Foundations of math and probabilistic theories in Lean

Davood Haji Taghi Tehrani

A thesis submitted for the degree of

Master Physics

December 2025

Department of Physics
The University Of Cologne

Supervisor: Prof. David Gross, Prof. Frank Vallentin

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 161 Abs. 1 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

[illegible]

Davood Haji Taghi Tehrani

Köln, den 17.12.2025

Abstract

Lean is a proof assistant that verifies the correctness of mathematical proofs and formalized logical arguments through a trusted small kernel. This thesis focuses on the formalization of tensor products of partially ordered spaces and of infinite tensor products in this framework. Its aim is to provide precise definitions of measurement devices, probabilities, and physical states in a general setting while assuming only the necessary conditions. The objective is to make all implicit assumptions explicit, and possibly make them leaner.

Contents

List of Figures

Introduction

In probability theory, de Finetti's theorem states the joint probability distribution of an infinite sequence of exchangeable random variables can be expressed as a mixture of distributions in which the variables are independent and identically distributed. This theorem has important applications in quantum cryptography, for example, it allows security analyses to reduce complex correlated attacks to mixtures of independent ones, which can be easier to analyze.

One way to formalize the theorem is through unit ordered vector spaces. The initial aim of this project was to develop such a formalization of de Finetti's theorem in a computer-based proof assistant. It is plausible that, in the near future, a substantial portion of mathematical and theoretical proofs will take place within such systems, analogous to the historical transition in numerical analysis from manual computation to computer-based calculation. Formal verification in such systems further enables the direct use of established results in computational procedures, providing guarantees regarding their correctness.

For this formalization, we chose Lean, an open-source programming language that verifies the correctness of proofs through a small trusted kernel. The largest mathematics library for Lean, Mathlib4, already contains a substantial portion of mathematical results. However, to the best of this author's knowledge, infinite ordered vector spaces have not yet been formalized. Not only that, but also certain fundamental results that are required for these spaces were also not available. Therefore, the focus of the project shifted from the initial goal to formalizing the mathematical basis required for the formulation of such theorems.

In particular, we began by establishing several elementary results, including specific isomorphisms between different tensor product spaces. These results were then used to construct ordered vector spaces and to prove initial theorems about their structure. Our work is mostly based on *Some Applications of Tensor Products of Partially-Ordered Linear Spaces* from A. Hulanicki and R. R. Phelps [1].

Outline

We begin with a brief explanation of Lean as a proof assistant 1.1. Next, we introduce infinite tensor products, also called quasi finite tensors, and provide a simple example of their practical use 1.2.1. The next section examines ordered vector spaces and their role in modeling measurement devices 1.3. It also explains why specific properties are required.

After the preliminaries, the implementation section explains in depth how key definitions are formalized in Lean and presents several important lemmas. It starts from isomorphisms on tensor products. They are required for the definition of infinite tensor products and for several lemmas concerning ordered vector spaces 2.1. The implementation of infinite tensor products in Lean is then explained in detail 2.2. Finally, we describe the formalization of ordered vector spaces 2.3.

Lastly, a brief conclusion is presented 3. The section Miscellaneous outlines the difficulties encountered during the work. This includes the technical problems that arose during the formalization of the structures and the rationale for certain mathematical decisions 4.

Chapter 1

Preliminaries

With the advancement of technology, computers quickly took on the burden of manual labor in numerical studies. It was also envisioned as early as 1959 [2] that they will assist logicians and mathematicians in proving theorems. However, it was known that the halting problem renders any attempt at creating fully automated proof system unsuccessful. This limited the use of computer proof systems until the emergence of advanced large language models (LLMs). Currently, LLMs propose mathematical proofs with potentially flawed reasoning, since they lack the necessary verification. Proof assistants such as Lean, Coq, and others provide a rich framework to fill this gap. A year ago, Deepmind's team successfully achieved the silver medal in international Olympiad of mathematics [3]. Their state of the art models, AlphaZero, generates potential solutions and then prove or disprove them in Lean.

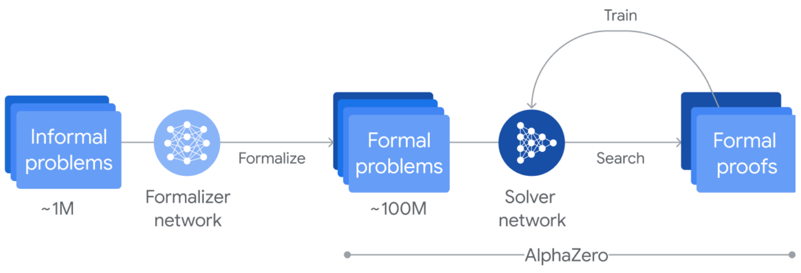

Figure 1.1: Textbook (informal) problems are converted into Lean (formal) problems and then solved by AlphaZero. The accepted solutions are used for further training. Source: Deepmind

This can motivate researchers to put more effort at developing these proof systems. Furthermore, it is not uncommon for complicated theorems to contain flaws, even in the well-accepted literature [4]. Stating the argument in a proof system that by design guarantees the soundness of the reasoning prevents potential human's errors, albeit at the cost of more effort in establishing seemingly trivial results. Moreover, formalizing mathematical problems for a proof assistant requires expressing assumptions clearly and explicitly. Not only this helps interested people like students to read proofs more easily and understand every small step necessary to achieve the goal, but also a mathematician that develops a theory may further drop unnecessary assumptions, thereby making more powerful theorems in the process.

In the following pages, a brief introduction on Lean, and mathematical theorems of interest is given. This is followed by implementation details of these theorems in Lean. Lastly, the conclusion presents our main findings.

1.1 Lean

Lean is a proof assistant with functional programming capabilities developed by Leonardo de Moura in 2013. Ever since then, it has gone through multiple major updates. Mathlib, the most notable open source library of this language, is also still in development [5]. This thesis exclusively uses the stable version of Lean4 and Mathlib4.

