp→isSet (is-true-prop (isDownwardsClosed P U))
```

Product of two posets

```
_×ₚ_ : (P : Poset ℓ₀ ℓ₁) (Q : Poset ℓ₀′ ℓ₁′) → Poset (ℓ₀ ⊔ ℓ₀′) (ℓ₁ ⊔ ℓ₁′)
P ×ₚ Q = (∣ P ∣ₚ × ∣ Q ∣ₚ) , _⊑_ , carrier-set , (⊑-refl , ⊑-trans , ⊑-antisym)
  where
    _⊑_ : ∣ P ∣ₚ × ∣ Q ∣ₚ → ∣ P ∣ₚ × ∣ Q ∣ₚ → hProp _
    _⊑_ (x₀ , y₀) (x₁ , y₁) = x₀ ⊑[ P ] x₁ ⊓ y₀ ⊑[ Q ] y₁

    carrier-set : isSet (∣ P ∣ₚ × ∣ Q ∣ₚ)
    carrier-set = isSetΣ (carrier-is-set P) λ _ → (carrier-is-set Q)

    ⊑-refl : (p : ∣ P ∣ₚ × ∣ Q ∣ₚ) → [ p ⊑ p ]
```

```
  _⊑-refl (x , y) = (_⊑[ P ]-refl x) , (_⊑[ Q ]-refl y)

  _⊑-trans : (p q r : ⌊ P ⌋ₚ × ⌊ Q ⌋ₚ) → [ p ⊑ q ] → [ q ⊑ r ] → [ p ⊑ r ]
  _⊑-trans (x₀ , y₀) (x₁ , y₁) (x₂ , y₂) (x₀⊑x₁ , y₀⊑y₁) (x₁⊑x₂ , y₁⊑y₂) =
    _⊑[ P ]-trans _ _ _ x₀⊑x₁ x₁⊑x₂ , _⊑[ Q ]-trans _ _ _ y₀⊑y₁ y₁⊑y₂

  _⊑-antisym : (p q : ⌊ P ⌋ₚ × ⌊ Q ⌋ₚ) → [ p ⊑ q ] → [ q ⊑ p ] → p ≡ q
  _⊑-antisym (x₀ , y₀) (x₁ , y₁) (x₀⊑x₁ , y₀⊑y₁) (x₁⊑x₀ , y₁⊑y₀) =
    sigmaPath→pathSigma (x₀ , y₀) (x₁ , y₁) (_⊑[ P ]-antisym _ _ x₀⊑x₁ x₁⊑x₀ , sym NTS)
    where
      NTS : y₁ ≡ transport refl y₀
      NTS = subst (_≡_ y₁) (sym (transportRefl y₀)) (_⊑[ Q ]-antisym _ _ y₁⊑y₀ y₀⊑y₁)
```

Poset univalence

```
-- Now, we would like to show that ordered structures, as given by `Order`, are a standard
-- notion of structure. As we have already written down what it means for a function to be
-- order-preserving, we can express what it means for a *type equivalence* to be order
-- preserving.

isAnOrderPreservingEqv : (M N : Σ (Type ℓ₀) (Order ℓ₁)) → π₀ M ≃ π₀ N → Type (ℓ₀ ⊔ ℓ₁)
isAnOrderPreservingEqv M N e@(f , _) =
  isOrderPreserving M N f × isOrderPreserving N M g
  where
    g = equivFun (invEquiv e)

-- `Order` coupled with `isAnOrdePreservingEqv` gives us an SNS.

Order-is-SNS : SNS {ℓ} (Order ℓ₁) isAnOrderPreservingEqv
Order-is-SNS {ℓ = ℓ} {ℓ₁ = ℓ₁} {X = X} _⊑₀_ _⊑₁_ = f , record { equiv-proof = f-equiv }
  where
    f : isAnOrderPreservingEqv (X , _⊑₀_) (X , _⊑₁_) (idEquiv X) → _⊑₀_ ≡ _⊑₁_
    f e@(φ , ψ) = funExt₂ λ x y → ⇔toPath (φ x y) (ψ x y)

    g : _⊑₀_ ≡ _⊑₁_ → isAnOrderPreservingEqv (X , _⊑₀_) (X , _⊑₁_) (idEquiv X)
    g p =
      subst
        (λ _⊑_ → isAnOrderPreservingEqv (X , _⊑₀_) (X , _⊑_) (idEquiv X))
        p
        ((λ _ _ x⊑₁y → x⊑₁y) , λ _ _ x⊑₁y → x⊑₁y)

    ret-f-g : retract f g
    ret-f-g (φ , ψ) =
      isPropΣ
        (isOrderPreserving-prop (X , _⊑₀_) (X , _⊑₁_) (id X))
        (λ _ → isOrderPreserving-prop (X , _⊑₁_) (X , _⊑₀_) (id X))
        (g (f (φ , ψ))) (φ , ψ)

    f-equiv : (p : _⊑₀_ ≡ _⊑₁_) → isContr (fiber f p)
    f-equiv p = ((to , from) , eq) , NTS
      where
        to : isOrderPreserving (X , _⊑₀_) (X , _⊑₁_) (id _)
```

```
        to x y = subst (λ _⊑_ → [ x ⊑₀ y ] → [ x ⊑ y ]) p (id _)

        from : isOrderPreserving (X , _⊑₁_) (X , _⊑₀_) (id _)
        from x y = subst (λ _⊑_ → [ x ⊑ y ] → [ x ⊑₀ y ]) p (id _)

        eq : f (to , from) ≡ p
        eq = Order-set ℓ₁ X _⊑₀_ _⊑₁_ (f (to , from)) p

        NTS : (fib : fiber f p) → ((to , from) , eq) ≡ fib
        NTS ((φ , ψ) , eq) =
          ΣProp≡
            (λ i' → isOfHLevelSuc 2 (Order-set ℓ₁ X) _⊑₀_ _⊑₁_ (f i') p)
            (ΣProp≡
              (λ _ → isOrderPreserving-prop (X , _⊑₁_) (X , _⊑₀_) (id _))
              (isOrderPreserving-prop (X , _⊑₀_) (X , _⊑₁_) (id _) to φ))

-- This is the substantial part of the work required to establish univalence for posets.
-- Adding partial order axioms on top of this is not too hard.

-- First, let us define what is means for a type equivalence to be monotonic.

isAMonotonicEqv : (P Q : Poset ℓ₀ ℓ₁) → ⟨ P ⟩₀ ≃ ⟨ Q ⟩₀ → Type (ℓ₀ ⊔ ℓ₁)
isAMonotonicEqv (A , (_⊑₀_ , _)) (B , (_⊑₁_ , _)) =
  isAnOrderPreservingEqv (A , _⊑₀_) (B , _⊑₁_)

isAMonotonicEqv-prop : (P Q : Poset ℓ₀ ℓ₁) (eqv : ⟨ P ⟩₀ ≃ ⟨ Q ⟩₀)
                     → isProp (isAMonotonicEqv P Q eqv)
isAMonotonicEqv-prop P Q e@(f , _) =
  isPropΣ (isMonotonic-prop P Q f) λ _ → isMonotonic-prop Q P g
  where
    g = equivFun (invEquiv e)

-- We denote by `_≃ₚ_` the type of monotonic poset equivalences.

_≃ₚ_ : Poset ℓ₀ ℓ₁ → Poset ℓ₀ ℓ₁ → Type (ℓ₀ ⊔ ℓ₁)
_≃ₚ_ P Q = Σ[ i ∈ ⟨ P ⟩₀ ≃ ⟨ Q ⟩₀ ] isAMonotonicEqv P Q i

-- From this, we can already establish that posets form an SNS and prove that the category
-- of posets is univalent.

poset-is-SNS : SNS {ℓ} (PosetStr ℓ₁) isAMonotonicEqv
poset-is-SNS {ℓ₁ = ℓ₁} =
  SNS-PathP→SNS-≡
    (PosetStr ℓ₁)
    isAMonotonicEqv
    (add-axioms-SNS _ NTS (SNS-≡→SNS-PathP isAnOrderPreservingEqv Order-is-SNS))
  where
    NTS : (A : Type ℓ) (_⊑_ : Order ℓ₁ A) → isProp [ PosetAx A _⊑_ ]
    NTS A _⊑_ = is-true-prop (PosetAx A _⊑_)

poset-univ₀ : (P Q : Poset ℓ₀ ℓ₁) → (P ≃ₚ Q) ≃ (P ≡ Q)
poset-univ₀ = SIP (SNS-≡→SNS-PathP isAMonotonicEqv poset-is-SNS)

-- This result is almost what we want but it is better talk directly about poset
-- _isomorphisms_ rather than equivalences. In the case when types `A` and `B` are sets, the
-- type of isomorphisms between `A` and `B` is equivalent to the type of equivalences betwee
-- them.
```

```
-- Let us start by writing down what a poset isomorphisms is.

isPosetIso : (P Q : Poset ℓ₀ ℓ₁) → (P →m Q) → Type (ℓ₀ ⊔ ℓ₁)
isPosetIso P Q (f , _) = Σ[ (g , _) ∈ (Q →m P) ] section f g × retract f g

isPosetIso-prop : (P Q : Poset ℓ₀ ℓ₁) (f : P →m Q)
                → isProp (isPosetIso P Q f)
isPosetIso-prop P Q (f , f-mono) (g₀ , sec₀ , ret₀) (g₁ , sec₁ , ret₁) =
  ΣProp≡ NTS g₀=g₁
  where
    NTS : ((g , _) : Q →m P) → isProp (section f g × retract f g)
    NTS (g , g-mono) = isPropΣ
                         (isPropΠ λ x → carrier-is-set Q (f (g x)) x) λ _ →
                           isPropΠ λ x → carrier-is-set P (g (f x)) x

    g₀=g₁ : g₀ ≡ g₁
    g₀=g₁ =
      forget-mono Q P g₀ g₁ (funExt λ x →
        π₀ g₀ x ≡⟨ sym (cong (λ - → π₀ g₀ -) (sec₁ x)) ⟩
        π₀ g₀ (f (π₀ g₁ x)) ≡⟨ ret₀ (π₀ g₁ x) ⟩
        π₀ g₁ x ∎)

-- We will denote by `P ≅ₚ Q` the type of isomorphisms between posets `P` and `Q`.

_≅ₚ_ : Poset ℓ₀ ℓ₁ → Poset ℓ₀ ℓ₁ → Type (ℓ₀ ⊔ ℓ₁)
P ≅ₚ Q = Σ[ f ∈ P →m Q ] isPosetIso P Q f

-- As we have mentioned before, `P ≅ₚ Q` is equivalent to `P ≃ₚ Q`.

≃ₚ≃≅ₚ : (P Q : Poset ℓ₀ ℓ₁) → (P ≅ₚ Q) ≃ (P ≃ₚ Q)
≃ₚ≃≅ₚ P Q = isoToEquiv (iso from to ret sec)
  where
    to : P ≃ₚ Q → P ≅ₚ Q
    to (e@(f , _) , (f-mono , g-mono)) = (f , f-mono) , (g , g-mono) , sec-f-g , ret-f-g
      where
        is = equivToIso e
        g = equivFun (invEquiv e)

        sec-f-g : section f g
        sec-f-g = Iso.rightInv (equivToIso e)

        ret-f-g : retract f g
        ret-f-g = Iso.leftInv (equivToIso e)

    from : P ≅ₚ Q → P ≃ₚ Q
    from ((f , f-mono) , ((g , g-mono) , sec , ret)) = isoToEquiv is , f-mono , g-mono
      where
        is : Iso ∣ P ∣ₚ ∣ Q ∣ₚ
        is = iso f g sec ret

    sec : section to from
    sec (f , _) = ΣProp≡ (isPosetIso-prop P Q) refl

    ret : retract to from
    ret (e , _) = ΣProp≡ (isAMonotonicEqv-prop P Q) (ΣProp≡ isPropIsEquiv refl)

-- Once this equivalence has been established, the main result follows easily: *the category
-- of posets is univalent*.

poset-univ : (P Q : Poset ℓ₀ ℓ₁) → (P ≅ₚ Q) ≃ (P ≡ Q)
poset-univ P Q = P ≅ₚ Q ≃⟨ ≃ₚ≃≅ₚ P Q ⟩ P ≃ₚ Q ≃⟨ poset-univ₀ P Q ⟩ P ≡ Q ∎
```

A.4 The Frame module

```
{-# OPTIONS --without-K --cubical --safe #-}

module Frame where

open import Basis                       hiding (A)
open import Cubical.Foundations.Function using (uncurry)
open import Cubical.Foundations.SIP                  renaming (SNS-≡ to SNS)
open import Cubical.Foundations.Equiv using (_≃⟨_⟩_) renaming (_■ to _QED)
open import Poset

module JoinSyntax (A : Type ℓ₀) {ℓ₂ : Level} (join : Fam ℓ₂ A → A) where

  join-of : {I : Type ℓ₂} → (I → A) → A
  join-of {I = I} f = join (I , f)

  syntax join-of (λ i → e) = ⋁⟨ i ⟩ e


RawFrameStr : (ℓ₁ ℓ₂ : Level) → Type ℓ₀ → Type (ℓ₀ ⊔ suc ℓ₁ ⊔ suc ℓ₂)
RawFrameStr ℓ₁ ℓ₂ A = PosetStr ℓ₁ A × A × (A → A → A) × (Fam ℓ₂ A → A)

pos-of-raw-frame : (Σ[ A ∈ Type ℓ₀ ] RawFrameStr ℓ₁ ℓ₂ A) → Poset ℓ₀ ℓ₁
pos-of-raw-frame (A , ps , _) = A , ps

RawFrameStr-set : (ℓ₁ ℓ₂ : Level) (A : Type ℓ₀)
                → isSet (RawFrameStr ℓ₁ ℓ₂ A)
RawFrameStr-set ℓ₁ ℓ₂ A = isSetΣ (PosetStr-set ℓ₁ A) NTS
  where
    NTS : _
    NTS pos = isSetΣ A-set λ _ →
              isSetΣ (isSetΠ2 λ _ _ → A-set) λ _ →
              isSetΠ λ _ → A-set
      where
        A-set : isSet A
        A-set = carrier-is-set (A , pos)

isTop : (P : Poset ℓ₀ ℓ₁) → ⟨ P ⟩₊ → hProp (ℓ₀ ⊔ ℓ₁)
isTop P x = ((y : ⟨ P ⟩₊) → [ y ⊑[ P ] x ]) , isPropΠ λ y → is-true-prop (y ⊑[ P ] x)

isGLB : (P : Poset ℓ₀ ℓ₁) → (⟨ P ⟩₊ → ⟨ P ⟩₊ → ⟨ P ⟩₊) → hProp (ℓ₀ ⊔ ℓ₁)
isGLB P _∧_ = ∧-GLB , ∧-GLB-prop
  where
    ∧-GLB = -- x ∧ y is a lower bound of {x, y}.
        ((x y   : ⟨ P ⟩₊) → [ (x ∧ y) ⊑[ P ] x ⊓ (x ∧ y) ⊑[ P ] y ])
        -- Given any other lower bound z of {x, y}, x ∧ y is _greater_ than that.
      × ((x y z : ⟨ P ⟩₊) → [ (z ⊑[ P ] x ⊓ z ⊑[ P ] y) ⇒ z ⊑[ P ] (x ∧ y) ])

    ∧-GLB-prop : isProp ∧-GLB
    ∧-GLB-prop =
      isPropΣ
        (isPropΠ2 λ x y → is-true-prop ((x ∧ y) ⊑[ P ] x ⊓ (x ∧ y) ⊑[ P ] y)) λ _ →
        isPropΠ3 λ x y z → is-true-prop (z ⊑[ P ] x ⊓ z ⊑[ P ] y ⇒ z ⊑[ P ] (x ∧ y))

isLUB : (P : Poset ℓ₀ ℓ₁) → (Fam ℓ₂ ⟨ P ⟩₊ → ⟨ P ⟩₊) → hProp (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂)
```

```
isLUB {ℓ₂ = ℓ₂} P ⋁_ = ⋁-LUB , ⋁-LUB-prop
  where
    ⋁-LUB = ((U : Fam ℓ₂ ∣ P ∣ₚ) → [ ∀[ x ∈ U ] (x ⊑[ P ] ⋁ U) ])
          × ((U : Fam ℓ₂ ∣ P ∣ₚ) (x : _) → [ (∀[ y ∈ U ] (y ⊑[ P ] x)) ⇒ ⋁ U ⊑[ P ] x ])

    ⋁-LUB-prop : isProp ⋁-LUB
    ⋁-LUB-prop = isPropΣ
                   (λ ψ ϑ → funExt λ U →
                     is-true-prop (∀[ y ∈ U ] (y ⊑[ P ] ⋁ U)) (ψ U) (ϑ U)) λ _ →
                   isPropΠ λ U → isPropΠ λ x →
                     is-true-prop (∀[ y ∈ U ] (y ⊑[ P ] x) ⇒ (⋁ U) ⊑[ P ] x)

isDist : (P : Poset ℓ₀ ℓ₁)
       → (∣ P ∣ₚ → ∣ P ∣ₚ → ∣ P ∣ₚ)
       → (Fam ℓ₂ ∣ P ∣ₚ → ∣ P ∣ₚ)
       → hProp (ℓ₀ ⊔ suc ℓ₂)
isDist {ℓ₂ = ℓ₂} P _⊓_ ⋁_ = ∧-dist-over-⋁ , ∧-dist-over-⋁-prop
  where
    open JoinSyntax ∣ P ∣ₚ ⋁_

    ∧-dist-over-⋁ = (x : ∣ P ∣ₚ) (U : Fam ℓ₂ ∣ P ∣ₚ) → x ⊓ (⋁ U) ≡ ⋁⟨ i ⟩ (x ⊓ (U $ i))

    ∧-dist-over-⋁-prop : isProp ∧-dist-over-⋁
    ∧-dist-over-⋁-prop p q = funExt₂ λ x U → carrier-is-set P _ _ (p x U) (q x U)

FrameAx : {A : Type ℓ₀} → RawFrameStr ℓ₁ ℓ₂ A → hProp (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂)
FrameAx {ℓ₀ = ℓ₀} {ℓ₁ = ℓ₁} {A = A} (s@(_⊑_ , _) , ⊤ , _∧_ , ⋁_) =
  isTop P ⊤ ⊓ isGLB P _∧_ ⊓ isLUB P ⋁_ ⊓ isDist P _∧_ ⋁_
  where
    P : Poset ℓ₀ ℓ₁
    P = A , s

FrameStr : (ℓ₁ ℓ₂ : Level) → Type ℓ₀ → Type (ℓ₀ ⊔ suc ℓ₁ ⊔ suc ℓ₂)
FrameStr ℓ₁ ℓ₂ A    = Σ[ s ∈ RawFrameStr ℓ₁ ℓ₂ A ] [ FrameAx s ]

Frame : (ℓ₀ ℓ₁ ℓ₂ : Level) → Type (suc ℓ₀ ⊔ suc ℓ₁ ⊔ suc ℓ₂)
Frame ℓ₀ ℓ₁ ℓ₂ = Σ[ A ∈ Type ℓ₀ ] FrameStr ℓ₁ ℓ₂ A

-- Projection for the carrier set of a frame
-- i.e., the carrier set of the underlying poset.
∣_∣F : Frame ℓ₀ ℓ₁ ℓ₂ → Type ℓ₀
∣ A , _ ∣F = A

-- The underlying poset of a frame.
pos : Frame ℓ₀ ℓ₁ ℓ₂ → Poset ℓ₀ ℓ₁
pos (A , (P , _) , _) = A , P

-- Projections for the top element, meet, and join of a frame.

⊤[_] : (F : Frame ℓ₀ ℓ₁ ℓ₂) → ∣ F ∣F
⊤[ _ , (_ , (⊤ , _)) , _ ] = ⊤

glb-of : (F : Frame ℓ₀ ℓ₁ ℓ₂) → ∣ F ∣F → ∣ F ∣F → ∣ F ∣F
glb-of (_ , (_ , _ , _⊓_ , _) , _) = _⊓_

syntax glb-of F x y = x ⊓[ F ] y
```

```
⋁[_]_ : (F : Frame ℓ₀ ℓ₁ ℓ₂) → Fam ℓ₂ ∣ F ∣F → ∣ F ∣F
⋁[ (_ , (_ , (_ , _ , ⋁_)) , _) ] U = ⋁ U

-- Projections for frame laws.

module _ (F : Frame ℓ₀ ℓ₁ ℓ₂) where
  private
    P = pos F

    _⊑_ : ∣ F ∣F → ∣ F ∣F → hProp ℓ₁
    x ⊑ y = x ⊑[ P ] y

    open JoinSyntax ∣ F ∣F (λ _ → ⋁[ F ] _)

  ⊤[_]-top : (x : ∣ F ∣F) → [ x ⊑ ⊤[ F ] ]
  ⊤[_]-top = let (_ , _ , frame-str) = F in π₀ frame-str

  ⊓[_]-lower₀ : (x y : ∣ F ∣F) → [ (x ⊓[ F ] y) ⊑ x ]
  ⊓[_]-lower₀ = let (_ , _ , str) = F in λ x y → π₀ (π₀ (π₁ str)) x y


  ⊓[_]-lower₁ : (x y : ∣ F ∣F) → [ (x ⊓[ F ] y) ⊑ y ]
  ⊓[_]-lower₁ = let (_ , _ , str) = F in λ x y → π₁ (π₀ (π₁ str)) x y

  ⊓[_]-greatest : (x y z : ∣ F ∣F) → [ z ⊑ x ] → [ z ⊑ y ] → [ z ⊑ (x ⊓[ F ] y) ]
  ⊓[_]-greatest =
    let (_ , _ , str) = F in λ x y z z⊑x z⊑y → π₁ (π₀ (π₁ str)) x y z (z⊑x , z⊑y)

  ⋁[_]-upper : (U : Fam ℓ₂ ∣ F ∣F) (o : ∣ F ∣F) → o ∈ U → [ o ⊑ (⋁[ F ] U) ]
  ⋁[_]-upper = let (_ , _ , str) = F in π₀ (π₀ (π₁ (π₁ str)))

  ⋁[_]-least : (U : Fam ℓ₂ ∣ F ∣F) (x : ∣ F ∣F)
             → [ ∀[ y ∈ U ] (y ⊑ x) ] → [ (⋁[ F ] U) ⊑ x ]
  ⋁[_]-least = let (_ , _ , str) = F in π₁ (π₀ (π₁ (π₁ str)))

  dist : (x : ∣ F ∣F) (U : Fam ℓ₂ ∣ F ∣F)
       → x ⊓[ F ] (⋁⟨ i ⟩ (U $ i)) ≡ ⋁⟨ i ⟩ (x ⊓[ F ] (U $ i))
  dist = let (_ , _ , str) = F in π₁ (π₁ (π₁ str))

  top-unique : (y : ∣ F ∣F) → ((x : ∣ F ∣F) → [ x ⊑ y ]) → y ≡ ⊤[ F ]
  top-unique y y-top = ⊑[ pos F ]-antisym y ⊤[ F ] (⊤[_]-top y) (y-top ⊤[ F ])

  ⊓-unique : (x y z : ∣ F ∣F)
           → [ z ⊑ x ] → [ z ⊑ y ] → ((w : ∣ F ∣F) → [ w ⊑ x ] → [ w ⊑ y ] → [ w ⊑ z ])
           → z ≡ x ⊓[ F ] y
  ⊓-unique x y z z⊑x z⊑y greatest =
    ⊑[ P ]-antisym z (x ⊓[ F ] y) (⊓[_]-greatest x y z z⊑x z⊑y) NTS
    where
      NTS : [ (x ⊓[ F ] y) ⊑ z ]
      NTS = greatest (x ⊓[ F ] y) (⊓[_]-lower₀ x y) (⊓[_]-lower₁ x y)

  ⋁-unique : (U : Fam ℓ₂ ∣ F ∣F) (z : ∣ F ∣F)
           → ((x : ∣ F ∣F) → x ∈ U → [ x ⊑ z ])
           → ((w : ∣ F ∣F) → ((o : ∣ F ∣F) → o ∈ U → [ o ⊑ w ]) → [ z ⊑ w ])
           → z ≡ ⋁[ F ] U
  ⋁-unique U z upper least =
    ⊑[ P ]-antisym z (⋁[ F ] U) (least (⋁[ F ] U) (⋁[_]-upper U)) NTS
```

```
  where
    NTS : [ (⋁[ F ] U) ⊑ z ]
    NTS = ⋁[_]-least U z upper

x⊑y⇒x=x∧y : {x y : ∣ F ∣F}
          → [ x ⊑ y ] → x ≡ x ⊓[ F ] y
x⊑y⇒x=x∧y {x} {y} x⊑y = ⊑[ pos F ]-antisym _ _ down up
  where
    down : [ x ⊑ (x ⊓[ F ] y) ]
    down = ⊓[_]-greatest x y x (⊑[_]-refl P x) x⊑y

    up : [ (x ⊓[ F ] y) ⊑ x ]
    up = ⊓[_]-lower₀ x y

x=x∧y⇒x⊑y : {x y : ∣ F ∣F}
          → x ≡ x ⊓[ F ] y → [ x ⊑ y ]
x=x∧y⇒x⊑y {x} {y} eq = x ⊑⟨ ≡⇒⊑ P eq ⟩ x ⊓[ F ] y ⊑⟨ ⊓[_]-lower₁ x y ⟩ y ■
  where
    open PosetReasoning (pos F)

comm : (x y : ∣ F ∣F) → x ⊓[ F ] y ≡ y ⊓[ F ] x
comm x y = ⊓-unique y x _ (⊓[_]-lower₁ x y) (⊓[_]-lower₀ x y) NTS
  where
    NTS = λ w w⊑y w⊑x → ⊓[_]-greatest x y w w⊑x w⊑y

family-iff : {U V : Fam ℓ₂ ∣ F ∣F}
           → ((x : ∣ F ∣F) → (x ∈ U → x ∈ V) × (x ∈ V → x ∈ U))
           → ⋁[ F ] U ≡ ⋁[ F ] V
family-iff {U = U} {V = V} h = ⋁-unique _ _ ub least
  where
    ub : (o : ∣ F ∣F) → o ∈ V → [ o ⊑ (⋁[ F ] U) ]
    ub o (i , p) =
      subst (λ - → [ - ⊑ _ ]) p (⋁[ _ ]-upper _ (π₁ (h (V $ i)) (i , refl)))

    least : (w : ∣ F ∣F)
          → ((o : ∣ F ∣F) → o ∈ V → [ o ⊑ w ])
          → [ (⋁[ F ] U) ⊑ w ]
    least w f = ⋁[ _ ]-least _ λ o o∈U → f o (π₀ (h o) o∈U)

flatten : (I : Type ℓ₂) (J : I → Type ℓ₂) (f : (i : I) → J i → ∣ F ∣F)
        → ⋁[ F ] (Σ I J , uncurry f) ≡ ⋁[ F ] [ ⋁[ F ] [ f i j ∣ j ∶ J i ] ∣ i ∶ I ]
flatten I J f = ⊑[ pos F ]-antisym _ _ down up
  where
    open PosetReasoning (pos F)

    LHS = ⋁[ F ] (Σ I J , uncurry f)
    RHS = ⋁[ F ] (I , (λ i → ⋁[ F ] (J i , f i)))

    down : [ LHS ⊑ RHS ]
    down = ⋁[_]-least _ _ isUB
      where
        isUB : (x : ∣ F ∣F) → x ∈ (Σ I J , uncurry f) → [ x ⊑ RHS ]
        isUB x ((i , j) , eq) =
            x                    ⊑⟨ ≡⇒⊑ (pos F) (sym eq) ⟩
            f i j                ⊑⟨ ⋁[_]-upper _ _ (j , refl) ⟩
            ⋁[ F ] (J i , λ - → f i -) ⊑⟨ ⋁[_]-upper _ _ (i , refl) ⟩
            RHS                  ■
```

```
      up : [ RHS ⊑ LHS ]
      up = ⋁[_]-least _ _ isUB
        where
          isUB : (x : ∣ F ∣F)
               → x ∈ [ ⋁[ F ] (J i , f i) ∣ i ∶ I ] → [ x ⊑ (⋁[ F ] (Σ I J , uncurry f)) ]
          isUB x (i , p) =
            x ⊑⟨ ≡⇒⊑ (pos F) (sym p) ⟩
            ⋁[ F ] [ f i j ∣ j ∶ J i ] ⊑⟨ ⋁[_]-least _ _ isUB′ ⟩
            ⋁[ F ] (Σ I J , uncurry f) ∎
            where
              isUB′ : (z : ∣ F ∣F) → z ∈ [ f i j ∣ j ∶ J i ] → [ z ⊑ LHS ]
              isUB′ z (j , q) = ⋁[_]-upper _ _ ((i , j) , q)

  sym-distr : (U@(I , _) V@(J , _) : Fam ℓ₂ ∣ F ∣F)
            → (⋁⟨ i ⟩ (U $ i)) ⊓[ F ] (⋁⟨ i ⟩ (V $ i))
            ≡ ⋁[ F ] [ (U $ i) ⊓[ F ] (V $ j) ∣ (i , j) ∶ (I × J) ]
  sym-distr U@(I , _) V@(J , _) =
    (⋁[ F ] U) ⊓[ F ] (⋁[ F ] V)
      ≡⟨ dist (⋁[ F ] U) V ⟩
    ⋁[ F ] ((λ - → (⋁[ F ] U) ⊓[ F ] -) <$> V)
      ≡⟨ cong (λ - → ⋁[ F ] (- <$> V)) NTS₀ ⟩
    ⋁[ F ] ((λ x → x ⊓[ F ] (⋁[ F ] U)) <$> V)
      ≡⟨ cong (λ - → ⋁[ F ] (- <$> V)) NTS₁ ⟩
    ⋁[ F ] ((λ x → ⋁[ F ] ((λ y → x ⊓[ F ] y) <$> U)) <$> V)
      ≡⟨ sym (flatten (index V) (λ _ → index U) λ j i → (V $ j) ⊓[ F ] (U $ i)) ⟩
    ⋁[ F ] [ (V $ j) ⊓[ F ] (U $ i) ∣ (j , i) ∶ (J × I) ]
      ≡⟨ family-iff NTS₂ ⟩
    ⋁[ F ] [ (U $ i) ⊓[ F ] (V $ j) ∣ (i , j) ∶ (I × J) ]
      ∎
    where
      open PosetReasoning (pos F)

      NTS₀ : (λ - → (⋁[ F ] U) ⊓[ F ] -) ≡ (λ - → - ⊓[ F ] (⋁[ F ] U))
      NTS₀ = funExt λ x → comm (⋁[ F ] U) x

      NTS₁ : (λ - → - ⊓[ F ] (⋁[ F ] U)) ≡ (λ - → ⋁[ F ] ((λ y → - ⊓[ F ] y) <$> U))
      NTS₁ = funExt λ x → dist x U

      NTS₂ : _
      NTS₂ x = down , up
        where
          down : _
          down ((j , i) , eq) =
            subst
              (λ - → x ∈ (_ , -))
              (funExt (λ { (i′ , j′) → comm (V $ j′) (U $ i′) })) ((i , j) , eq)

          up : _
          up ((i , j) , eq) =
            subst
              (λ - → x ∈ (_ , -))
              (funExt (λ { (j′ , i′) → comm (U $ i′) (V $ j′) })) ((j , i) , eq)

isRawFrameHomo : (M : Σ[ A ∈ Type ℓ₀ ] RawFrameStr ℓ₁ ℓ₂ A)
                 (N : Σ[ B ∈ Type ℓ₀′ ] RawFrameStr ℓ₁′ ℓ₂ B)
```