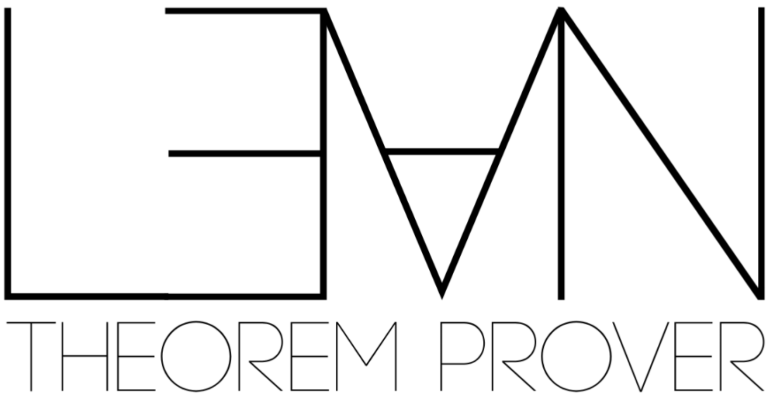

Figure 1.2: Lean, the programming language developed by Microsoft and many open source developers. Source: Microsoft

In a strongly typed language like C, every value has a type. For example, `int v = 2;` means the value `2` has the `int` type, where `int` is programmed to encapsulate the notion of integers as a type. If someone attempts to assign a value from another type to `v` (e.g. `v = "Hi";` with `"Hi"` being a string), the compiler throws a warning. Lean takes this idea to the extremes; it treats every logical proposition as a type. In other words, roughly speaking, to define the very notion of a proposition one first creates a class that encapsulates the logic of that proposition. Then, to prove a proposition one must construct a value (in other words, a proof) that inhabits that type. This point of view prevents assigning a value (a proof) of one type (proposition) to another one, making logical statements reasonable. One may wonder, how is it possible to conclude one proposition from another, if type safety stated above prevents such an assignment? To answer this question, consider the following:

Firstly, the falsehood should never become provable. So whatever class that wants to encapsulate the notion of falsehood, should have no constructor. Negation of a proposition is proved by showing that the proposition itself has no constructor. Similarly, the logical conjunction ”and” is represented in Lean as a type that pairs two propositions, like a list with two elements. To construct a proof of such a conjunction, one must supply a proof for each of the component propositions (create an object of each type). Existence is similar to logical "and", except that it takes a pair of a witness and the proof of the proposition corresponding to that witness. The logical "or" unlike logical "and" has two constructors corresponding to each proposition. So providing proof of one proposition is sufficient for proving logical "or".

Perhaps the most interesting logical operator is implication, as it is treated as a function. It takes proof of a proposition, and returns proof of another proposition (If P_1 is true in $P_1 \to P_2$, logically P_2 is proven, so $P_1 \to P_2$ is like a function from P_1 to P_2). The logical "for all" is also a function, except that its output depends on the type of its input. For example, $\forall x, P(x)$ is a function that takes any x and outputs $P(x)$. The correspondence between logical operators and their respective programming counterparts is known as Curry-Howard correspondence [6] [7].

Logic (Formula)	**Type Theory**	**Can be represented as a**
$\neg A$	$A \to \bot$	Class without constructor
$A \wedge B$	Product type $A \times B$	List with two inputs: $\langle a, b\rangle$
$A \vee B$	Sum type $A \bigoplus B$	Tagged union: $a \bigoplus b$
$\exists x. P(x)$	Σ-type $x \times P(x)$	List of a witness x with its proof $P(x)$
$A \to B$	Function type $A \to B$	Function from a to b
$\forall x, P(x)$	dependent product type	Function from any x to $P(x)$

Table 1.1: The Curry–Howard correspondence. Small letters correspond to object of their respective type, i.e. a is an object of type A.

It is worth mentioning that while Lean supports all these operators (and much more) in its standard library, everything except implication and logical "for all" is not truly kernel's level native [8], making the kernel itself extremely small.

New types can be declared via *inductive* keyword. For example, the following illustrates how natural numbers are constructed in Lean:

```
inductive Nat where
  -- Constructor of zero, the smallest natural number.
  | zero : Nat
  -- Constructor of other natural numbers.
  | succ (n : Nat) : Nat
```

As one can see, it has two constructors, one makes zero itself and requires no input, and another one that takes a natural number and returns the successor of that number. Note that there is nothing special about the name of *zero* or *succ (n: Nat)*, i.e. the following would be an equivalent type:

```
inductive mNat where
  | mzero : mNat
  | msucc (n : mNat) : mNat
```

There are other ways of declaring new types, such as using *structure* for inductive types with 1 constructor, or *class* keyword to take advantage of the class inference system.

Lean is a powerful functional language: it supports pure functions, higher order functions, pattern matching, etc. To continue our previous example, one defines addition for natural numbers as follows:

```
def Nat.add : Nat → Nat → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

$Nat \to Nat \to Nat$ is a type that takes two natural numbers and returns a natural number. We can fix a natural number b in *add* to get a function $Nat \to Nat$ as *add* b equivalent to $b + .$ with dot referring to an empty slot to be filled later, another expected property of a functional language.

Lastly, to prove a proposition, one uses *theorem* keyword (or *lemma*, another syntactic sugar). It is worth mentioning that *theorem* is not different from *def* keyword, except that the type of its output is always a proposition. It is basically a function from its assumptions to the proof of its output, a proposition. There are two modes to prove a theorem, term and tactic mode. In the term mode, one uses Curry-Howard correspondence directly. As an example, the commutativity of logical "and" is proven as:

```
/-! A function from assumption h: p ∧ q to q ∧ p,
    proves p ∧ q → q ∧ p. And.intro, the sole
    constructor of logical `And`, is given the
    proof of q (h.right) and p (h.left) .-/
theorem and_comm': p ∧ q → q ∧ p :=
  fun h: p ∧ q => And.intro h.right h.left
```

As one might suspect, for more complicated proofs, term mode becomes rather cumbersome and tactic mode comes in handy.