```
                → let M-pos = pos-of-raw-frame M ; N-pos = pos-of-raw-frame N in
                  (M-pos —m→ N-pos) → Type (ℓ₀ ⊔ suc ℓ₂ ⊔ ℓ₀′)
isRawFrameHomo M@(A , ps₀ , ⊤₀ , _∧₀_ , V₀_) N@(B , ps₁ , ⊤₁ , _∧₁_ , V₁_) (f , f-mono) =
  resp-⊤ × resp-∧ × resp-V
  where
    resp-⊤ : Type _
    resp-⊤ = f ⊤₀ ≡ ⊤₁

    resp-∧ : Type _
    resp-∧ = (x y : A) → f (x ∧₀ y) ≡ (f x) ∧₁ (f y)

    resp-V : Type _
    resp-V = (U : Fam _ A) → f (V₀ U) ≡ V₁ [ f x | x ∈ U ]

isRawFrameHomo-prop : (M : Σ[ A ∈ Type ℓ₀ ] RawFrameStr ℓ₁ ℓ₂ A)
                      (N : Σ[ B ∈ Type ℓ₀′ ] RawFrameStr ℓ₁′ ℓ₂ B)
                    → let M-pos = pos-of-raw-frame M ; N-pos = pos-of-raw-frame N in
                      (f : M-pos —m→ N-pos) → isProp (isRawFrameHomo M N f)
isRawFrameHomo-prop M N (f , f-mono) =
  isPropΣ (B-set _ _) λ _ →
  isPropΣ (λ x y → funExt₂ λ a b → B-set _ _ (x a b) (y a b) ) λ _ →
          λ _ _ → funExt λ x → B-set _ _ _ _
  where
    M-pos = pos-of-raw-frame M
    N-pos = pos-of-raw-frame N
    A-set = carrier-is-set M-pos
    B-set = carrier-is-set N-pos

-- Frame homomorphisms.
isFrameHomomorphism : (F : Frame ℓ₀ ℓ₁ ℓ₂) (G : Frame ℓ₀′ ℓ₁′ ℓ₂)
                    → (pos F —m→ pos G) → Type (ℓ₀ ⊔ suc ℓ₂ ⊔ ℓ₀′)
isFrameHomomorphism (A , rs , _) (B , rs′ , _) = isRawFrameHomo (A , rs) (B , rs′)

isFrameHomomorphism-prop : (F : Frame ℓ₀ ℓ₁ ℓ₂) (G : Frame ℓ₀′ ℓ₁′ ℓ₂)
                         → (f : pos F —m→ pos G) → isProp (isFrameHomomorphism F G f)
isFrameHomomorphism-prop (A , s , _) (B , s′ , _) = isRawFrameHomo-prop (A , s) (B , s′)

_—f→_ : Frame ℓ₀ ℓ₁ ℓ₂ → Frame ℓ₀′ ℓ₁′ ℓ₂ → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂ ⊔ ℓ₀′ ⊔ ℓ₁′)
_—f→_ {ℓ₂ = ℓ₂} F G = Σ[ f ∈ (pos F —m→ pos G) ] (isFrameHomomorphism F G f)

_$f_ : {F : Frame ℓ₀ ℓ₁ ℓ₂} {G : Frame ℓ₀′ ℓ₁′ ℓ₂} → F —f→ G → ⌞ F ⌟ → ⌞ G ⌟
(f , _) $f x = f $ₘ x

isFrameIso : {F : Frame ℓ₀ ℓ₁ ℓ₂} {G : Frame ℓ₀′ ℓ₁′ ℓ₂}
           → (F —f→ G) → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂ ⊔ ℓ₀′ ⊔ ℓ₁′)
isFrameIso {F = F} {G} ((f , _) , _) =
  Σ[ ((g , _) , _) ∈ (G —f→ F) ] section f g × retract f g

isFrameIso-prop : {F : Frame ℓ₀ ℓ₁ ℓ₂} {G : Frame ℓ₀′ ℓ₁′ ℓ₂}
                → (f : F —f→ G) → isProp (isFrameIso {F = F} {G} f)
isFrameIso-prop {F = F} {G} ((f , _) , _) (g₀h , sec₀ , ret₀) (g₁h , sec₁ , ret₁) =
  ΣProp≡ NTS₀ NTS₁
  where
    g₀ = _$f_ {F = G} {F} g₀h
    g₁ = _$f_ {F = G} {F} g₁h

    NTS₀ : (((g , _) , _) : G —f→ F) → isProp (section f g × retract f g)
```

```
    NTS₀ ((g , _) , g-homo) =
      isPropΣ (λ s s' → funExt λ x → carrier-is-set (pos G) _ _ (s x) (s' x)) λ _ r r' →
      funExt λ y → carrier-is-set (pos F) _ _ (r y) (r' y)

    g₀~g₁ : g₀ ~ g₁
    g₀~g₁ x = g₀ x           ≡⟨ cong g₀ (sym (sec₁ x)) ⟩
              g₀ (f (g₁ x)) ≡⟨ ret₀ (g₁ x) ⟩
              g₁ x           ∎

    NTS₁ : g₀h ≡ g₁h
    NTS₁ = ΣProp≡
             (isFrameHomomorphism-prop G F)
             (forget-mono (pos G) (pos F) (π₀ g₀h) (π₀ g₁h) (funExt g₀~g₁))

_≅f_ : (F : Frame ℓ₀ ℓ₁ ℓ₂) (G : Frame ℓ₀' ℓ₁' ℓ₂) → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂ ⊔ ℓ₀' ⊔ ℓ₁')
F ≅f G = Σ[ f ∈ F –f→ G ] isFrameIso {F = F} {G} f

-- An element of the poset is like a finite observation whereas an element of the
-- frame of downward closed posets is like a general observation.

-- The set of downward-closed subsets of a poset forms a frame.
DCPoset : (P : Poset ℓ₀ ℓ₁) → Poset (suc ℓ₀ ⊔ ℓ₁) ℓ₀
DCPoset {ℓ₀ = ℓ₀} P = 𝔻 , _≪_ , 𝔻-set , ≪-refl , ≪-trans , ≪-antisym
  where
    𝔻     = DCSubset     P
    𝔻-set = DCSubset-set P

    _≪_ : 𝔻 → 𝔻 → hProp ℓ₀
    _≪_ (S , _) (T , _) = S ⊆ T

    abstract
      ≪-refl : [ isReflexive _≪_ ]
      ≪-refl (U , U-down) x xϵU = xϵU

      ≪-trans : [ isTransitive _≪_ ]
      ≪-trans _ _ _ S≪T T≪U x xϵS = T≪U x (S≪T x xϵS)

      ≪-antisym : [ isAntisym 𝔻-set _≪_ ]
      ≪-antisym X Y S⊆T T⊆S =
        ΣProp≡ (is-true-prop ∘ isDownwardsClosed P) (⊆-antisym S⊆T T⊆S)

-- The set of downward-closed subsets of a poset forms a frame.
DCFrame : (P : Poset ℓ₀ ℓ₁) → Frame (suc ℓ₀ ⊔ ℓ₁) ℓ₀ ℓ₀
DCFrame {ℓ₀ = ℓ₀} {ℓ₁ = ℓ₁} (X , P) =
    𝔻
  , (str ℙ𝕡 , ⊤ , (_∧_ , V_))
  , ⊤-top
  , ( (λ x y → ⊓-lower₀ x y , ⊓-lower₁ x y)
    , λ { x y z (z⊑x , z⊑y) → ⊓-greatest x y z z⊑x z⊑y })
  , (⊔-upper , ⊔-least)
  , distr
  where
    ℙ𝕡 = DCPoset (X , P)
    𝔻  = | ℙ𝕡 |ₚ

    -- Function that forget the downwards-closure information.
    |_|𝔻 : 𝔻 → 𝒫 X
```

```
| S , _ |𝔻 = S

⊤ = (λ _ → Unit 𝓁₀ , Unit-prop) , λ _ _ _ _ → tt

∩-down : (S T : 𝒫 X)
       → [ isDownwardsClosed (X , P) S ]
       → [ isDownwardsClosed (X , P) T ]
       → [ isDownwardsClosed (X , P) (S ∩ T) ]
∩-down S T S↓ T↓ x y x∈S∩T y⊑x = S↓ x y (π₀ x∈S∩T) y⊑x , T↓ x y (π₁ x∈S∩T) y⊑x

_∧_ : 𝔻 → 𝔻 → 𝔻
(S , S-dc) ∧ (T , T-dc) = (S ∩ T) , ∩-down S T S-dc T-dc

⊤-top : (D : 𝔻) → [ D ⊑[ 𝔻ₚ ] ⊤ ]
⊤-top D _ _ = tt

-- Given a family U over 𝔻 and some x : X, `in-some-set U x` holds iff there is some
-- set S among U such that x ∈ S.
in-some-set-of : (U : Fam 𝓁₀ 𝔻) → X → Type 𝓁₀
in-some-set-of U x = Σ[ i ∈ index U ] [ x ∈ | U $ i |𝔻 ]

⋁_ : Fam 𝓁₀ 𝔻 → 𝔻
⋁ U = (λ x → ‖ in-some-set-of U x ‖ , ‖‖-prop _) , ⊔U↓
  where
    NTS : (x y : X)
        → [ y ⊑[ (X , P) ] x ] → in-some-set-of U x → ‖ in-some-set-of U y ‖
    NTS x y y⊑x (i , x∈Uᵢ) = | i , π₁ (U $ i) x y x∈Uᵢ y⊑x |

    ⊔U↓ : [ isDownwardsClosed (X , P) (λ x → ‖ in-some-set-of U x ‖ , ‖‖-prop _) ]
    ⊔U↓ x y |p| y⊑x = ‖‖-rec (‖‖-prop _) (NTS x y y⊑x) |p|

open JoinSyntax 𝔻 ⋁_

⊔-upper : (U : Fam 𝓁₀ 𝔻) (D : 𝔻) → D ∈ U → [ D ⊑[ 𝔻ₚ ] (⋁ U) ]
⊔-upper U D D∈S@(i , p) x x∈D = | i , subst (λ V → [ | V |𝔻 x ]) (sym p) x∈D |

⊔-least : (U : Fam 𝓁₀ 𝔻) (z : 𝔻) → [ ∀[ x ∈ U ] (x ⊑[ 𝔻ₚ ] z) ] → [ (⋁ U) ⊑[ 𝔻ₚ ] z ]
⊔-least U D φ x x∈⊔S = ‖‖-rec (∈-prop | D |𝔻) NTS x∈⊔S
  where
    NTS : in-some-set-of U x → [ x ∈ | D |𝔻 ]
    NTS (i , x∈Uᵢ) = φ (U $ i) (i , refl) x x∈Uᵢ

⊓-lower₀ : (U V : 𝔻) → [ (U ∧ V) ⊑[ 𝔻ₚ ] U ]
⊓-lower₀ _ _ _ (x∈U , _) = x∈U

⊓-lower₁ : (U V : 𝔻) → [ (U ∧ V) ⊑[ 𝔻ₚ ] V ]
⊓-lower₁ _ _ _ (_ , x∈V) = x∈V

⊓-greatest : (U V W : 𝔻) → [ W ⊑[ 𝔻ₚ ] U ] → [ W ⊑[ 𝔻ₚ ] V ] → [ W ⊑[ 𝔻ₚ ] (U ∧ V) ]
⊓-greatest U V W W⊑U W⊑V x x∈W = (W⊑U x x∈W) , (W⊑V x x∈W)

distr : (U : 𝔻) (V : Fam 𝓁₀ 𝔻) → U ∧ (⋁ V) ≡ ⋁⟨ i ⟩ (U ∧ (V $ i))
distr U V@(I , _) = ⊑[ 𝔻ₚ ]-antisym _ _ down up
  where
    LHS = | U ∧ (⋁ V) |𝔻
    RHS = | ⋁⟨ i ⟩ (U ∧ (V $ i)) |𝔻
```

```
        down : [ LHS ⊆ RHS ]
        down x (x∈D , x∈⊔U) =
          ∥∥-rec (∥∥-prop _) (λ { (i , x∈Uᵢ) → ∣ i , x∈D , x∈Uᵢ ∣ }) x∈⊔U

        up : [ RHS ⊆ LHS ]
        up x = ∥∥-rec (is-true-prop (x ∈ LHS)) φ
          where
            φ : in-some-set-of [ U ∧ (V $ i) ∣ i ∶ I ] x → [ ∣ U ∣𝔻 x ] × [ ∣ ⋁ V ∣𝔻 x ]
            φ (i , x∈D , x∈Uᵢ) = x∈D , ∣ i , x∈Uᵢ ∣

-- Frames form an SNS.

-- Similar to the poset case, we start by expressing what it means for an equivalence to
-- preserve the structure of a frame
isARawHomoEqv : {ℓ₁ ℓ₂ : Level} (M N : Σ (Type ℓ₀) (RawFrameStr ℓ₁ ℓ₂))
              → π₀ M ≃ π₀ N
              → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂)
isARawHomoEqv {ℓ₂ = ℓ₂} M N e@(f , _) =
  Σ[ f-mono ∈ isMonotonic M-pos N-pos f ]
  Σ[ g-mono ∈ isMonotonic N-pos M-pos g ]
    isRawFrameHomo M N (f , f-mono) × isRawFrameHomo N M (g , g-mono)
  where
    M-pos = pos-of-raw-frame M
    N-pos = pos-of-raw-frame N
    g     = equivFun (invEquiv e)

pos-of : Σ (Type ℓ₀) (RawFrameStr ℓ₁ ℓ₂) → Σ (Type ℓ₀) (Order ℓ₁)
pos-of (A , ((RPS , _) , _)) = (A , RPS)

top-of : (F : Σ (Type ℓ₀) (RawFrameStr ℓ₁ ℓ₂)) → π₀ F
top-of (_ , _ , ⊤ , _) = ⊤

-- Frame univalence

RF-is-SNS : SNS {ℓ₀} (RawFrameStr ℓ₁ ℓ₂) isARawHomoEqv
RF-is-SNS {ℓ₁ = ℓ₁} {ℓ₂ = ℓ₂} {X = A}
          F@(s@(_⊑₀_ , _) , ⊤₀ , _⊓₀_ , V₀)
          G@(t@(_⊑₁_ , _) , ⊤₁ , _⊓₁_ , V₁) =
  isoToEquiv (iso f g sec-f-g ret-f-g)
  where
    C = RawFrameStr ℓ₁ ℓ₂ A

    A-set₀ = carrier-is-set (A , s)

    PS-A = π₀ s
    PS-B = π₀ t

    F-pos = pos-of-raw-frame (A , F)
    G-pos = pos-of-raw-frame (A , G)

    f : isARawHomoEqv (A , F) (A , G) (idEquiv A) → F ≡ G
    f (mono , mono′ , (eq-⊤ , ⊓₀~⊓₁ , V₀~V₁) , k , h) =
      s , ⊤₀ , _⊓₀_ , V₀    ≡⟨ cong (λ - → (s , - , _⊓₀_ , V₀)) eq-⊤ ⟩
      s , ⊤₁ , _⊓₀_ , V₀    ≡⟨ cong {B = λ _ → C} (λ - → s , ⊤₁ , - , V₀) ⊓-eq ⟩
      s , ⊤₁ , _⊓₁_ , V₀    ≡⟨ cong {B = λ _ → C} (λ - → s , _ , _ , -) V-eq ⟩
      s , ⊤₁ , _⊓₁_ , V₁    ≡⟨ cong {B = λ _ → C} (λ - → - , _ , _ , _) eq ⟩
      t , ⊤₁ , _⊓₁_ , V₁    ∎
```

```
      where
        eq : s ≡ t
        eq = ΣProp≡
               (is-true-prop ∘ PosetAx A)
               (funExt₂ λ x y → ⇔toPath (mono x y) (mono' x y))

        ⊓-eq : _⊓₀_ ≡ _⊓₁_
        ⊓-eq = funExt₂ ⊓₀~⊓₁

        V-eq : V₀ ≡ V₁
        V-eq = funExt V₀~V₁

    g : F ≡ G → isARawHomoEqv (A , F) (A , G) (idEquiv A)
    g p = subst (λ - → isARawHomoEqv (A , F) (A , -) (idEquiv A)) p id-iso
      where
        id-iso : isARawHomoEqv (A , F) (A , F) (idEquiv A)
        id-iso = (λ x y x⊑y → x⊑y)
               , (λ x y p → p)
               , (refl , ((λ _ _ → refl) , λ U → refl))
               , refl , (λ _ _ → refl) , λ _ → refl

    sec-f-g : section f g
    sec-f-g p = RawFrameStr-set ℓ₁ ℓ₂ A F G (f (g p)) p

    ret-f-g : retract f g
    ret-f-g a@(mono , mono' , q , r) = ΣProp≡ NTS₀ NTS₁
      where
        NTS₀ : _
        NTS₀ _ = isPropΣ (isMonotonic-prop G-pos F-pos (id A)) λ _ →
                 isPropΣ NTS₀' λ _ → NTS₀''
          where
            NTS₀' : _
            NTS₀' = isRawFrameHomo-prop (A , F) (A , G) (id A , λ x y x⊑y → mono x y x⊑y)
            NTS₀'' : _
            NTS₀'' = isRawFrameHomo-prop (A , G) (A , F) (id A , mono')

        NTS₁ : g (f (mono , mono' , q , r)) .π₀ ≡ mono
        NTS₁ = isMonotonic-prop F-pos G-pos (id A) _ _

-- A predicate expressing that an equivalence between the underlying types of two frames
-- is frame-homomorphic.
isHomoEqv : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → π₀ F ≃ π₀ G → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂)
isHomoEqv {ℓ₁ = ℓ₁} {ℓ₂ = ℓ₂} (A , (s , _)) (B , (t , _)) = isARawHomoEqv (A , s) (B , t)

-- We collect all frame-homomorphic equivalences between two frames in the following type.
_≃f_ : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → Type (ℓ₀ ⊔ ℓ₁ ⊔ suc ℓ₂)
F ≃f G = Σ[ e ∈ ⟨ F ⟩ ≃ ⟨ G ⟩ ] isHomoEqv F G e

isHomoEqv-prop : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (e : ⟨ F ⟩ ≃ ⟨ G ⟩) → isProp (isHomoEqv F G e)
isHomoEqv-prop F G e@(f , _) =
  isPropΣ (isMonotonic-prop (pos F) (pos G) f) λ f-mono →
  isPropΣ (isMonotonic-prop (pos G) (pos F) g) λ g-mono →
  isPropΣ (isRawFrameHomo-prop ⟨ F , F-rs ⟩ ⟨ G , G-rs ⟩ (f , f-mono)) λ _ →
  isPropΣ (carrier-is-set (pos F) _ _) λ _ →
  isPropΣ (λ p q → funExt₂ λ x y → carrier-is-set (pos F) _ _ (p x y) (q x y)) λ _ →
          λ p q → funExt λ U → carrier-is-set (pos F) _ _ (p U) (q U)
  where
```

```
    F-rs : RawFrameStr _ _ | F |F
    F-rs = π₀ (π₁ F)
    G-rs : RawFrameStr _ _ | G |F
    G-rs = π₀ (π₁ G)
    g = equivFun (invEquiv e)

-- Notice that ≃f is equivalent to ≅f.
≃f≃≅f : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (F ≃f G) ≃ (F ≅f G)
≃f≃≅f F G = isoToEquiv (iso to from sec ret)
  where
    to : F ≃f G → F ≅f G
    to (e@(f , _) , (f-mono , g-mono , f-homo , g-homo)) = f₀ , f₀-frame-iso
      where
        g = equivFun (invEquiv e)

        f₀ : F –f→ G
        f₀ = (f , f-mono) , f-homo

        g₀ : G –f→ F
        g₀ = (g , g-mono) , g-homo

        f₀-frame-iso : isFrameIso {F = F} {G} f₀
        f₀-frame-iso = g₀ , Iso.rightInv (equivToIso e) , Iso.leftInv (equivToIso e)

    from : F ≅f G → F ≃f G
    from (((f , f-mono) , f-homo) , ((g , g-mono) , g-homo) , sec , ret) =
      isoToEquiv (iso f g sec ret) , f-mono , g-mono , f-homo , g-homo

    sec : section to from
    sec (f , g , sec , ret) = ΣProp≡ (isFrameIso-prop {F = F} {G = G}) refl

    ret : retract to from
    ret (e , f-homo , g-homo) = ΣProp≡ (isHomoEqv-prop F G) (ΣProp≡ isPropIsEquiv refl)

FrameAx-props : (A : Type ℓ₀) (str : RawFrameStr ℓ₁ ℓ₂ A)
                → isProp [ FrameAx str ]
FrameAx-props A str = is-true-prop (FrameAx str)

frame-is-SNS : SNS {ℓ₀} (FrameStr ℓ₁ ℓ₂) isHomoEqv
frame-is-SNS {ℓ₁ = ℓ₁} {ℓ₂ = ℓ₂} =
  SNS-PathP→SNS-≡
    (FrameStr ℓ₁ ℓ₂)
    isHomoEqv
    (add-axioms-SNS _ FrameAx-props (SNS-≡→SNS-PathP isARawHomoEqv RF-is-SNS))

frame-is-SNS-PathP : SNS-PathP {ℓ₀} (FrameStr ℓ₁ ℓ₂) isHomoEqv
frame-is-SNS-PathP = SNS-≡→SNS-PathP isHomoEqv frame-is-SNS

-- Similar to the poset case, this is sufficient to establish that the category of frames
-- is univalent

≃f≃≡ : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (F ≃f G) ≃ (F ≡ G)
≃f≃≡ = SIP frame-is-SNS-PathP

-- However, there are two minor issues with this.
--
-- 1. We do not have to talk about equivalences as we are talking about sets;
```

```
-- isomorphisms are well-behaved in our case as we are dealing with sets.
--
-- 2. We do not have to require the frame data to be preserved. We can show that any
-- poset isomorphism preserves the frame operators.
--
-- We will therefore strengthen our result to work with the notion of poset isomorphism.

-- We start by showing the equivalence between ≃f and ≅ₚ.

≃f≃≅ₚ : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (pos F ≅ₚ pos G) ≃ (F ≃f G)
≃f≃≅ₚ F G = isoToEquiv (iso from to ret-to-from sec-to-from)
  where
    to : F ≃f G → pos F ≅ₚ pos G
    to (e@(f , _) , f-mono , g-mono , _) =
      (f , f-mono) , (g , g-mono) , retEq e , secEq e
      where
        g = equivFun (invEquiv e)

    from : pos F ≅ₚ pos G → F ≃f G
    from ((f , f-mono) , (g , g-mono) , sec , ret) =
        isoToEquiv (iso f g sec ret)
      , f-mono , g-mono , (resp-⊤ , resp-∧ , resp-∨) , (g-resp-⊤ , g-resp-∧ , g-resp-∨)
      where
        open PosetReasoning (pos G)

        resp-⊤ : f ⊤[ F ] ≡ ⊤[ G ]
        resp-⊤ = top-unique G (f ⊤[ F ]) NTS
          where
            NTS : (x : ∣ G ∣F) → [ x ⊑[ pos G ] (f ⊤[ F ]) ]
            NTS x = x       ⊑⟨ ≡⇒⊑ (pos G) (sym (sec x)) ⟩
                    f (g x) ⊑⟨ f-mono (g x) ⊤[ F ] (⊤[ F ]-top (g x)) ⟩
                    f ⊤[ F ] ■

        g-resp-⊤ : g ⊤[ G ] ≡ ⊤[ F ]
        g-resp-⊤ = g ⊤[ G ] ≡⟨ cong g (sym resp-⊤) ⟩ g (f ⊤[ F ]) ≡⟨ ret ⊤[ F ] ⟩ ⊤[ F ] ∎

        resp-∧ : (x y : ∣ F ∣F) → f (x ⊓[ F ] y) ≡ (f x) ⊓[ G ] (f y)
        resp-∧ x y = ⊓-unique G (f x) (f y) (f (x ⊓[ F ] y)) NTS₀ NTS₁ NTS₂
          where
            NTS₀ : [ f (x ⊓[ F ] y) ⊑[ pos G ] (f x) ]
            NTS₀ = f-mono (x ⊓[ F ] y) x (⊓[ F ]-lower₀ x y)

            NTS₁ : [ f (x ⊓[ F ] y) ⊑[ pos G ] (f y) ]
            NTS₁ = f-mono (x ⊓[ F ] y) y (⊓[ F ]-lower₁ x y)

            NTS₂ : (w : ∣ G ∣F)
                 → [ w ⊑[ pos G ] f x ]
                 → [ w ⊑[ pos G ] f y ]
                 → [ w ⊑[ pos G ] f (x ⊓[ F ] y) ]
            NTS₂ w w⊑fx w⊑fy = w       ⊑⟨ ≡⇒⊑ (pos G) (sym (sec w)) ⟩
                               f (g w) ⊑⟨ f-mono _ _ gw⊑x∧y ⟩
                               f (x ⊓[ F ] y) ■
              where
                gw⊑x : [ g w ⊑[ pos F ] x ]
                gw⊑x = subst (λ - → [ g w ⊑[ pos F ] - ]) (ret x) (g-mono w (f x) w⊑fx)

                gw⊑y : [ g w ⊑[ pos F ] y ]
```

```
        gw⊑y = subst (λ - → [ g w ⊑[ pos F ] - ]) (ret y) (g-mono w (f y) w⊑fy)

        gw⊑x∧y : [ g w ⊑[ pos F ] (x ⊓[ F ] y) ]
        gw⊑x∧y = ⊓[ F ]-greatest x y (g w) gw⊑x gw⊑y

g-resp-∧ : (x y : | G |F) → g (x ⊓[ G ] y) ≡ (g x) ⊓[ F ] (g y)
g-resp-∧ x y =
  g (x ⊓[ G ] y)          ≡⟨ cong (λ - → g (- ⊓[ G ] y)) (sym (sec x)) ⟩
  g (f (g x) ⊓[ G ] y)    ≡⟨ cong (λ - → g (_ ⊓[ G ] -)) (sym (sec y)) ⟩
  g (f (g x) ⊓[ G ] f (g y)) ≡⟨ cong g (sym (resp-∧ (g x) (g y)))      ⟩
  g (f (g x ⊓[ F ] g y)) ≡⟨ ret (g x ⊓[ F ] g y)                       ⟩
  g x ⊓[ F ] g y          ∎

resp-V : (U : Fam _ | F |F) → f (V[ F ] U) ≡ (V[ G ] [ f x | x ε U ])
resp-V U = V-unique G [ f x | x ε U ] (f (V[ F ] U)) NTS₀ NTS₁
  where
    NTS₀ : (x : | G |F) → x ε [ f y | y ε U ] → [ x ⊑[ pos G ] f (V[ F ] U) ]
    NTS₀ x (i , p) = x           ⊑⟨ ≡⇒⊑ (pos G) (sym (sec _)) ⟩
                     f (g x)     ⊑⟨ f-mono _ _ g⊑fVU           ⟩
                     f (g (f (V[ F ] U))) ⊑⟨ ≡⇒⊑ (pos G) (sec _) ⟩
                     f (V[ F ] U) ■
      where
        g⊑fVU : [ g x ⊑[ pos F ] (g (f (V[ F ] U))) ]
        g⊑fVU =
          subst
            (λ - → [ rel (pos F) (g x) - ])
            (sym (ret (V[ F ] U)))
            (V[ F ]-upper U (g x) (subst (λ - → g - ε U) p (i , (sym (ret _)))))

    NTS₁ : (w : | G |F)
         → ((o : | G |F) → o ε [ f x | x ε U ] → [ o ⊑[ pos G ] w ])
         → [ f (V[ F ] U) ⊑[ pos G ] w ]
    NTS₁ w h = f (V[ F ] U) ⊑⟨ fVU⊑fgw ⟩ f (g w) ⊑⟨ ≡⇒⊑ (pos G) (sec _) ⟩ w ■
      where
        gfVU⊑gw : [ g (f (V[ F ] U)) ⊑[ pos F ] g w ]
        gfVU⊑gw = subst
                    (λ - → [ - ⊑[ pos F ] g w ])
                    (sym (ret _))
                    (V[ F ]-least U (g w) NTS′)
          where
            NTS′ : [ ∀[ u ε U ] (u ⊑[ pos F ] (g w)) ]
            NTS′ u (i , p) =
              subst (λ - → [ - ⊑[ pos F ] (g w) ]) p
                (subst
                  (λ - → [ - ⊑[ pos F ] g w ])
                  (ret _)
                  (g-mono _ _ (h (f (π₁ U i)) (i , refl))))

        fVU⊑fgw : [ f (V[ F ] U) ⊑[ pos G ] f (g w) ]
        fVU⊑fgw = f-mono _ _ (subst (λ - → [ - ⊑[ pos F ] g w ]) (ret _) gfVU⊑gw)

g-resp-V : (U : Fam _ | G |F) → g (V[ G ] U) ≡ V[ F ] [ g x | x ε U ]
g-resp-V U =
  g (V[ G ] U)                ≡⟨ cong (λ - → g (V[ G ] (π₀ U , -))) NTS ⟩
  g (V[ G ] [ f (g x) | x ε U ]) ≡⟨ cong g (sym (resp-V [ g x | x ε U ])) ⟩
  g (f (V[ F ] [ g x | x ε U ])) ≡⟨ ret (V[ F ] [ g x | x ε U ])          ⟩
```

```
          V[ F ] [ g x | x ∈ U ]       ∎
          where
            NTS : π₁ U ≡ f ∘ g ∘ π₁ U
            NTS = funExt λ x → sym (sec (π₁ U x))

    sec-to-from : section to from
    sec-to-from is@((f , f-mono) , ((g , g-mono) , sec , ret)) =
      ΣProp≡
        (isPosetIso-prop (pos F) (pos G))
        (forget-mono (pos F) (pos G) (f , f-mono) (π₀ (to (from is))) refl)

    ret-to-from : retract to from
    ret-to-from (eqv , eqv-homo) =
      ΣProp≡ (isHomoEqv-prop F G ) (ΣProp≡ isPropIsEquiv refl)

-- Now that we have this result, we can move on to show that given two frames F and G,
-- (pos F) ≅ₚ (pos G) is equivalent to F ≡ G.

≅ₚ≃≡ : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (pos F ≅ₚ pos G) ≃ (F ≡ G)
≅ₚ≃≡ F G = pos F ≅ₚ pos G ≃⟨ ≃f≃≅ₚ F G ⟩ F ≃f G ≃⟨ ≃f≃≡ F G ⟩ F ≡ G ∎

≅ₚ≃≅f : (F G : Frame ℓ₀ ℓ₁ ℓ₂) → (pos F ≅ₚ pos G) ≃ (F ≅f G)
≅ₚ≃≅f F G = pos F ≅ₚ pos G ≃⟨ ≃f≃≅ₚ F G ⟩ F ≃f G ≃⟨ ≃f≃≅f F G ⟩ F ≅f G ∎
```

A.5 The Nucleus module

```
{-# OPTIONS --cubical --safe #-}

module Nucleus where

open import Basis
open import Poset
open import Frame

-- A predicate expressing whether a function is a nucleus.
isNuclear : (L : Frame ℓ₀ ℓ₁ ℓ₂) → (| L |F → | L |F) → Type (ℓ₀ ⊔ ℓ₁)
isNuclear L j = N₀ × N₁ × N₂
  where
    N₀ = (x y : | L |F) → j (x ⊓[ L ] y) ≡ (j x) ⊓[ L ] (j y)
    N₁ = (x   : | L |F) → [ x ⊑[ pos L ] (j x) ]
    N₂ = (x   : | L |F) → [ j (j x) ⊑[ pos L ] j x ]

-- The type of nuclei.
Nucleus : Frame ℓ₀ ℓ₁ ℓ₂ → Type (ℓ₀ ⊔ ℓ₁)
Nucleus L = Σ (| L |F → | L |F) (isNuclear L)

-- The top element is fixed point for every nucleus.
nuclei-resp-⊤ : (L : Frame ℓ₀ ℓ₁ ℓ₂) ((j , _) : Nucleus L) → j ⊤[ L ] ≡ ⊤[ L ]
nuclei-resp-⊤ L (j , N₀ , N₁ , N₂) = ⊑[ pos L ]-antisym _ _ (⊤[ L ]-top _) (N₁ _)

-- Every nucleus is idempotent.
idem : (L : Frame ℓ₀ ℓ₁ ℓ₂) → ((j , _) : Nucleus L) → (x : | L |F) → j (j x) ≡ j x
idem L (j , N₀ , N₁ , N₂) x = ⊑[ pos L ]-antisym _ _ (N₂ x) (N₁ (j x))
```