```
theorem and_comm': p ∧ q → q ∧ p := by tauto
```

The *by* keyword activates the tactic mode, and *tauto* is a tactic for proving tautologies. Note that there is nothing special about tactic mode; under the hood, the available relevant theorems are searched and used to simplify and prove the proposition if possible. For example, in this case, *and_ comm* already exists in the standard library and *tauto* can take advantage of them to prove the results. Lean supports many more tactics, the full list is available in official website [9].

1.2 Infinite Tensor Products

This section begins by explaining the motivation behind the definition of infinite tensor products. In the literature they appear under several names such as quasi local, restricted, or incomplete tensor products. After outlining the motivation, it proceeds to a careful explanation of their construction.

1.2.1 Motivation from quantum spin systems

In elementary quantum mechanics, individual systems are combined to a joint system by means of tensor products. For example, if we associate a Hilbert space $\mathbb{C}^2$ to each spin $1/2$ system, then the joint Hilbert space would be $\mathbb{C}^2 \otimes \mathbb{C}^2 \simeq \mathbb{C}^4$. Since it is a finite-dimensional vector space, no closure with respect to the norm is necessary as all Cauchy sequences automatically converge.

Similarly, the bounded operators $\mathcal{B}(\mathbb{C}^2)$ acting on $\mathbb{C}^2$ are isomorphic to $M_2(\mathbb{C})$, so for the joint observables we also have $M_2(\mathbb{C}) \otimes M_2(\mathbb{C}) \simeq M_4(\mathbb{C})$. But what if we keep adding more and more particles to this one-dimensional system? At the limit of infinite particles, one attempt could be defining the Hilbert space as $\mathcal{H} = \bigotimes_{i=0}^{\infty} \mathcal{H}_i = \bigotimes_{i=0}^{\infty} \mathbb{C}^2$ where i ranges over natural numbers. However, one can immediately see why this construction fails: Take an arbitrary non-zero element from each Hilbert space $v_i \in \mathcal{H}_i$, then the sign of the canonical inner product $\langle \bigotimes_i v_i, \bigotimes_i -v_i \rangle = \prod_{i=0}^{\infty} -v_i^* v_i$ becomes ambiguous. Therefore, this construction fails to become a Hilbert space with respect to this obvious choice of inner product.

To address this difficulty, two approaches are considered. The first is to select a suitable global Hilbert space that avoids this issue altogether. The second is to begin with the space corresponding to observables instead, and proceed from that setting to define states as functionals acting on this space. We begin with an examination of the first approach, including the issues that emerge from its implementation. Subsequently, the second approach is considered in detail, and the reasons it is preferred.

Measuring spins along a fixed axis, for example the z axis, yields outcomes in a sequence of $\{-1, 1\}$. Since physical measurements only reveal a finite number of excitations, let us assume that “physical“ sequences are the ones with finitely many -1s. If we denote the collection of all such sequences as S^+, then the Hilbert space can be defined as $\mathcal{H}^+ = l^2(S^+)$. That is, the space of infinite sequences labeled by elements of S^+. The observables are bounded operators acting on this space. While this method works, it has some flaws. For example, there is no reason to prefer $+$z-direction over any other direction. While this may look harmless at first, note that no physical operator can convert a sequence with finitely many -1 to a sequence with finitely many 1, as it requires the application of infinitely many local operators. So in fact $l^2(S^+) \neq l^2(S^-)$ with S^- denoting the collection of sequences with finitely many 1. A priori, it is not clear what choice should be made.

To circumvent this issue, one can start from observables instead. Let us consider $\mathcal{A}_i = M_2(\mathbb{C})$ as an algebra of observables acting on a single site i. Furthermore, $\mathcal{I}_i$ denotes the identity operator that acts trivially on the site i. $\mathcal{I}_i$ allows us to define the operators that act on trivially everywhere, except for a finite region. For example, $\mathcal{A}_1 \otimes \mathcal{I}_2 \otimes \mathcal{I}_3$... acts non-trivially only on the first site. In other words, the family of identity operators "pad" local operators to enlarge their domain. The algebra of observables can be constructed as $\mathcal{A} = \bigcup_{\Lambda \subset \mathbb{N}} \bigotimes_{i \in \Lambda} \mathcal{A}_i$, with Λ being a finite subset of natural numbers. $\mathcal{A}$ is the algebra of all observables that act trivially everywhere, except for finitely many sites. In the next section $\mathcal{A}$'s construction is explained in more details. One can show that the norm closure of $\mathcal{A}$ is a C^* algebra and therefore, it can be represented as bounded operators acting on a Hilbert space through the GNS construction. Once observables are specified, states are set to be positive functionals acting on observables and normalized by the identity, that is, $S = \{\omega | \forall a \in \mathcal{A}, 0 \leq \omega(a^*a) \wedge \omega(\mathcal{I}) = 1\}$ denotes the set of all states. In contrast to the first approach, where the selection of a particular reference configuration lacked justification, the identity elements $\mathcal{I}_i$ are natural reference vectors for this construction.

1.2.2 Mathematical theory

For the mathematical description, we will follow the exposition in a set of lecture notes [10] by Guichardet. The final aim is to define infinite tensor products as a *direct limit* of a *directed system*. These two concepts will be introduced later.

We fix a field R and will consider all vector spaces occurring below to be over that field. Let S be a set, and for each $a \in S$, let V_a be a vector space. For any *finite* subset $F \subset S$, the tensor product

$$V_F := \bigotimes_{a \in F} V_a$$

is well-defined. To extend the definition, now assume that a non-zero distinguished element $I_a \in V_a$ has been singled out for each $a \in S$. These elements will generalize the role of the identity operator in the construction above. Now, we want to define a linear embedding that maps tensor products over smaller sets to larger ones. Intuitively, the measurement devices act on finite regions. But these regions, in principle, can be extended. This is what we are trying to model, and it requires the possibility of breaking tensor products:

$$A : \bigotimes_{a \in F} V_a \simeq \bigotimes_{a \in F'} V_a \otimes \bigotimes_{a \in F \setminus F'} V_a \tag{1.1}$$

with F' being a subset of F. This isomorphism defines a natural linear embedding P_{F_s,F_l} from a smaller set F_s to a larger F_l indexing set:

$$P_{F_s,F_l}(x) = A(x \otimes \bigotimes_{a \in F_l \setminus F_s} \mathcal{I}_a) \tag{1.2}$$

with $F_s \subseteq F_l$ and x being an element of $\bigotimes_{a \in F_s} V_a$. Intuitively, if x is a product tensor $x = \bigotimes_{a \in F_s} v_a$, then P_{F_s,F_l} for every $a \in F_l$ simply chooses v_a if $a \in F_s$ else $\mathcal{I}_a$ is chosen. The end result is just taking the product tensor of the chosen vectors. Lastly, it is easy to show that P_{F_s,F_l} satisfies the following properties:

$$\begin{aligned} &1) \quad P_{F,F}(x) = x \\ &2) \quad \forall F_i \subseteq F_j \subseteq F_k, \quad P_{F_i,F_k} = P_{F_i,F_j} \circ P_{F_j,F_k}. \end{aligned} \tag{1.3}$$