```
-- Every nucleus is monotonic.
mono : (L : Frame ℓ₀ ℓ₁ ℓ₂) ((j , _) : Nucleus L)
     → (x y : ∣ L ∣F) → [ x ⊑[ pos L ] y ] → [ (j x) ⊑[ pos L ] (j y) ]
mono L (j , N₀ , N₁ , N₂) x y x⊑y =
  j x           ⊑⟨ ≡⇒⊑ (pos L) (cong j (x⊑y⇒x=x∧y L x⊑y)) ⟩
  j (x ⊓[ L ] y) ⊑⟨ ≡⇒⊑ (pos L) (N₀ x y)      ⟩
  j x ⊓[ L ] j y ⊑⟨ ⊓[ L ]-lower₁ (j x) (j y) ⟩
  j y           ■
  where
    open PosetReasoning (pos L)

-- The set of fixed points for nucleus `j` is equivalent hence equal to its image.
-- This is essentially due to the fact that j (j ())
nuclear-image : (L : Frame ℓ₀ ℓ₁ ℓ₂)
              → let ∣L∣ = ∣ L ∣F in (j : ∣L∣ → ∣L∣)
              → isNuclear L j
              → (Σ[ b ∈ ∣L∣ ] ∥ Σ[ a ∈ ∣L∣ ] (b ≡ j a) ∥) ≡ (Σ[ a ∈ ∣L∣ ] (j a ≡ a))
nuclear-image L j N@(n₀ , n₁ , n₂) = isoToPath (iso f g sec-f-g ret-f-g)
  where
    A-set = carrier-is-set (pos L)

    f : (Σ[ b ∈ ∣ L ∣F ] ∥ Σ[ a ∈ ∣ L ∣F ] (b ≡ j a) ∥) → Σ[ a ∈ ∣ L ∣F ] (j a ≡ a)
    f (b , img) = b , ∥∥-rec (A-set (j b) b) ind img
      where
        ind : Σ[ a ∈ ∣ L ∣F ](b ≡ j a) → j b ≡ b
        ind (a , img) =
          j b   ≡⟨ cong j img ⟩
          j (j a) ≡⟨ idem L (j , N) a ⟩
          j a   ≡⟨ sym img ⟩
          b     ∎

    g : (Σ[ a ∈ ∣ L ∣F ] (j a ≡ a)) → (Σ[ b ∈ ∣ L ∣F ] ∥ Σ[ a ∈ ∣ L ∣F ] (b ≡ j a) ∥)
    g (a , a-fix) = a , ∣ a , (sym a-fix) ∣

    sec-f-g : section f g
    sec-f-g (x , jx=x) = ΣProp≡ (λ y → A-set (j y) y) refl

    ret-f-g : retract f g
    ret-f-g (x , p) = ΣProp≡ (λ y → ∥∥-prop (Σ[ a ∈ ∣ L ∣F ] y ≡ j a)) refl

-- The set of fixed points for a nucleus `j` forms a poset.
fix-pos : (L : Frame ℓ₀ ℓ₁ ℓ₂) → (N : Nucleus L) → Poset ℓ₀ ℓ₁
fix-pos {ℓ₀ = ℓ₀} {ℓ₁} L (j , N₀ , N₁ , N₂) =
  𝔽 , _≤_ , 𝔽-set , ≤-refl , ≤-trans , ≤-antisym
  where
    P = pos L
    A-set = carrier-is-set (pos L)

    𝔽 : Type ℓ₀
    𝔽 = Σ[ a ∈ ∣ L ∣F ] j a ≡ a

    𝔽-set : isSet 𝔽
    𝔽-set = isSetΣ A-set (λ a → isProp→isSet (A-set (j a) a))

    _≤_ : 𝔽 → 𝔽 → hProp ℓ₁
    (a , _) ≤ (b , _) = [ a ⊑[ P ] b ] , is-true-prop (a ⊑[ P ] b)
```

```
    ≤-refl : [ isReflexive _≤_ ]
    ≤-refl (x , _) = ⊑[ P ]-refl x

    ≤-trans : [ isTransitive _≤_ ]
    ≤-trans (x , _) (y , _) (z , _) x≤y y≤x = ⊑[ P ]-trans x y z x≤y y≤x

    ≤-antisym : [ isAntisym ℙ-set _≤_ ]
    ≤-antisym (x , _) (y , _) x≤y y≤x =
      ΣProp≡ (λ z → A-set (j z) z) (⊑[ P ]-antisym x y x≤y y≤x)

-- The set of fixed points of a nucleus `j` forms a frame.
-- The join of this frame is define as ⨆ᵢ Uᵢ := j (⨆'ᵢ Uᵢ) where ⨆' denotes the join of L.
fix : (L : Frame ℓ₀ ℓ₁ ℓ₂) → (N : Nucleus L) → Frame ℓ₀ ℓ₁ ℓ₂
fix {ℓ₁ = ℓ₁} {ℓ₂ = ℓ₂} L N@(j , N₀ , N₁ , N₂) =
                          | fix-pos L N |ₚ
  , (strₚ (fix-pos L N) , (⊤[ L ] , nuclei-resp-⊤ L N) , _∧_ , ⋁_)
  , top
  , ( (λ x y → ⊓-lower₀ x y , ⊓-lower₁ x y)
    , λ { x y z (z⊑x , x⊑y) → ⊓-greatest x y z z⊑x x⊑y })
  , ((⨆-upper , ⨆-least) , distr)
  where
    A = π₀ (fix-pos L N)

    _⊏_ : | pos L |ₚ → | pos L |ₚ → hProp ℓ₁
    _⊏_       = λ x y → x ⊑[ pos L ] y

    _⊏N_ : A → A → hProp ℓ₁
    _⊏N_  = λ x y → x ⊑[ fix-pos L N ] y

    ⋁L_ : Fam ℓ₂ | L |F → | L |F
    ⋁L x = ⋁[ L ] x

    ⊑N-antisym = ⊑[ fix-pos L N ]-antisym
    A-set      = carrier-is-set (fix-pos L N)

    open PosetReasoning (pos L)

    _∧_ : A → A → A
    _∧_ (x , x-f) (y , y-f) =
      x ⊓[ L ] y , NTS
      where
        NTS : j (x ⊓[ L ] y) ≡ x ⊓[ L ] y
        NTS = j (x ⊓[ L ] y) ≡⟨ N₀ x y ⟩
              j x ⊓[ L ] j y ≡⟨ cong (λ - → - ⊓[ L ] _) x-f ⟩
                x ⊓[ L ] j y ≡⟨ cong (λ - → _ ⊓[ L ] -) y-f ⟩
                x ⊓[ L ] y   ∎

    ⋁_ : Fam ℓ₂ A → A
    ⋁ (I , F) = j (⋁[ L ] 𝒢) , j⨆L-fixed
      where
        𝒢 = I , π₀ ∘ F
        j⨆L-fixed : j (j (⋁[ L ] 𝒢)) ≡ j (⋁[ L ] 𝒢)
        j⨆L-fixed = ⊑[ pos L ]-antisym _ _ (N₂ (⋁[ L ] 𝒢)) (N₁ (j (⋁[ L ] 𝒢)))

    open JoinSyntax A ⋁_

    top : (o : A) → [ o ⊏N (⊤[ L ] , nuclei-resp-⊤ L N) ]
```

```
top = ⊤[ L ]-top ∘ π₀

⊓-lower₀ : (o p : ℋ) → [ (o ∧ p) ⊑N o ]
⊓-lower₀ (o , _) (p , _) = ⊓[ L ]-lower₀ o p

⊓-lower₁ : (o p : ℋ) → [ (o ∧ p) ⊑N p ]
⊓-lower₁ (o , _) (p , _) = ⊓[ L ]-lower₁ o p

⊓-greatest : (o p q : ℋ) → [ q ⊑N o ] → [ q ⊑N p ] → [ q ⊑N (o ∧ p) ]
⊓-greatest (o , _) (p , _) (q , _) q⊑o q⊑p = ⊓[ L ]-greatest o p q q⊑o q⊑p

⊔-least : (U : Fam ℓ₂ ℋ) (u : ℋ) → [ ∀[ x ∈ U ] (x ⊑N u) ] → [ (⋁ U) ⊑N u ]
⊔-least U (u , fix) U⊑u = subst (λ - → [ j (⋁[ L ] 𝒢) ⊑ - ]) fix NTS₀
  where
    𝒢 : Fam ℓ₂ ∣ pos L ∣ₚ
    𝒢 = π₀ <$> U

    𝒢⊑u : [ ∀[ o ∈ 𝒢 ] (o ⊑ u) ]
    𝒢⊑u o (i , eq′) = o  ⊑⟨ ≡⇒⊑ (pos L) (sym eq′) ⟩
                         𝒢 $ i ⊑⟨ U⊑u (𝒢 $ i , π₁ (U $ i)) (i , refl) ⟩
                         u ■

    NTS₀ : [ j (⋁[ L ] 𝒢) ⊑ j u ]
    NTS₀ = mono L N (⋁[ L ] 𝒢) u (⋁[ L ]-least 𝒢 u 𝒢⊑u)

⊔-upper : (U : Fam ℓ₂ ℋ) (x : ℋ) → x ∈ U → [ x ⊑N (⋁ U) ]
⊔-upper U (x , _) o∈U@(i , eq) =
  x                  ⊑⟨ NTS ⟩
  ⋁[ L ] (π₀ <$> U) ⊑⟨ N₁ (⋁[ L ] (π₀ <$> U)) ⟩
  j (⋁[ L ] (π₀ <$> U)) ■
  where
    NTS : [ x ⊑ (⋁[ L ] (π₀ <$> U)) ]
    NTS = ⋁[ L ]-upper (π₀ <$> U) x (i , λ j → π₀ (eq j))

distr : (x : Σ[ x ∈ ∣ L ∣F ] j x ≡ x) (U@(I , _) : Fam ℓ₂ ℋ)
      → x ∧ (⋁ U) ≡ ⋁⟨ i ⟩ (x ∧ (U $ i))
distr x@(x , jx=x) U@(I , F) = ΣProp≡ (λ x → carrier-is-set (pos L) (j x) x) NTS
  where
    -- U is a family of inhabitants of ∣ L ∣F paired with proofs that they are fixed
    -- points for j. U₀ is the family obtained by discarding the proofs
    U₀ : Fam ℓ₂ ∣ L ∣F
    U₀ = ⁅ π₀ x ∣ x ∈ U ⁆

    x=jx = sym jx=x

    NTS : π₀ (x ∧ (⋁ U)) ≡ π₀ (⋁⟨ i ⟩ (x ∧ (U $ i)))
    NTS =
      π₀ (x ∧ (⋁ U))                ≡⟨ refl                  ⟩
      x ⊓[ L ] j (⋁L U₀)            ≡⟨ cong (λ - → - ⊓[ L ] j (⋁L U₀)) x=jx ⟩
      j x ⊓[ L ] j (⋁L U₀)          ≡⟨ sym (N₀ x (⋁[ L ] U₀)) ⟩
      j (x ⊓[ L ] (⋁L U₀))          ≡⟨ cong j (dist L x U₀)  ⟩
      j (⋁L ⁅ x ⊓[ L ] yᵢ ∣ yᵢ ∈ U₀ ⁆) ≡⟨ refl                ⟩
      π₀ (⋁⟨ i ⟩ (x ∧ (U $ i)))     ∎
```

A.6 The `Cover` module

```
-- # Some lemmas about the cover relation

{-# OPTIONS --cubical --safe #-}

module Cover where

open import Level
open import FormalTopology
open import Poset
open import Basis

-- We define a submodule `CoverFromFormalTopology` parameterised by a formal topology `ℱ`
-- since all of the functions in this module take as argument a certain formal topology.

module CoverFromFormalTopology (ℱ : FormalTopology ℓ ℓ') where

-- We refer to the underlying poset of the formal topology `ℱ` as `P`, and its outcome
-- function as `out`.

  private
    P   = pos ℱ
    out = outcome
```

Definition of the covering relation

```
-- The covering relation is defined as follows:

  data _◁_ (a : | P |₀) (U : | P |₀ → hProp ℓ) : Type ℓ where
    dir : [ U a ] → a ◁ U
    branch : (b : exp ℱ a) → (f : (c : out ℱ b) → next ℱ c ◁ U) → a ◁ U
    squash : (p q : a ◁ U) → p ≡ q

  ◁-prop : (a : | P |₀) (U : 𝒫 | P |₀) → isProp (a ◁ U)
  ◁-prop a U = squash
```

Lemmas about the covering relation

```
-- We now prove four crucial lemmas about the cover.

-- Lemma 1

  module _ {U : | P |₀ → hProp ℓ} (U-down : [ isDownwardsClosed P U ]) where

    ◁-lem₁ : {a a' : | P |₀} → [ a' ⊑[ P ] a ] → a ◁ U → a' ◁ U
    ◁-lem₁ {_} {_} h (squash p₀ p₁ i) = squash (◁-lem₁ h p₀) (◁-lem₁ h p₁) i
```

```
    ◁-lem₁ {_} {_} h (dir q)           = dir (U-down _ _ q h)
    ◁-lem₁ {a = a} {a'} h (branch b f) = branch b' g
      where
        b' : exp ℱ a'
        b' = π₀ (sim ℱ a' a h b)

        g : (c' : out ℱ b') → next ℱ c' ◁ U
        g c' = ◁-lem₁ δc'⊑δc (f c)
          where
            c : out ℱ b
            c = π₀ (π₁ (sim ℱ a' a h b) c')

            δc'⊑δc : [ next ℱ c' ⊑[ P ] next ℱ c ]
            δc'⊑δc = π₁ (π₁ (sim ℱ a' a h b) c')

-- Lemma 2

  module _ (U : 𝒫 ∣ P ∣ₚ) (V : 𝒫 ∣ P ∣ₚ) (V-dc : [ isDownwardsClosed P V ]) where

    ◁-lem₂ : {a : ∣ P ∣ₚ} → a ◁ U → [ a ∈ V ] → a ◁ (U ∩ V)
    ◁-lem₂ (squash p₀ p₁ i) h = squash (◁-lem₂ p₀ h) (◁-lem₂ p₁ h) i
    ◁-lem₂ (dir q)      h = dir (q , h)
    ◁-lem₂ (branch b f) h = branch b λ c → ◁-lem₂ (f c) (V-dc _ _ h (mono ℱ _ b c))

-- Lemma 3

  module _ (U : 𝒫 ∣ P ∣ₚ) (V : 𝒫 ∣ P ∣ₚ)
           (U-dc : [ isDownwardsClosed P U ])
           (V-dc : [ isDownwardsClosed P V ]) where

    ◁-lem₃ : {a a' : ∣ P ∣ₚ} → [ a' ⊑[ P ] a ] → a ◁ U → a' ◁ V → a' ◁ (V ∩ U)
    ◁-lem₃ {a} {a'} a'⊑a (squash p q i) r = squash (◁-lem₃ a'⊑a p r) (◁-lem₃ a'⊑a q r) i
    ◁-lem₃ {a} {a'} a'⊑a (dir a∈U)    r = ◁-lem₂ V U U-dc r (U-dc a a' a∈U a'⊑a)
    ◁-lem₃ {a} {a'} a'⊑a (branch b f) r = branch b' g
      where
        b' : exp ℱ a'
        b' = π₀ (sim ℱ a' a a'⊑a b)

        g : (c' : out ℱ b') → next ℱ c' ◁ (V ∩ U)
        g c' = ◁-lem₃ NTS (f c) (◁-lem₁ V-dc (mono ℱ a' b' c') r)
          where
            c : out ℱ b
            c = π₀ (π₁ (sim ℱ a' a a'⊑a b) c')

            NTS : [ next ℱ c' ⊑[ P ] next ℱ c ]
            NTS = π₁ (π₁ (sim ℱ a' a a'⊑a b) c')

-- Lemma 4

  ◁-lem₄ : (U : 𝒫 ∣ P ∣ₚ) (V : 𝒫 ∣ P ∣ₚ)
    → ((u : ∣ P ∣ₚ) → [ u ∈ U ] → u ◁ V) → (a : ∣ P ∣ₚ) → a ◁ U → a ◁ V
  ◁-lem₄ U V h a (squash p₀ p₁ i) = squash (◁-lem₄ U V h a p₀) (◁-lem₄ U V h a p₁) i
  ◁-lem₄ U V h a (dir p)      = h a p
  ◁-lem₄ U V h a (branch b f) = branch b λ c → ◁-lem₄ U V h (next ℱ c) (f c)
```

A.7 The CoverFormsNucleus module

```
{-# OPTIONS --cubical --safe #-}

module CoverFormsNucleus where

open import Basis hiding (A) renaming (squash to squash')
open import Poset
open import Frame
open import Cover
open import Nucleus        using    (isNuclear; Nucleus; fix; idem)
open import Cubical.Data.Bool using (Bool; true; false)
open import FormalTopology renaming (pos to pos')

-- We use a module that takes some formal topology `F` as a parameter to reduce
-- parameter-passing.

module NucleusFrom (ℱ : FormalTopology 𝓁₀ 𝓁₀) where

-- We refer to the underlying poset of `F` as `P` and the frame of downwards-closed subsets
-- of `P` as `P↓`.

  private
    P       = pos' ℱ
    P↓      = DCFrame P
    _⊏_     = λ (x y : stage ℱ) → x ⊑[ P ] y

  open CoverFromFormalTopology ℱ public

-- Now, we define the *covering nucleus* which we denote by `j`. At its heart, this is
-- nothing but the map `U ↦ - ◁ U`.

  j : ∣ P↓ ∣F → ∣ P↓ ∣F
  j (U , U-down) = (λ - → U ▷ -) , U▷-dc
    where
      -- This is not propositional unless we force it to be using the HIT definition!
      _▷_ : 𝒫 ∣ P ∣ₚ → 𝒫 ∣ P ∣ₚ
      U ▷ a = a ◁ U , squash

      U▷-dc : [ isDownwardsClosed P (λ - → (- ◁ U) , squash) ]
      U▷-dc a a₀ aϵU₁ a₀⊑a = ◁-lem₁ U-down a₀⊑a aϵU₁

  j-nuclear : isNuclear P↓ j
  j-nuclear = N₀ , N₁ , N₂
    where
      -- We reason by antisymmetry and prove in (d) j (a₀ ⊓ a₁) ⊑ (j a₀) ⊓ (j a₁) and
      -- in (u) (j a₀) ⊓ (j a₁) ⊑ j (a₀ ⊓ a₁).
      N₀ : (U 𝒰 : ∣ P↓ ∣F) → j (U ⊓[ P↓ ] 𝒰) ≡ (j U) ⊓[ P↓ ] (j 𝒰)
      N₀ 𝒰@(U , U-down) 𝒱@(V , V-down) =
        ⊑[ pos P↓ ]-antisym (j (𝒰 ⊓[ P↓ ] 𝒱)) (j 𝒰 ⊓[ P↓ ] j 𝒱) down up
        where
          down : [ (j (𝒰 ⊓[ P↓ ] 𝒱)) ⊑[ pos P↓ ] (j 𝒰 ⊓[ P↓ ] j 𝒱) ]
          down a (dir (a∈U , a∈V)) = dir a∈U , dir a∈V
          down a (branch b f) = branch b (π₀ ∘ IH) , branch b (π₁ ∘ IH)
            where
```

```
            IH : (c : outcome 𝓕 b) → [ π₀ (j 𝕌 ⊓[ P↓ ] j 𝕍) (next 𝓕 c) ]
            IH c = down (next 𝓕 c) (f c)
          down a (squash p q i) = squash (π₀ IH₀) (π₀ IH₁) i , squash (π₁ IH₀) (π₁ IH₁) i
            where
              _ : a ◁ π₀ (glb-of P↓ (U , U-down) (V , V-down))
              _ = p
              IH₀ = down a p
              IH₁ = down a q

          up : [ (j 𝕌 ⊓[ P↓ ] j 𝕍) ⊑[ pos P↓ ] j (𝕌 ⊓[ P↓ ] 𝕍) ]
          up a (a◁U , a◁V) = ◁-lem₃ V U V-down U-down (⊑[ P ]-refl a) a◁V a◁U

        N₁ : (𝕌 : | P↓ |F) → [ 𝕌 ⊑[ pos P↓ ] (j 𝕌) ]
        N₁ _ a₀ a∈U = dir a∈U

        N₂ : (𝕌 : | P↓ |F) → [ π₀ (j (j 𝕌)) ⊆ π₀ (j 𝕌) ]
        N₂ 𝕌@(U , _) = ◁-lem₄ (π₀ (j 𝕌)) U (λ _ q → q)

-- We denote by `L` the frame of fixed points for `j`.

  L : Frame (suc ℓ₀) ℓ₀ ℓ₀
  L = fix P↓ (j , j-nuclear)

-- The following is a just a piece of convenient notation for projecting out the underlying
-- set of a downwards-closed subset equipped with the information that it is a fixed point
-- for `j`.

  ⟪_⟫ : | L |F → 𝒫 | P |ₚ
  ⟪ ((U , _) , _) ⟫ = U

-- Given some `x` in `F`, we define a map taking `x` to its *downwards-closure*.

  ↓-clos : stage 𝓕 → | P↓ |F
  ↓-clos x = x↓ , down-DC
    where
      x↓ = λ y → y ⊑[ P ] x
      down-DC : [ isDownwardsClosed P x↓ ]
      down-DC z y z⊑x y⊑z = ⊑[ P ]-trans y z x y⊑z z⊑x

  x◁x↓ : (x : stage 𝓕) → x ◁ (λ - → - ⊑[ P ] x)
  x◁x↓ x = dir (⊑[ P ]-refl x)

-- By composing this with the covering nucleus, we define a map `e` from `F` to `P↓`.

  e : | P |ₚ → | P↓ |F
  e z = (λ a → (a ◁ (π₀ (↓-clos z))) , squash) , NTS
    where
      NTS : [ isDownwardsClosed P (λ a → (a ◁ (λ - → - ⊑[ P ] z)) , squash) ]
      NTS _ _ x y = ◁-lem₁ (λ _ _ x⊑y y⊑z → ⊑[ P ]-trans _ _ z y⊑z x⊑y) y x

-- We can further refine the codomain of `e` to `L`. In other words, we can prove that `j (e
-- x) = e x` for every `x`. We call the version `e` with the refined codomain `η`.

  fixing : (x : | P |ₚ) → j (e x) ≡ e x
  fixing x = ⊑[ pos P↓ ]-antisym (j (e x)) (e x) down up
    where
      down : ∀ y → [ π₀ (j (e x)) y ] → [ π₀ (e x) y ]
```

```
      down = ◃-lem₄ (π₀ (e x)) (π₀ (↓-clos x)) (λ _ q → q)

      up : [ e x ⊑[ pos P↓ ] j (e x) ]
      up = π₀ (π₁ j-nuclear) (e x)

  η : stage ℱ → | L |F
  η x = (e x) , (fixing x)

-- Furthermore, `η` is a monotonic map.

  ηm : P –m→ pos L
  ηm = η , η-mono
    where
      η-mono : isMonotonic P (pos L) η
      η-mono x y x⊑y = ◃-lem₄ (π₀ (↓-clos x)) (π₀ (↓-clos y)) NTS
        where
          NTS : (u : | P |ₚ) → [ u ∈ π₀ (↓-clos x) ] → u ◃ π₀ (↓-clos y)
          NTS _ p = ◃-lem₁ (π₁ (↓-clos y)) p (dir x⊑y)
```

A.8 The FormalTopology module

```
{-# OPTIONS --cubical --safe #-}

module FormalTopology where

open import Basis
open import Poset

InteractionStr : (A : Type ℓ) → Type (suc ℓ)
InteractionStr {ℓ = ℓ} A =
  Σ[ B ∈ (A → Type ℓ) ] Σ[ C ∈ ({x : A} → B x → Type ℓ) ]({x : A} → {y : B x} → C y → A)

InteractionSys : (ℓ : Level) → Type (suc ℓ)
InteractionSys ℓ = Σ (Type ℓ) InteractionStr

state    : InteractionSys ℓ → Type ℓ
action   : (D : InteractionSys ℓ) → state D → Type ℓ
reaction : (D : InteractionSys ℓ) → {x : state D} → action D x → Type ℓ
δ        : (D : InteractionSys ℓ) {x : state D} {y : action D x}
         → reaction D y → state D

state  (A , _ , _ , _) = A
action (_ , B , _ , _) = B
reaction (_ , _ , C , _) = C
δ      (_ , _ , _ , d) = d

hasMono : (P : Poset ℓ₀ ℓ₁) → InteractionStr | P |ₚ → Type (ℓ₀ ⊔ ℓ₁)
hasMono P i =
  (a : state IS) (b : action IS a) (c : reaction IS b) → [ δ IS c ⊑[ P ] a ]
  where
    IS : InteractionSys _
    IS = | P |ₚ , i
```

```
module _ (P : Poset ℓ₀ ℓ₁) (𝒥-str : InteractionStr | P |ₚ) where
  𝒥 : InteractionSys ℓ₀
  𝒥 = (| P |ₚ , 𝒥-str)

  hasSimulation : Type (ℓ₀ ⊔ ℓ₁)
  hasSimulation =
    (a' a : | P |ₚ) → [ a' ⊑[ P ] a ] →
      (b : action 𝒥 a) → Σ[ b' ∈ action 𝒥 a' ]
        ((c' : reaction 𝒥 b') → Σ[ c ∈ reaction 𝒥 b ] [ δ 𝒥 c' ⊑[ P ] δ 𝒥 c ])

FormalTopology : (ℓ₀ ℓ₁ : Level) → Type (suc ℓ₀ ⊔ suc ℓ₁)
FormalTopology ℓ₀ ℓ₁ =
  Σ[ P ∈ Poset ℓ₀ ℓ₁ ] Σ[ 𝒥 ∈ InteractionStr | P |ₚ ] hasMono P 𝒥 × hasSimulation P 𝒥

pos : FormalTopology ℓ₀ ℓ₁ → Poset ℓ₀ ℓ₁
pos (P , _) = P

IS : (ℱ : FormalTopology ℓ₀ ℓ₁) → InteractionStr | pos ℱ |ₚ
IS (_ , is , _) = is

stage : FormalTopology ℓ₀ ℓ₁ → Type ℓ₀
stage (P , 𝒥-str , _) = state (| P |ₚ , 𝒥-str)

exp : (ℱ : FormalTopology ℓ₀ ℓ₁) → stage ℱ → Type ℓ₀
exp (P , 𝒥-str , _) = action (| P |ₚ , 𝒥-str)

outcome : (ℱ : FormalTopology ℓ₀ ℓ₁) → {a : stage ℱ} → exp ℱ a → Type ℓ₀
outcome (P , 𝒥-str , _) = reaction (| P |ₚ , 𝒥-str)

next : (ℱ : FormalTopology ℓ₀ ℓ₁) {a : stage ℱ} {b : exp ℱ a} → outcome ℱ b → stage ℱ
next (P , 𝒥-str , _) = δ (| P |ₚ , 𝒥-str)

mono : (ℱ : FormalTopology ℓ₀ ℓ₁) → hasMono (pos ℱ) (IS ℱ)
mono (_ , _ , φ , _) = φ

sim : (ℱ : FormalTopology ℓ₀ ℓ₁) → hasSimulation (pos ℱ) (IS ℱ)
sim (_ , _ , _ , φ) = φ
```

A.9 The UniversalProperty module

```
{-# OPTIONS --cubical --safe #-}

module UniversalProperty where

open import Basis
open import Frame
open import Poset
open import FormalTopology renaming (pos to pos')
open import CoverFormsNucleus

compr : {X : Type ℓ₀} {Y : Type ℓ₁} → (g : X → Y) → 𝒫 X → Fam ℓ₀ Y
compr g U = (index ❴ U ❵) , g ∘ (_$_ ❴ U ❵)
```

```
syntax compr (λ x → e) 𝓕 = [ e | x ∈ 𝓕 ]

module _ (F : FormalTopology ℓ₀ ℓ₀) where

  P   = pos′ F
  F↓  = DCFrame P
  P↓  = pos F↓
  _⊏_ = λ (x y : stage F) → x ⊑[ P ] y

  open NucleusFrom F
```

Representation

```
  represents : (R : Frame (suc ℓ₀) ℓ₀ ℓ₀) → (P ─m→ pos R) → Type ℓ₀
  represents R (f , _) =
    (a : | P |ₚ) (b : exp F a) →
      [ f a ⊑[ pos R ] ⋁[ R ] [ f (next F c) | c ∶ outcome F b ] ]

-- By the way, note that the converse is always true.

  represents⁻¹ : (R : Frame (suc ℓ₀) ℓ₀ ℓ₀) → (m : P ─m→ pos R)
                 → Type ℓ₀
  represents⁻¹ R (f , _) =
    (a : | P |ₚ) (b : exp F a) →
      [ (⋁[ R ] [ f (next F c) | c ∶ outcome F b ]) ⊑[ pos R ] (f a) ]

  conv : (R : Frame (suc ℓ₀) ℓ₀ ℓ₀) (f : P ─m→ pos R) → represents⁻¹ R f
  conv R (f , f-mono) a b =
    ⋁[ R ]-least ([ f (next F c) | c ∶ outcome F b ]) (f a) NTS
    where
      NTS : [ ⋁[ a′ ε [ f (next F c) | c ∶ outcome F b ] ] (a′ ⊑[ pos R ] f a) ]
      NTS a′ (i , eq) = subst (λ - → [ rel (pos R) - (f a) ]) eq NTS′
        where
          NTS′ : [ f (next F i) ⊑[ pos R ] f a ]
          NTS′ = f-mono (next F i) a (mono F a b i)
```

Flatness

```
_↓_↓ : | P |ₚ → | P |ₚ → 𝒫 | P |ₚ
_↓_↓ a b = λ - → - ⊑[ P ] a ∩ - ⊑[ P ] b

isFlat : (F : Frame (suc ℓ₀) ℓ₀ ℓ₀) → (m : P ─m→ pos F) → Type (suc ℓ₀)
isFlat F (f , _) = (⊤[ F ] ≡ ⋁[ F ] [ f a | a ∶ | P |ₚ ])
                 × ((a b : | P |ₚ) → f a ∩[ F ] f b ≡ ⋁[ F ] (f ⟨$⟩ ⟪ a ↓ b ↓ ⟫))
```

The universal property

```
-- Statement.

  universal-prop : Type (suc (suc ℓ₀))
  universal-prop =
    (R : Frame (suc ℓ₀) ℓ₀ ℓ₀) (f : P –m→ pos R) → isFlat R f → represents R f →
      isContr (Σ[ g ∈ (L –f→ R) ] (_∘m_ {P = P} {Q = pos L} {R = pos R} (π₀ g) ηm) ≡ f)

-- Before the proof we will need some lemmas.

  cover+ : {x y : | P |ₚ} ((U , _) : | F↓ |F) → [ x ∈ ❨ η y ❩ ] → [ y ∈ U ] → x ◁ U
  cover+ (_ , U-dc) x∈ηy y∈U = ◁-lem₄ _ _ (λ z z⊑y → dir (U-dc _ z y∈U z⊑y)) _ x∈ηy

  main-lemma : (U : | L |F) → U ≡ ⋁[ L ] ｛ η u | u ∈ ❨ U ❩ ｝
  main-lemma U@((U , U-dc) , U-fix) = ⊑[ pos L ]-antisym _ _ down up
    where
      down : [ U ⊑[ pos L ] (⋁[ L ] ｛ η x | x ∈ U ｝) ]
      down x x∈U = dir | (x , x∈U) , dir (⊑[ P ]-refl x) |

      up : [ (⋁[ L ] ｛ η x | x ∈ U ｝) ⊑[ pos L ] U ]
      up x (dir x∈V) = ∥∥-rec (is-true-prop (U x)) NTS x∈V
        where
          NTS : Σ[ y ∈ _ ] [ x ∈ ❨ η (π₀ y) ❩ ] → [ x ∈ U ]
          NTS ((y , y∈U) , x◁y↓) =
            subst (λ V → [ π₀ V x ]) U-fix (cover+ (U , U-dc) x◁y↓ y∈U)
      up x (branch b f) = subst (λ V → [ π₀ V x ]) U-fix (branch b (dir ∘ IH))
        where
          IH : (c : outcome F b) → [ next F c ∈ U ]
          IH c = up (next F c) (f c)
      up x (squash x◁V₀ x◁V₁ i) = is-true-prop (U x) (up x x◁V₀) (up x x◁V₁) i