Furthermore, S is a directed set since its subsets are partially ordered by $F_1 \leq F_2$ iff $F_1 \subseteq F_2$, and for any $F_1 \leq F_2$ there exist F_3 such that $F_1 \leq F_3$ and $F_2 \leq F_3$ (simply take $F_3 = F_1 \cup F_2$). The tensor product and the embedding P_{F_s,F_l} are said to form a directed system over the directed set S.
Now, we are able to take the direct limit. More information is available at [11]. The direct limit essentially glues finite objects V_F together to form $\mathcal{V}$. This is done by first taking the direct sum of $C = \bigoplus_{F \in pow(S)} V_F$, and identifying each vector space V_F with its canonical image $\mathcal{V}_{\mathcal{F}}$ in C. Next, let D be the submodule of C generated by all elements of the form $v_{F_S} - P_{F_S,F_L}(v_{F_S})$ with $v_{F_S} \in \mathcal{V}_{\mathcal{F}}$. The direct limit $\mathcal{V} = C/D$ is then obtained by modding out D, denoted as

$$\mathcal{V} = \lim_{\rightarrow} V_F. \tag{1.4}$$

1.3 Tensor product of partially ordered spaces

Similar to infinite tensor products, this section starts by a motivating example in Quantum mechanics. After demonstrating the motivation, the mathematical details and some lemmas will be stated.

1.3.1 Why are they interesting?

Modern theories are often evaluated on empirical grounds. While this approach effectively eliminates those that yield incorrect predictions, it becomes inadequate when technological limitations or insufficiently precise data prevent decisive testing. For this reason, a unifying mathematical framework with a high degree of generality is required to encapsulate these theories in a comparable manner. Such a framework allows one to discern the essential differences between two theorems, not merely in terms of their predictive outcomes, but more importantly in relation to the underlying assumptions that remain otherwise concealed.

Here, we introduce such a framework that is rich enough to not only cover our currently widely accepted theories such as quantum and classical mechanics, but also models with vastly different outcomes such as PR-Boxes [12]. Following the work of Hulanicki and Phelps [1], we model measurement devices as a convex cone that contains a distinguished element in its interior. It will follow that this structure is also salient (i.e., no line goes through the origin) and therefore it has a natural order. As an example, the collection of all projective operator-valued measures (POVMs) in quantum mechanics forms such a structure.

To illustrate this, consider the probability of measuring a spin-1/2 system in +z-direction that can be calculated as $tr(\rho P_z^+)$ where ρ is a density matrix and P_z^+ represents the projected operator-valued measurement (POVM) of the Pauli matrix σ_z in the up direction.

1. Convexity: If the axis of the measurement is chosen at random, e.g. between x and z with probability λ for x direction, then it is not hard to show $P = \lambda P_x^+ + (1-\lambda)P_z^+$ is also a POVM and $tr(\rho P)$ is the probability of spin-up outcome in such a measurement.

2. Distinguished element: All density matrices, when paired with identity matrix, evaluate to 1. One can then prove that this property is unique to the identity matrix.

3. Cone: The elements of these collection can be scaled by a positive number, preserving their positivity, and normalized when needed. Apparently, this makes calculations easier.

4. Algebraic interior: By assuming that the distinguished element is inside Algebraic interior, one ensures normalization works as intended. For example, an Orthant as a self-dual structure can represent both measurement devices and states at the same time with this property.

1.3.2 Mathematical background

Perhaps one of the most minimalist formalizations of such tensor products is proposed by Hulanicki and Phelps [1] as follows:

Let V be a vector space, P a convex cone, and e a distinguished element in the algebraic interior of P. We define the algebraic interior of P as

$$core(P) = \{v \in P | \forall w, \exists \epsilon > 0, \forall \delta, |\delta| \leq \epsilon \rightarrow v + \delta w \in P\} \tag{1.5}$$

The triplet (V, P, e) is called a partially ordered vector space. It is easy to see that the salient property, along with the algebraic interior definition, guarantees $e \neq 0$.

Lemma 1.3.1. *If P is salient then $e \neq 0$, provided that V is nontrivial vector space.*

Proof. If $e = 0$, then for an arbitrary non-zero vector w, the definition of the algebraic interior definition implies that both ϵw and $-\epsilon w$ are members of P, thus contradicting the salient property. □

Interestingly, in finite dimension, if the topological interior is non-empty and convex, it coincides with the algebraic interior [13]. The advantage of using algebraic interior as opposed to topological interior is that no topology on V is required. This significantly simplifies the process of taking the tensor product of such vector spaces, since there is no ambiguity in the choice of the tensor product itself.

Now, consider the set S of all dual vectors that evaluate to a nonnegative number against elements of P and normalized by e, i.e.

$$S(P) = \{dv | \forall v \in P, 0 \leq dv(v)\} \cap \{dv | dv(e) = 1\} \tag{1.6}$$

It is possible to show that S is weak* compact and convex. Furthermore, we assume S is separating:

$$\forall v, v \neq 0 \to \exists dv \in S, dv(v) \neq 0 \tag{1.7}$$

From which the following lemma follows:

Lemma 1.3.2. *If S is separating then P is salient.*

Proof. Let us assume that both w and $-w$ are nonzero elements of P, then by the separating property, there exists a dual vector dv in S such that $dv(w) \neq 0$. Since the evaluation of dual vectors is nonnegative, it follows that $0 < dv(w)$ and therefore $0 > dv(-w)$, but dv should be nonnegative on $-w$ as it is an element of P, contradicting the initial assumption. $\square$

Here, we slightly deviated from the original work [1] by not assuming P is salient, as it is provable by the separating property of S. Moreover, while the separating implies P is salient, the converse is not true as S can be empty [14]. Now, it is possible to construct tensor products as follows:

Let A be a possibly infinite set of indices. Then for each $a \in A$, V_a, P_a, S_a is an arbitrary vector space, the cone in V_a and the corresponding set of nonnegative dual vectors. If F is a finite subset of A, it is possible to define

$$V_F = \bigotimes_{a \in F} V_a \tag{1.8}$$

To create the corresponding S_F, Hulanicki takes the Cartesian product of each S_a [1]

$$S_F = \prod_{a \in F} S_a \tag{1.9}$$

Naturally, the elements of S_F are equipped with a product topology of weak* topologies. The canonical pairing is

$$v(s_F) = \sum_i \prod_{a \in F} \langle v_a^i, s_a \rangle \tag{1.10}$$

with $v \in V_F$ and $s_F \in S_F$. Note, since v is a general tensor in V_F it can be decomposed into a finite sum of product tensors v^i. Equation (1.10) is just extending the pairing between the product tensor v^i paired with s_f by linearity. Two natural convex cones immediately arise as

$$P_{max}(F) = \{v \in V_F | \forall s_F \in S_F, 0 \leq v(s_F)\} \tag{1.11}$$

$$P_{min}(F) = \{v \in V_F | v = \sum_i \bigotimes_{a \in F} v_a^i \wedge v_a^i \in P_a\} \tag{1.12}$$