-- Proof.

  module MainProof (R      : Frame (suc ℓ₀) ℓ₀ ℓ₀)
                   (fm     : P –m→ pos R)
                   (f-flat : isFlat R fm)
                   (rep    : represents R fm) where
    f      = _$ₘ_ fm
    f-mono = π₁ fm

    _⊑R_ : | R |F → | R |F → hProp ℓ₀
    x ⊑R y = x ⊑[ pos R ] y

    infix 9 _⊑R_

    g : | L |F → | R |F
    g U = ⋁[ R ] ｛ f u | u ∈ ❨ U ❩ ｝

    g-mono : isMonotonic (pos L) (pos R) g
    g-mono ((U , _) , _) ((V , _) , _) U⊑V =
      ⋁[ R ]-least _ _ (λ o o∈fU → ⋁[ R ]-upper _ _ (NTS o o∈fU ))
      where
        NTS : (x : | R |F) → x ∈ (∃ U , f ∘ π₀) → x ∈ (∃ V , f ∘ π₀)
        NTS x ((x' , x'∈fU) , fU∋i=x) = (x' , U⊑V x' x'∈fU) , fU∋i=x
```

```
    gm : pos L ─m→ pos R
    gm = g , g-mono

-- `g` respects the top element

    g-resp-⊤ : g ⊤[ L ] ≡ ⊤[ R ]
    g-resp-⊤ = g ⊤[ L ]                          ≡⟨ refl           ⟩
               ⋁[ R ] (f ⟨$⟩ (∃ ⟪ ⊤[ L ] ⟫ , π₀)) ≡⟨ family-iff R NTS ⟩
               ⋁[ R ] (| P |ₚ , f)               ≡⟨ sym (π₀ f-flat) ⟩
               ⊤[ R ]                            ∎
      where
        NTS : (x : | R |F)
            → (x ε (f ⟨$⟩ (∃ ⟪ ⊤[ L ] ⟫ , π₀)) → x ε (| P |ₚ , f))
            × (x ε (| P |ₚ , f) → x ε (f ⟨$⟩ (∃ ⟪ ⊤[ L ] ⟫ , π₀)))
        NTS x = (λ { ((y , _) , eq) → y , eq }) , (λ { (y , eq) → (y , tt) , eq })

-- `g` respects the binary meets

    g-resp-⊓ : (U D : | L |F) → g (U ⊓[ L ] D) ≡ g U ⊓[ R ] g D
    g-resp-⊓ U D =
      g (U ⊓[ L ] D)
        ≡⟨ refl ⟩
      ⋁[ R ] ｛ f a ∣ a ∈ ⟪ U ⊓[ L ] D ⟫ ｝
        ≡⟨ I ⟩
      ⋁[ R ] ｛ ⋁[ R ] (f ⟨$⟩ ⟪ u ↓ v ↓ ⟫) ∣ ((u , _) , (v , _)) : (∃ ⟪ U ⟫ × ∃ ⟪ D ⟫) ｝
        ≡⟨ cong (λ - → (⋁[ R ] ((∃ ⟪ U ⟫ × ∃ ⟪ D ⟫) , -))) II ⟩
      ⋁[ R ] (((∃ ⟪ U ⟫) × (∃ ⟪ D ⟫)) , λ { ((u , _) , (v , _)) → f u ⊓[ R ] f v })
        ≡⟨ sym (sym-distr R (｛ f u ∣ u ∈ ⟪ U ⟫ ｝) (｛ f v ∣ v ∈ ⟪ D ⟫ ｝)) ⟩
      (⋁[ R ] ｛ f u ∣ u ∈ ⟪ U ⟫ ｝) ⊓[ R ] (⋁[ R ] ｛ f v ∣ v ∈ ⟪ D ⟫ ｝)
        ≡⟨ refl ⟩
      g U ⊓[ R ] g D
        ∎
      where
        II : (λ { ((u , _) , (v , _)) → ⋁[ R ] (f ⟨$⟩ ⟪ u ↓ v ↓ ⟫) })
           ≡ (λ { ((u , _) , (v , _)) → (f u) ⊓[ R ] (f v) })
        II = sym (funExt λ { ((u , _) , (v , _)) → π₁ f-flat u v })
        I  : _
        I  = ⊑[ pos R ]-antisym _ _ down up
          where
            LHS = ⋁[ R ] ｛ f a ∣ a ∈ ⟪ U ⊓[ L ] D ⟫ ｝
            RHS =
              ⋁[ R ] ｛ ⋁[ R ] (f ⟨$⟩ ⟪ u ↓ v ↓ ⟫)
                      ∣ ((u , _) , (v , _)) : (∃ ⟪ U ⟫ × ∃ ⟪ D ⟫) ｝

            down : [ LHS ⊑[ pos R ] RHS ]
            down = ⋁[ R ]-least _ _ isUB
              where
                isUB : _
                isUB o ((a , (a∈U , a∈V)) , eq) =
                  ⋁[ R ]-upper _ _ NTS
                  where
                    u : ∃ ⟪ U ⟫
                    u = a , a∈U

                    v : ∃ ⟪ D ⟫
```

```
                  v = a , a∈V

                  NTS : o ε [ V[ R ] (f <$> ⟪ u ↓ v ↓ ⟫)
                            | ((u , _) , (v , _)) : (∃ ⟪ U ⟫ × ∃ ⟪ D ⟫) ]
                  NTS = (u , v) , subst (λ o′ → _ ≡ o′) eq NTS′
                    where
                      down′ : [ V[ R ] (f <$> ⟪ a ↓ a ↓ ⟫) ⊑[ pos R ] f a ]
                      down′ =
                        V[ R ]-least _ _ λ { z ((_ , (k , _)) , eq′) →
                          subst (λ - → [ - ⊑[ pos R ] _ ]) eq′ (f-mono _ _ k) }
                      up′ : [ f a ⊑[ pos R ] V[ R ] (f <$> ⟪ a ↓ a ↓ ⟫) ]
                      up′ =
                        V[ R ]-upper _ _ ((a , ⊑[ P ]-refl a , ⊑[ P ]-refl a) , refl)

                      NTS′ : V[ R ] (f <$> ⟪ a ↓ a ↓ ⟫) ≡ f a
                      NTS′ = ⊑[ pos R ]-antisym _ _ down′ up′

            up : [ LHS ⊒[ pos R ] RHS ]
            up = V[ R ]-least _ _ isUB
              where
                isUB : _
                isUB o (i@((x , xεD) , (y , yεU)) , eq) =
                  subst (λ o′ → [ o′ ⊑[ pos R ] _ ]) eq (V[ R ]-least _ _ NTS)
                  where
                    NTS : _
                    NTS w (j@(z , (z⊑x , z⊑y)) , eq′) = V[ R ]-upper _ _ ((z , φ) , eq′)
                      where
                        φ : [ z ∈ (⟪ U ⟫ ∩ ⟪ D ⟫) ]
                        φ = (π₁ (π₀ U) x z xεD z⊑x) , (π₁ (π₀ D) y z yεU z⊑y)

-- `g` respects the joins

    open PosetReasoning (pos R)

    resp-V-lem : (U@(I , _) : Fam ℓ₀ | L |F)
               → V[ R ] [ f a | a ∈ ⟪ V[ L ] U ⟫ ]
               ≡ (V[ R ] [ f a | (_ , a , _) : (Σ[ i ∈ I ] ∃ ⟪ U $ i ⟫) ])
    resp-V-lem U@(I , _) = ⊑[ pos R ]-antisym _ _ down up
      where
        LHS = V[ R ] [ f a | a ∈ ⟪ V[ L ] U ⟫ ]
        RHS = V[ R ] [ f a | (_ , a , _) : (Σ[ i ∈ I ] ∃ ⟪ U $ i ⟫) ]

        ϑ : (x : | P |ₚ) → [ x ∈ ⟪ V[ L ] U ⟫ ] → [ f x ⊑R RHS ]
        ϑ x (squash p q i) = is-true-prop (f x ⊑R _) (ϑ x p) (ϑ x q) i
        ϑ x (dir     xεVU) = ∥∥-rec (is-true-prop (f x ⊑R _)) NTS xεVU
                             where
                               NTS : _
                               NTS (j , cov) = V[ R ]-upper _ _ ((j , x , cov) , refl)
        ϑ x (branch b h)   =
          f x                    ⊑⟨ rep x b ⟩
          V[ R ] (_ , f ∘ next F) ⊑⟨ V[ R ]-least _ _ NTS ⟩
          RHS                    ■
          where
            NTS : (r : | R |F) → r ε (_ , f ∘ next F) → [ r ⊑R _ ]
            NTS r (c , p) = subst (λ - → [ - ⊑R _ ]) p (ϑ (next F c) (h c))
```

```
      down : [ LHS ⊑R RHS ]
      down =
        ⋁[ R ]-least _ _ λ r ((x , cov) , p) → subst (λ - → [ - ⊑R _ ]) p (ϑ x cov)

      up : [ LHS ⊒[ pos R ] RHS ]
      up = ⋁[ R ]-least _ _ NTS
        where
          NTS : _
          NTS r ((i , (x , xεU)) , p) = ⋁[ R ]-upper _ _ ((x , dir | i , xεU |) , p)

  g-resp-⊔ : (U : Fam ℓ₀ | L |F) → g (⋁[ L ] U) ≡ ⋁[ R ] (g ⟨$⟩ U)
  g-resp-⊔ U@(I , h) =
    ⋁[ R ] ❴ f a | a ∈ ❨ ⋁[ L ] U ❩ ❵
      ≡⟨ resp-⋁-lem U ⟩
    ⋁[ R ] ❴ f a | (_ , (a , _)) ∶ Σ[ i ∈ I ] Σ[ x ∈ | P |ₚ ] [ x ∈ ❨ U $ i ❩ ] ❵
      ≡⟨ flatten R I (λ i → Σ[ x ∈ | P |ₚ ] [ x ∈ ❨ U $ i ❩ ]) (λ { _ (a , _) → f a }) ⟩
    ⋁[ R ] ❴ ⋁[ R ] ❴ f a | a ∈ ❨ U $ i ❩ ❵ | i ∶ I ❵
      ≡⟨ refl ⟩
    ⋁[ R ] ❴ g (U $ i) | i ∶ I ❵
      ∎

-- `g` is a frame homomorphism

  g-frame-homo : isFrameHomomorphism L R gm
  g-frame-homo = g-resp-𝟏 , (g-resp-⊓ , g-resp-⊔)

-- `g` makes the diagram commute

  lem : (a a' : | P |ₚ) → a' ◃ π₀ (↓-clos a) → [ f a' ⊑[ pos R ] f a ]
  lem a a' (squash p q i) = is-true-prop (f a' ⊑[ pos R ] f a) (lem _ _ p) (lem _ _ q) i
  lem a a' (dir      a'⊑a)  = f-mono a' a a'⊑a
  lem a a' (branch b h)     =
    f a' ⊑⟨ rep a' b                                  ⟩
    ⋁[ R ] (outcome F b , f ∘ next F) ⊑⟨ ⋁[ R ]-least _ _ isUB ⟩
    f a  ■
    where
      isUB : ∀ a₀ → a₀ ε (outcome F b , f ∘ next F) → [ a₀ ⊑[ pos R ] f a ]
      isUB a₀ (c , p) = a₀    ⊑⟨ ≡⇒⊑ (pos R) (sym p) ⟩
                        f (next F c) ⊑⟨ lem a (next F c) (h c) ⟩
                        f a ■

  gm∘ηm = _∘m_ {P = P} {Q = pos L} {R = pos R} gm ηm

  gm∘ηm~f : (x : | P |ₚ) → gm $ₘ (ηm $ₘ x) ≡ fm $ₘ x
  gm∘ηm~f x = ⊑[ pos R ]-antisym _ _ down (⋁[ R ]-upper _ _ ((x , x◃x↓ x) , refl))
    where
      down : [ (⋁[ R ] (∃ π₀ (e x) , f ∘ π₀)) ⊑[ pos R ] f x ]
      down = ⋁[ R ]-least _ _ λ { o ((y , φ) , eq) → subst (λ _ → _) eq (lem x y φ) }

  g∘η=f : gm∘ηm ≡ fm
  g∘η=f = forget-mono P (pos R) gm∘ηm fm (funExt gm∘ηm~f)

  g∘η=f' : g ∘ η ≡ f
  g∘η=f' = subst (λ { (h , _) → h ≡ f }) (sym g∘η=f) refl

-- `g` is uniquely determined
```

```
    g-unique : (y : Σ[ g' ∈ (L -f→ R) ]
                    (_∘m_ {P = P} {Q = pos L} {R = pos R} (π₀ g') ηm ≡ fm))
             → ((gm , g-frame-homo) , g∘η=f) ≡ y
    g-unique ((g'm , g'-frame-homo) , φ) = ΣProp≡ I II
      where
        g' = _$ₘ_ g'm

        f=g'∘η : f ≡ g' ∘ η
        f=g'∘η = subst (λ { (f' , _) → f' ≡ g' ∘ η }) φ refl

        NTS₀ : (y : Σ (| pos L |ₚ → | pos R |ₚ) (isMonotonic (pos L) (pos R)))
             → isProp ((_∘m_ {P = P} {Q = pos L} {R = pos R} y ηm) ≡ fm)
        NTS₀ y = isSetΣ
                   (isSetΠ λ _ → carrier-is-set (pos R))
                   (λ h → isProp→isSet (isMonotonic-prop P (pos R) h))
                   (_∘m_ {P = P} {Q = pos L} {R = pos R} y ηm) fm

        I : (h : L -f→ R) → isProp (_∘m_ {P = P} {Q = pos L} {R = pos R} (π₀ h) ηm ≡ fm)
        I h = isSetΣ
                (isSetΠ λ _ → carrier-is-set (pos R))
                (λ h → isProp→isSet (isMonotonic-prop P (pos R) h))
                (_∘m_ {P = P} {Q = pos L} {R = pos R} (π₀ h) ηm) fm

        g~g' : (U : | L |F) → g U ≡ g' U
        g~g' U =
          g U                                ≡⟨ refl ⟩
          ⋁[ R ] ⁅ f u      | u ∈ ❨ U ❩ ⁆ ≡⟨ eq₀ ⟩
          ⋁[ R ] ⁅ g'(η u) | u ∈ ❨ U ❩ ⁆ ≡⟨ eq₁ ⟩
          g' (⋁[ L ] ⁅ η u | u ∈ ❨ U ❩ ⁆) ≡⟨ cong g' (sym (main-lemma U)) ⟩
          g' U ∎
          where
            eq₀ : ⋁[ R ] (f ⟨$⟩ ⟪ ❨ U ❩ ⟫) ≡ ⋁[ R ] ((g' ∘ η) ⟨$⟩ ⟪ ❨ U ❩ ⟫)
            eq₀ = cong (λ - → ⋁[ R ] (- ⟨$⟩ ⟪ ❨ U ❩ ⟫)) f=g'∘η
            eq₁ : ⋁[ R ] ((g' ∘ η) ⟨$⟩ ⟪ ❨ U ❩ ⟫) ≡ g' (⋁[ L ] (η ⟨$⟩ ⟪ ❨ U ❩ ⟫))
            eq₁ = sym (π₁ (π₁ g'-frame-homo) (η ⟨$⟩ ⟪ ❨ U ❩ ⟫))

        II : (gm , g-frame-homo) ≡ (g'm , g'-frame-homo)
        II = ΣProp≡
               (isFrameHomomorphism-prop L R)
               (ΣProp≡ (isMonotonic-prop (pos L) (pos R)) (funExt g~g'))

-- The final proof

  main : universal-prop
  main R fm@(f , f-mono) f-flat rep =
    (((g , g-mono) , g-resp-𝟙 , g-resp-⊓ , g-resp-⊔) , g∘η=f) , g-unique
    where
      open MainProof R fm f-flat rep
```

A.10 The CantorSpace module

```
{-# OPTIONS --cubical --safe #-}
```

```
module CantorSpace where

open import Basis                    hiding (A; B)
open import Cubical.Data.Empty.Base using (⊥; rec)
open import Cubical.Data.Bool.Base   using (true; false; _≟_) renaming (Bool to 𝔹)
open import Cubical.Data.List        using (List; _∷_; []) renaming (_++_ to _^_)
open import Cover
open import Poset
open import FormalTopology

-- We open the `SnocList` module with the type `𝔹` of booleans.

open import SnocList 𝔹 _≟_ renaming (SnocList to ℂ; SnocList-set to ℂ-set)

-- The empty list and the snoc operator are called `[]` and `∷` respectively. Concatenation
-- operation on snoc lists is called `_++_`. Note that concatenation on lists is therefore
-- renamed to `_^_` to prevent conflict.
```

The Cantor poset

```
-- `xs` is less than `ys` if there is some `zs` such that `xs = ys ++ zs`.