Note that $P_{min}(F) \subseteq P_{max}(F)$. To define a partially ordered tensor product space $(V_F, P_{min}(F), e_F)$, it suffices to find an element $e_F \in core(P_{min}(F))$.

Theorem 1.3.1. *Let $e_F = \bigotimes_{a \in F} e_a$ such that $\forall a \in F, e_a \in core(P_a)$ then $e_F \in core(P_{min}(F))$.*

Proof. We proceed by performing tensor rank induction on the elements of V_F, followed by an induction on the length of product tensors. Our goal is to show that

$$\forall v \in V_F, \exists \epsilon > 0, \forall \delta, |\delta| \leq \epsilon \to e_F + \delta v \in P_{min}(F)$$

First, assume for two arbitrary elements $a, b \in V_F$, $\exists \epsilon_a > 0, \forall \delta, |\delta| \leq \epsilon_a \to e_F + \delta a \in P_{min}(F)$ and $\exists \epsilon_b > 0, \forall \delta, |\delta| \leq \epsilon_b \to e_F + \delta b \in P_{min}$ is true. Let us choose $\epsilon_{ab} = min(\epsilon_a, \epsilon_b)/2$. Note that since $P_{min}(F)$ is a cone

$$\forall \delta, |\delta| \leq \epsilon_{ab} \to e_F/2 + \delta a \in P_{min}(F)$$

$$\forall \delta, |\delta| \leq \epsilon_{ab} \to e_F/2 + \delta b \in P_{min}(F)$$

and from convexity, it follows that $e_F + \epsilon_{ab}(a+b) \in P_{min}(F)$ proving the proposition holds under addition of tensors. The proof will be complete once we verify the claim for product tensors. We proceed by induction on the tensor length: the statement is true for tensors with length 1 trivially; therefore we assume the induction hypothesis

$$\forall w, \exists \epsilon, \forall \delta, |\delta| \leq e_{F-\{q\}} + \delta w \in P_{min}(F - \{q\})$$

with q being an arbitrary singleton $q \in F$ to show $e_F \in core(P_{min}(F))$ for an arbitrary product tensor w. Using the isomorphism defined in (1.1), one can write $A(w) = u \otimes v$ with $u \in V_{F-\{q\}}$ and $v \in V_q$. Applying the induction hypothesis on u gives ϵ_u, and similarly for v one chooses ϵ_v from the main assumption of the theorem.

It suffices to choose $\epsilon_F = min(\epsilon_u, \epsilon_v)^2$ to prove that

$$\forall w, \exists \epsilon_F, \forall \delta, |\delta| \leq e_F + \delta w \in P_{min}(F)$$

Without loss of generality, let us assume $\epsilon_F = \epsilon_v^2$, that is, $\epsilon_v \leq \epsilon_u$, then it is easy to see that

$$a = e_u + \epsilon_v u \in P_{min}(F - \{q\})$$

$$b = e + \epsilon_v v \in P_q$$

$$c = e_u - \epsilon_v u \in P_{min}(F - \{q\})$$

$$d = e - \epsilon_v v \in P_q$$

$$\frac{A}{2}(a \otimes b + c \otimes d) = e_F + \epsilon_v^2 w \in P_{min}(F)$$

$$\frac{A}{2}(a \otimes d + b \otimes c) = e_F - \epsilon_v^2 w \in P_{min}(F)$$

The last two equalities follow because if $x \in P_q$ and $y \in P_{min}(F - \{q\})$ then $A(x \otimes y) \in P_{min}(F)$. This completes the proof, since $\forall \delta, |\delta| \leq \epsilon_F, e_F + \delta w \in P_{min}(F)$.

□

Lastly, similar to (1.6), one can define S_{max} and S_{min} as

$$S_{max/min} = S(P_{min/max}) = \{dv | \forall v \in P_{min/max}, 0 \leq v(dv)\} \cap \{dv | e_F(dv) = 1\} \quad (1.13)$$

Chapter 2

Implementation

A brief description for formalization of the introduction is given here. To keep things short, the details of certain theorems or definitions are omitted. Furthermore, unless necessary, we avoid describing the available theorems in Mathlib4. The reader is encouraged to look at Mathlib4's documentation [5] for more details.

2.1 Isomorphisms of PiTensorProduct

In Lean4, `PiTensorProduct` encapsulates the notion of tensor products indexed by an arbitrary type. It is formalized as a Quotient of an equivalence relation `Eqv` that make tensor products behave in the expected manner. For example, `Eqv` asserts all product tensors with a zero factor are equivalent to zero or one can factor out scalar multipliers from each tensor factors.

`Eqv` is an equivalence relation on `FreeAdddMonoid` of product functions $\Pi_{i \in I} c_i$ with I being an arbitrary (potentially infinite) type and each c_i is an element of a module. The product functions would later give raise to product tensors. `FreeAdddMonoid`, on the other hand, defines the formal sum of these functions. It is necessary, because we want to make tensor product space a module (that is, addition of tensors must be defined and behave properly). The way `FreeAdddMonoid` works is quite simple, it just inserts each product function into a `List` and interpret the addition as concatenation of these lists. Lastly, one has to show that `Eqv` preserves addition and is actually an equivalence relation. This is established by using `addConGen` that takes a relation, and make it an equivalence relation that preserves addition.

For our purposes, we would like to implement equation 1.1. The closest match in the standard library is `tmulEquivDep`. It is a natural isomorphism between `PiTensorProduct`s and binary `TensorProduct`s. It shows that for a family of vectors indexed by a direct sum $N : i_1 \oplus i_2 \to \mathit{Type}$ for arbitrary type i_1 and i_2, one has

$$\bigotimes_{i : i_1 \oplus i_2} N_i \approx \bigotimes_{j : i_1} N^l_j \otimes \bigotimes_{k : i_2} N^r_k \tag{2.1}$$

where N^l and N^r are to be understood as functions that put j and k in the left and right side of disjoint sum and use N to produce a vector. Note that the use of colon emphasizes the fact that i_1 and i_2 are not `Set`s, but arbitrary `Type`s.

As is evident, the most immediate difficulty arises from the use of `Sum` in the indexing. Let i_1 and i_2 denote the `Type`s of even and odd natural numbers. Given the right-hand side of 2.2, one might hope to obtain a tensor product indexed by all natural numbers on the left hand side. However, the `Type` (i :$i_1 \oplus i_2$) is not definitionally equal to the `Type Nat`. The type of `Nat` is defined inductively. In general, arbitrary `Type`s are not defined by predicates, so proving an equality between two types is not possible. In this case, for example, one cannot establish a type-level equality between the disjoint sum of even and odd numbers and the type of all natural numbers.

Furthermore, the set difference on the right hand side of equation 1.1 is not necessarily meaningful for arbitrary types. This is why we need to fix the indexing types to Sets and show that the disjoint sum is as same as taking disjoint union between the sets. Following the previous example, since Sets unlike arbitrary types are defined by their underlying predicates, it will be then easy to show that $s_1 \cup s_2$ (with s_1 and s_2 being set of even and odd numbers), is indeed equal to set of all natural numbers. This motivates the use of Sets rather than arbitrary index types and requires certain isomorphisms to be established, which are explained briefly below. Given $V : \iota \to \mathit{Type}$ as a family of vector spaces and with ι being some arbitrary type, one can prove

$$\bigotimes_{i \in s_1 \cup s_2} V_i \approx \bigotimes_{j \in s_1} V_j \otimes \bigotimes_{k \in s_2} V_k \tag{2.2}$$

through `PiTensorProduct.reindex` in the combination of `tmulEquivDep`. `reindex` takes an equivalence between two types (In our case, `Equiv.Set.union`: $s_1 \oplus s_2 \approx s_1 \cup s_2$ for disjoint sets) and reindex a `PiTensorProduct` as follows:

$$\bigotimes_{i : i_1} X_i \approx \bigotimes_{j : i_2} X_{e(j)} \tag{2.3}$$

with e being an equivalence between two arbitrary types i_1 and i_2 and $X : i_1 \to \mathit{Type}$. Here is the code snippet that demonstrates how this works:

```
variable {ι : Type*} {R : Type*} [CommSemiring R] {V : ι → Type*}
variable [∀ i, AddCommMonoid (V i)] [∀ i, Module R (V i)]

def tmulUnionEquiv (hdisj : Disjoint S₁ S₂):
    ((⨂[R] (i₁ : S₁), V i₁) ⊗[R] (⨂[R] (i₂ : S₂), V i₂))
      ≃ₗ[R] ⨂[R] (i : ↑(S₁ ∪ S₂)), V i :=
  (tmulEquivDep R (fun i => V ((Equiv.Set.union hdisj).symm i))).trans
    (reindex R (fun i : (S₁ ∪ S₂) => V i) (Equiv.Set.union hdisj)).symm
```

Now we can prove 1.1 from `tmulUnionEquiv` since S and $T - S$ are always `Disjoint`:

```
def tmulUnifyEquiv {S T : Set ι} (hsub : S ⊆ T):
    (⨂[R] i₁ : S, V i₁) ⊗[R] (⨂[R] i₂ : (T \ S), V i₂) ≃ₗ[R]
    ⨂[R] i : T, V i :=
  (tmulUnionEquiv (disjoint_sdiff_right)).trans
  (reindex R (fun i : (S ∪ T \ S)  V i)
      (Equiv.subtypeEquivProp (union_diff_cancel hsub)))
```

Here, `Equiv.subtypeEquivProp` just establishes the fact that $(S \cup (T - S)) \approx T$ through `union_diff_cancel`. Next, we would like to do induction on the length of `PiTensorProduct`. This is used, for example, in theorem 1.3.1. For this purpose, first we establish the equivalence between tensor products indexed by a `Subsingleton` type and an the elements of the underlying vector space as

```
def subsingletonEquivDep {ι : Type*} [Subsingleton ι] (q : ι) :
    (⨂[R] i : ι, V i) ≃ₗ[R] V q
```

`Subsingleton` is a type that contains at most one element, possibly none. An analogous equivalence was initially formulated for the `Unique` type, which contains exactly one element. However, the consistency with the existing isomorphism below led to the adoption of `Subsingleton`. To the best of the author's knowledge, the preference for `Subsingleton` over `Unique` in this context is not yet fully understood. At the moment of writing this thesis, `subsingletonEquiv` exists in mathlib4 library:

```
def PiTensorProduct.subsingletonEquiv [Subsingleton ι] (q : ι) :
    (⨂[R] (x : ι), V q) ≃ₗ[R] V q
```

However, the difference between `subsingletonEquiv` and `subsingletonEquivDep` is that the former uses a dependent instance of vector spaces as `[∀ i, Module R (V i)]` while the latter is a non dependent version [Module R (V q)]. Both are provable from each other. However, the proof of former from latter requires using `PiTensorProduct.congr` which at current version, is marked as `noncomputable`. This is why we used a different implementation as `subsingletonEquivDep`. To show this isomorphism from left to right, we used the fact that the underlying function `f` of a product tensor has only one index, and therefore it is isomorphic to `f q`, we can extend this by universal property to all tensors. Next, for the right to left: any given vector can be reinterpreted as a function that is zero everywhere, except at the given index `q` with the output being `V q`. Taking tensor product of all indices of such a function, results in a `PiTensorProduct`. However, since the indices are actually `Subsingleton`, only `V q` will be used in this tensor product. A rather obvious equivalence that immediately follows is

```
def singletonSetEquiv (q : ι) :
    (⨂[R] i : ({q} : Set ι), s i) ≃ₗ[R] s q :=
  subsingletonEquivDep (⟨q, by simp⟩ : ({q} : Set ι))
```

Finally, we can define the isomorphism between tensor product of a tensor with a vector and tensor products as