_≤_ : ℂ → ℂ → hProp zero
xs ≤ ys = (Σ[ zs ∈ ℂ ] xs ≡ ys ++ zs) , prop
  where
    prop : isProp (Σ[ zs ∈ ℂ ] xs ≡ ys ++ zs)
    prop (_ , p) (_ , q) = ΣProp≡ (λ ws → ℂ-set xs (ys ++ ws)) (++-lemma p q)

-- As `_≤_` is a partial order, we package it up as a poset.

ℂ-pos : Poset zero zero
ℂ-pos = ℂ , _≤_ , ℂ-set , ≤-refl , ≤-trans , ≤-antisym
  where
    ≤-refl : (xs : ℂ) → [ xs ≤ xs ]
    ≤-refl xs = [] , refl

    ≤-trans : (xs ys zs : ℂ) → [ xs ≤ ys ] → [ ys ≤ zs ] → [ xs ≤ zs ]
    ≤-trans xs ys zs (as , p) (bs , q) =
      (bs ++ as) , NTS
      where
        NTS : xs ≡ zs ++ (bs ++ as)
        NTS = xs       ≡⟨ p                              ⟩
              ys ++ as ≡⟨ cong (λ - → - ++ as) q ⟩
              (zs ++ bs) ++ as ≡⟨ sym (assoc zs bs as) ⟩
              zs ++ (bs ++ as) ∎

    ≤-antisym : (xs ys : ℂ) → [ xs ≤ ys ] → [ ys ≤ xs ] → xs ≡ ys
    ≤-antisym xs ys ([] , p) ([]     , q) = p
    ≤-antisym xs ys ([] , p) (bs ∷ x , q) = p
    ≤-antisym xs ys (as ∷ x , p) ([]     , q) = sym q
    ≤-antisym xs ys (as ∷ a , p) (bs ∷ b , q) =
      rec (lemma3 NTS)
```

```
    where
      NTS : xs ≡ xs ++ ((bs ∷ʳ b) ++ (as ∷ʳ a))
      NTS = xs           ≡⟨ p                          ⟩
            ys ++ (as ∷ʳ a) ≡⟨ cong (λ - → - ++ as ∷ʳ a) q ⟩
            (xs ++ (bs ∷ʳ b)) ++ (as ∷ʳ a) ≡⟨ sym (assoc xs (bs ∷ʳ b) (as ∷ʳ a)) ⟩
            xs ++ ((bs ∷ʳ b) ++ (as ∷ʳ a)) ∎
```

The Cantor formal topology

```
-- We give the formal topology of the Cantor space as an
-- [interaction system](http://www.dcs.ed.ac.uk/home/pgh/interactive_systems.html).

-- 1. Each inhabitant of `ℂ` is like a stage of information.

-- 1. At each stage of information we can perform a trivial experiment: querying the next
-- bit.
ℂ-exp = λ (_ : ℂ) → Unit zero

-- 1. Outcome of the trivial experiment is the delivery of the new bit.
ℂ-out = λ (_ : Unit zero) → 𝔹

-- 1. This takes us to a new stage information, obtained by snoc'ing in the new bit to the
-- current stage of information.
ℂ-rev : {_ : ℂ} → 𝔹 → ℂ
ℂ-rev {xs} b = xs ∷ʳ b

-- These four components together form an interaction system that satiesfies the monotonicity
-- and simulation properties (given in `ℂ-mono` and `ℂ-sim`).

ℂ-IS : InteractionStr ℂ
ℂ-IS = ℂ-exp , ℂ-out , λ {xs} → ℂ-rev {xs}

ℂ-mono : hasMono ℂ-pos ℂ-IS
ℂ-mono _ _ c = [] ∷ʳ c , refl

ℂ-sim : hasSimulation ℂ-pos ℂ-IS
ℂ-sim xs ys xs≤ys@([] , p) tt = tt , NTS
  where
    NTS : (b₁ : 𝔹) → Σ[ b₀ ∈ 𝔹 ] [ (xs ∷ʳ b₁) ≤ (ys ∷ʳ b₀) ]
    NTS b₁ = b₁ , subst (λ - → [ (xs ∷ʳ b₁) ≤ (- ∷ʳ b₁) ]) p (⊑[ ℂ-pos ]-refl _)
ℂ-sim xs ys xs≤ys@(zs ∷ʳ z , p) tt = tt , NTS
  where
    NTS : (c₀ : 𝔹) → Σ[ c ∈ 𝔹 ] [ (xs ∷ʳ c₀) ≤ (ys ∷ʳ c) ]
    NTS c₀ =
      head (zs ∷ʳ z) tt , subst (λ - → [ (- ∷ʳ c₀) ≤ _ ]) (sym p) NTS′
      where
        φ   = cong (λ - → ys ++ (- ∷ʳ c₀)) (sym (hd-tl-lemma (zs ∷ʳ z) tt))
        ψ   = cong (λ - → - ∷ʳ c₀) (sym (snoc-lemma ys _ _))
        rem = (ys ++ zs) ∷ʳ z ∷ʳ c₀ ≡⟨ φ ⟩
              (ys ++ (([] ∷ʳ head (zs ∷ʳ z) tt) ++ (tail (zs ∷ʳ z) tt))) ∷ʳ c₀ ≡⟨ ψ ⟩
              ((ys ∷ʳ head (zs ∷ʳ z) tt) ++ tail (zs ∷ʳ z) tt) ∷ʳ c₀ ∎
        NTS′ : [ ((ys ++ zs) ∷ʳ z ∷ʳ c₀) ≤ (ys ∷ʳ head (zs ∷ʳ z) tt) ]
```

```
        NTS′ = ((tail (zs ⌢ z) tt) ⌢ c₀) , rem

-- We finally package up all this as a formal topology

cantor : FormalTopology zero zero
cantor = 𝕔-pos , 𝕔-IS , 𝕔-mono , 𝕔-sim

-- from which we get a covering relation

open CoverFromFormalTopology cantor renaming (_◁_ to _<𝕔|_)

_ : 𝕔 → (𝕔 → hProp zero) → Type zero
_ = _<𝕔|_
```

Statement of compactness

```
-- The statement of compactness then is as follows.

module _ (F : FormalTopology ℓ₀ ℓ₀) where

  open CoverFromFormalTopology F using (_◁_)

  private
    A = stage   F
    B = exp     F
    C = outcome F
    d = next    F

  down : List A → 𝒫 A
  down []       = λ _ → bot ℓ₀
  down (x ∷ xs) = λ y → ∥ [ y ⊑[ pos F ] x ] ∔ [ y ∈ down xs ] ∥ , ∥∥-prop _

  isCompact : Type (suc ℓ₀)
  isCompact = (a : A) (U : 𝒫 A) (U-dc : [ isDownwardsClosed (pos F) U ]) →
              a ◁ U → ∥ Σ[ as ∈ List A ] (a ◁ down as) × [ down as ⊆ U ] ∥
```

The Cantor formal topology is compact

```
-- We now want to view a list of `𝕔`s as a _finite cover_. We associate with some
-- `xss : List 𝕔` a subset, being covered by which corresponds to being covered by this list.

𝕔-down : List 𝕔 → 𝒫 𝕔
𝕔-down = down cantor

syntax 𝕔-down xss xs = xs ↓ xss

-- This subset is downwards-closed.

↓-dc : (xss : List 𝕔) → [ isDownwardsClosed 𝕔-pos (λ - → - ↓ xss) ]
```

```
↓-dc (xs ∷ xss) ys zs ys◁xs∷xss zs≤ys =
  ∥∥-rec (is-true-prop (zs ↓ (xs ∷ xss))) NTS ys◁xs∷xss
  where
    open PosetReasoning ℂ-pos using (_⊑⟨_⟩_; _■)

    NTS : [ ys ≤ xs ] ∔ [ ys ↓ xss ] → [ zs ↓ (xs ∷ xss) ]
    NTS (inl ys≤xs) = ∣ inl (zs ⊑⟨ zs≤ys ⟩ ys ⊑⟨ ys≤xs ⟩ xs ■) ∣
    NTS (inr ys◁xss) = ∣ inr (↓-dc xss ys zs ys◁xss zs≤ys) ∣

-- We claim that the Cantor space is compact.

compact : isCompact cantor

-- Two little lemmas

U⊆V→◁U⊆◁V : (xs : ℂ) (U : 𝓟 ℂ) (V : 𝓟 ℂ) → [ U ⊆ V ] → xs <ℂ| U → xs <ℂ| V
U⊆V→◁U⊆◁V xs U V U⊆V = ◁-lem₄ U V NTS xs
  where
    NTS : (u : ℂ) → [ u ∈ U ] → u <ℂ| V
    NTS u u∈U = dir (U⊆V u u∈U)

↓-++-left : (xss yss : List ℂ) → [ (λ - → - ↓ xss) ⊆ (λ - → - ↓ (xss ^ yss)) ]
↓-++-left []        yss _ ()
↓-++-left (xs ∷ xss) yss ys ys∈down-xs-xss =
  ∥∥-rec (is-true-prop (ys ↓ ((xs ∷ xss) ^ yss))) NTS ys∈down-xs-xss
  where
    NTS : [ ys ≤ xs ] ∔ [ ys ↓ xss ] → [ ys ↓ (xs ∷ xss ^ yss) ]
    NTS (inl ys≤xs) = ∣ inl ys≤xs ∣
    NTS (inr ys∈down-xss) = ∣ inr (↓-++-left xss yss ys ys∈down-xss) ∣

↓-++-right : (xss yss : List ℂ) → [ (λ - → - ↓ yss) ⊆ (λ - → - ↓ (xss ^ yss)) ]
↓-++-right xss [] _ ()
↓-++-right []  (ys ∷ yss) zs zs∈◁ys∷yss = zs∈◁ys∷yss
↓-++-right (xs ∷ xss) (ys ∷ yss) zs zs∈◁ys∷yss =
  ∥∥-rec (is-true-prop (zs ↓ (xs ∷ xss ^ ys ∷ yss))) NTS zs∈◁ys∷yss
  where
    NTS : [ zs ≤ ys ] ∔ [ zs ↓ yss ] → [ zs ↓ (xs ∷ xss ^ ys ∷ yss) ]
    NTS (inl zs≤ys) = let IH = ↓-++-right xss _ _ ∣ inl (⊑[ ℂ-pos ]-refl ys) ∣
                      in ∣ inr (↓-dc (xss ^ ys ∷ yss) ys zs IH zs≤ys) ∣
    NTS (inr zs◁yss) = ∣ inr (↓-++-right xss _ zs ∣ inr zs◁yss ∣) ∣

◁^-decide : (xs : ℂ) (yss zss : List ℂ)
          → [ xs ↓ (yss ^ zss) ]
          → ∥ [ xs ↓ yss ] ∔ [ xs ↓ zss ] ∥
◁^-decide xs []         zss k = ∣ inr k ∣
◁^-decide xs (ys ∷ yss) zss k = ∥∥-rec (∥∥-prop _) NTS₀ k
  where
    NTS₀ : [ xs ≤ ys ] ∔ [ xs ↓ (yss ^ zss) ] → ∥ [ xs ↓ (ys ∷ yss) ] ∔ [ xs ↓ zss ] ∥
    NTS₀ (inl xs≤ys) = ∣ inl ∣ inl xs≤ys ∣ ∣
    NTS₀ (inr xs◁yss^zss) = ∥∥-rec (∥∥-prop _) NTS₁ (◁^-decide xs yss zss xs◁yss^zss)
      where
        NTS₁ : [ xs ↓ yss ] ∔ [ xs ↓ zss ] → ∥ [ xs ↓ (ys ∷ yss) ] ∔ [ xs ↓ zss ] ∥
        NTS₁ (inl xs◁yss) = ∣ inl ∣ inr xs◁yss ∣ ∣
        NTS₁ (inr xs◁zss) = ∣ inr xs◁zss ∣

-- The proof
```

```
-- The proof is by induction on the proof of `xs ◁ U`.

compact xs U U-dc (dir xs∈U) = ∣ xs ∷ [] , NTS₀ , NTS₁ ∣
  where
    NTS₀ : xs <ℂ| (λ - → - ↓ (xs ∷ []))
    NTS₀ = dir ∣ inl (⊑[ ℂ-pos ]-refl xs) ∣

    NTS₁ : [ (λ - → - ↓ (xs ∷ [])) ⊆ U ]
    NTS₁ ys ∣ys◁[xs]∣ = ∥∥-rec (is-true-prop (ys ∈ U)) NTS₁′ ∣ys◁[xs]∣
      where
        NTS₁′ : [ ys ≤ xs ] ⊎ [ ys ↓ [] ] → [ U ys ]
        NTS₁′ (inl ys≤xs) = U-dc xs ys xs∈U ys≤xs

compact xs U U-dc (branch tt f) =
  let
    IH₀ : ∥ Σ[ yss₀ ∈ List ℂ ]
              ((xs ⌢ true) <ℂ| (λ - → - ↓ yss₀)) × [ ℂ-down yss₀ ⊆ U ] ∥
    IH₀ = compact (xs ⌢ true) U U-dc (f true)
    IH₁ : ∥ Σ[ yss ∈ List ℂ ]
              ((xs ⌢ false) <ℂ| (λ - → - ↓ yss) × [ ℂ-down yss ⊆ U ]) ∥
    IH₁ = compact (xs ⌢ false) U U-dc (f false)
  in
    ∥∥-rec (∥∥-prop _) (λ φ → ∥∥-rec (∥∥-prop _) (λ ψ → ∣ NTS φ ψ ∣) IH₁) IH₀
  where
    NTS : Σ[ yss₀ ∈ _ ] ((xs ⌢ true) <ℂ| λ - → - ↓ yss₀) × [ ℂ-down yss₀ ⊆ U ]
        → Σ[ yss₁ ∈ _ ] ((xs ⌢ false) <ℂ| λ - → - ↓ yss₁) × [ ℂ-down yss₁ ⊆ U ]
        → Σ[ yss ∈ _ ] (xs <ℂ| λ - → - ↓ yss) × [ ℂ-down yss ⊆ U ]
    NTS (yss , φ , p) (zss , ψ , q) = yss ^ zss , branch tt g , NTS′
      where
        g : (c : 𝔹) → (xs ⌢ c) <ℂ| (λ - → ℂ-down (yss ^ zss) -)
        g false = U⊆V⇒◁U⊆◁V _ (ℂ-down zss) (ℂ-down (yss ^ zss)) (↓-++-right yss zss) ψ
        g true = U⊆V⇒◁U⊆◁V _ (ℂ-down yss) (ℂ-down (yss ^ zss)) (↓-++-left yss zss) φ

        NTS′ : [ (λ - → - ↓ (yss ^ zss)) ⊆ U ]
        NTS′ ys ys◁yss₀^yss₁ =
          ∥∥-rec (is-true-prop (ys ∈ U)) NTS₂ (◁^-decide _ yss _ ys◁yss₀^yss₁)
          where
            NTS₂ : [ ys ↓ yss ] ⊎ [ ys ↓ zss ] → [ ys ∈ U ]
            NTS₂ (inl ys◁yss₀) = p ys ys◁yss₀
            NTS₂ (inr ys◁yss₁) = q ys ys◁yss₁

compact xs U U-dc (squash xs◁U₀ xs◁U₁ i) =
  squash (compact xs U U-dc xs◁U₀) (compact xs U U-dc xs◁U₁) i
```