```
def tmulInsertEquiv {S : Set ι} {q} (hq : q ∉ S) :
    ((V q) ⊗[R] (⨂[R] i₁ : S, V i₁))
    ≃ₗ[R] (⨂[R] i₁ : (insert q S), V i₁) :=
  (TensorProduct.congr (singletonSetEquiv q).symm
    (LinearEquiv.refl _ _)).trans
  (tmulUnionEquiv (Set.disjoint_singleton_left.mpr hq))
```

The noteworthy part is that to use `tmulUnifyEquiv` we need to change the type of `V q` to `(⨂[R] (i : ↑(insert q S₁ - S₁)), V ↑i)`. It can be done by using `subsingletonEquivDep.symm`. What makes `tmulInsertEquiv` special is that it allows us to reinterpret any tensor with length L to tensor product of a tensor with length $L - 1$ and a vector.

2.2 Infinite Tensor Products

Given `tmulUnifyEquiv`, we can formalize the padding map defined in equation (1.2) as follows:

```
def extendTensor {S T : Set ι} (v : (i : ι) → V i) (hsub : S ⊆ T) :
    (⨂[R] (i : S), V i) →ₗ[R] (⨂[R] (i : T), V i) where
  toFun t := (tmulUnifyEquiv hsub) (t ⊗ₜ[R] (⨂ₜ[R] i : (T \ S), v i))
  map_add' := by simp [TensorProduct.add_tmul]
  map_smul' := by simp [←TensorProduct.smul_tmul']
```

Note that `v` refer to the family of distinguished elements $\mathcal{I}$ defined in section 2.2. The fact that this map is linear is immediate, as it is defined via a linear equivalence `tmulUnifyEquiv`. Next, we need to show that this maps creates a `DirectedSystem`:

```
instance : DirectedSystem
    (fun s : Finset ι => ⨂[R] (i : s), V i)
    (fun _ _ hsub => extendTensor hsub ids)
```

Induction on the rank of `PiTensorProduct` proves both properties specified in (1.3). Finally, taking direct limit of this map formalizes the equation (1.4).

```
def Finsupp :=
    DirectLimit
    (fun s : Finset ι => ⨂[R] (i : s), V i)
    (fun _ _ hsub => extendTensor hsub v)
```

This yields what is known as infinite tensor products. The choice of `Finsupp` as the name for `InfiniteTensorProduct` is motivated by the fact that `PiTensorProduct` already permits an infinite index set, which can obscure the intended meaning of infinite tensor products. The designation `Finsupp` makes explicit that the tensors involved have finitely many non-trivial factors and infinitely many trivial ones.

Two natural choices were available for taking the `DirectLimit`. The first is a general construction under `Order/DirectedInverseSystem`, applicable to arbitrary categories, which requires explicitly providing an instance of `DirectedSystem`. The second, specialized for modules, is defined in `Algebra/Colimit/Module` and does not necessitate such an instance. Both constructions align with the treatment in [11], as discussed in section 2.2. After careful consideration, we opted for `Module.DirectLimit`. Firstly, because it is more specialized an aligns exactly with the definitions given in section 2.2 but also more importantly, it is guaranteed that the two definitions are isomorphic when a `DirectedSystem` instance is supplied by `Module.DirectLimit.linearEquiv`.

Another issue concerned the choice between `Finset` and `Set` when creating an instance of `DirectedSystem`. The function `extendTensor` is general enough to accommodate arbitrary sets, including infinite ones, and it is possible to verify the required properties of a `DirectedSystem` for infinite sets. Using `Set` would therefore have yielded a higher level of generality. The intended notion of infinite tensor products, however, involves only finitely many non trivial factors, for the reasons given in section 2.2. Although employing `Sets` was feasible, the resulting structure lacked mathematical clarity.

2.3 Tensor product of partially ordered spaces

Following section 1.3, the algebraic interior definition can be formalized as:

```
variable (V : Type*) [AddCommGroup V] [Module 𝑅 V]
variable {W : Type*} [AddCommGroup W] [Module 𝑅 W]

/-- Algebraic interior, a.k.a radial kernel -/
def core (S : Set W) :=
  {v ∈ S | ∀ w, ∃ ϵ > (0 : 𝑅), ∀ δ, |δ| ≤ ϵ → v + δ • w ∈ S}
```

If a vector is an element of the algebraic interior of a set contained in a larger set, it is also an element of the algebraic interior of the larger set.

```
theorem mem_core_of_subset_mem_core {w} {s₁ s₂ : Set W}
  (hsub : s₁ ⊆ s₂) (hw : w ∈ core s₁) : w ∈ core s₂
```

Since the vector space itself from triplet (V, P, e) can be inferred from the type of elements of P or e, we can define it as

```
structure OrderCone extends ConvexCone 𝑅 V where
  ref : V
  hcore : ref ∈ (core carrier)
  pointed : ConvexCone.Pointed toConvexCone
```

where `ref` refers to e and `hcore` is the proof that it is in the algebraic interior. Since `OrderCone` extends `ConvexCone`, it also has a `carrier` field that formalizes the set of elements of P. It is noteworthy that we do not assume `ConvexCone.Salient` for the `carrier` as this property follows from later assumptions. Additionally, while `ConvexCone` ensures convexity and cone property of the `OrderCone`, it does not enforce 0 to be an element of the `carrier` which is why the `pointed` field is explicitly required. It is possible to prove that `o : OrderCone W` is a generating cone:

```
theorem generating : Set.univ = {z| ∃x ∈ o, ∃y ∈ o, z = x - y}
```

It follows from the fact that any vector x can be written as the difference between $\frac{1}{\epsilon}(e + \epsilon x)$ and $\frac{1}{\epsilon}e$. Clearly both of these vectors are elements of P. Next, we formalize S as

```
/-- Set of all positive dual vectors on the order cone,
    normalized by fixing their evaluation on `OrderCone.e` to 1 -/
def PosDual (o : OrderCone W) : Set (Dual R W) :=
  {s | ∀ v ∈ o, 0 ≤ s v} ∩ {s | s o.ref = 1}

def PosDual.separating : Prop :=
  ∀ {w}, w ≠ 0 → ∃ f ∈ PosDual o, f w ≠ 0
```

where `PosDual.separating` is assumed only in the required context, for example in

```
theorem OrderCome.salient (hs : PosDual.separating o) :
  ∀ x ∈ o, x ≠ 0 → -x ∉ o

instance : PartialOrder W :=
  ConvexCone.toPartialOrder o.toConvexCone o.pointed (o.salient hs)

instance : @IsOrderedAddMonoid W _ (instPartialOrder hs) :=
  ConvexCone.toIsOrderedAddMonoid o.toConvexCone o.pointed (o.salient hs)
```

As stated in the section 1.3, dual vectors are equipped with the weak* topology. Unfortunately, while `WeakDual` is already defined in Mathlib4 under Topology/Algebra/Module/WeakDual.lean; it requires a preexisting topology on the vector space V itself. Since in our case this is not available, we repeat the definitions and results for algebraic dual `Dual` as

```
variable {R : Type*} [CommRing R] [TopologicalSpace R]
variable {V : Type*} [AddCommGroup V] [Module R V]

instance instAddCommGroup: AddCommGroup (Dual R V) :=
  WeakBilin.instAddCommGroup (dualPairing R V)

instance instModule: Module R (Dual R V) :=
  WeakBilin.instModule (dualPairing R V)

instance instTopologicalSpace: TopologicalSpace (Dual R V) :=
  WeakBilin.instTopologicalSpace (dualPairing R V)
```

This lets `PosDual` elements inherit the weak* topology when required. It can be shown that this set is compact

Lemma 2.3.1. *S is point-wise bounded.*

Proof. The evaluation of any vector w on all elements of S is bounded by $\frac{1}{\epsilon}$. This follows from the fact that $e + \epsilon w$ and $e - \epsilon w$ are both elements of P and therefore, the evaluation of any $f \in S$ on these elements must be nonnegative by definition. Since $f(e) = 1$, therefore $0 \leq 1 + \epsilon f(w)$ and $0 \leq 1 - \epsilon f(w)$ which implies $\forall w, \exists \epsilon, \forall f \in S, |f(w)| \leq 1/\epsilon$. □

Lemma 2.3.2. *S is closed.*

Proof. The intersection of two closed sets is closed, therefore it suffices to show that both $\{f | \forall w \in P, 0 \leq f(w)\}$ and $\{f | f(e) = 1\}$ are closed. $\{f | f(e) = 1\}$ is closed, since the evaluation of vectors is a continuous action in weak* topology by definition and the singleton $\{1\}$ is closed; therefore the preimage of evaluation of e must be closed. $\{f | \forall w \in P, 0 \leq f(w)\}$ can be written as $\bigcap_w \{f | 0 \leq f(w)\}$. The intersection of closed sets is closed, and the set $\{t | 0 \leq t\}$ in real numbers is closed, hence the preimage of this set under any w (i.e $\{f | 0 \leq f(w)\}$) is closed. □

Theorem 2.3.1. *S is weak* compact.*

Proof. The compactness of S then immediately follows from Tychonoff's theorem that states the Cartesian product of compact sets is compact.

For any vector w we can define a compact set $c(w) = \{r | |r| \leq \frac{1}{\epsilon_w}\}$ where $\frac{1}{\epsilon_w}$ is the witness of lemma 2.3.1. By Tychonoff's theorem, the Cartesian product $C = \prod_w c(w)$ is also a compact set. It should be clear that S is a subset of C, because the evaluation of any dual vector $f \in S$ lies in the interval of $c(w)$ for a given w. From lemma 2.3.2 S is closed, and since it is a subset of compact set, it is compact. □

```
theorem DualPos.isCompact : IsCompact (PosDual o) := by
  let M : W → ℝ := fun w => (pointwise_bounded o w).choose
  let family : W → Set ℝ := fun w => Metric.closedBall 0 (M w)
  let prod := Set.pi Set.univ family
  have prod_compact : IsCompact prod := by
    simpa [prod, Set.pi] using isCompact_pi_infinite
        (fun w => isCompact_closedBall 0 (M w))
  have h_subset : dualEmbed '' (PosDual o) ⊆ prod := by
   simp [dualEmbed, Set.subset_def, prod, family]
   exact fun fembed hf w =>(pointwise_bounded o w).choose_spec ⟨fembed, hf⟩
  exact dualembed_dual_iscompact (isClosed o) prod_compact h_subset
```

The code snippet for `DualPos.isCompact` is added for demonstration only. What makes `DualPos.isCompact` slightly different from theorem 2.3.1 is the fact that in Lean, the linear structure of dual vectors must be forgotten first. Otherwise, due to type restrection, one cannot show that it is a subset of Caresian product of compact sets. `dualEmbed` map removes that linear structure.

The Cartesian product of a family of `PosDual` is formalized as

```
variable {ι : Type*}
variable {S : Set ι} (S' : Set ι)
variable {V : ι → Type*} [∀ i, AddCommGroup (V i)]
  [∀ i, Module R (V i)] (O : ∀i, OrderCone (V i))

def PiPosDual := Set.pi Set.univ (fun (i : S') => PosDual (O i))
```

Note that this set is trivially compact by Tychonoff's theorem. `PiPosDual` let us define P_{max} and P_{min} as

```
def MaximalProduct [Fintype S'] [Nonempty S'] :=
  {x | ∀ dv ∈ PiPosDual S' O, 0 ≤ embedVec x dv}

def MinimalProduct [Fintype S'] [Nonempty S'] :=
  {x| ∃ (n : ℕ) (vf : Fin n → (i : S') → V i),
      ∑ i, (PiTensorProduct.tprod R) (vf i) = x
      ∧ ∀ i, ∀ j : S', vf i j ∈ O j}
```

where `embedVec` defines the canonical pairing

```
/-- Embedding of a tensor product to a function
    on Cartasien product of a family of dual vectors -/
def embedVec: (⨂[R] i, V i) →ₗ[R] (((i:ι) → Dual R (V i)) → R) :=
  lift ({toFun vf dv := ∏ i: ι, (dv i) (vf i)})
```

It is worth mentioning that `Fintype` must be assumed on `MaximalProduct`, because `embedVec` by definition requires it. As was previously noted in section 2.2, the naive implementation of inner product of an infinite tensor product is ill-defined. On the other hand, while the inclusion of `Fintype` in the definition of `MinimalProduct` may appear redundant, it is necessary because virtually every nontrivial theorem (e.g., that `MinimalProduct` is a subset of `MaximalProduct`) requires this assumption. Explicitly stating `Fintype` in `MinimalProduct` prevents the need for excessive "omit" calls in later proofs as well.

Furthermore, when the indexing set of a `PiTensorProduct` is empty, tensor products become isomorphic to their ring; this may make requiring the indexing set to be `Nonempty` more subtle. After all, it seems no immediate contradiction arises. However, `MinimalProduct` is the formalized version of P_{min}, therefore it should satisfy the cone property (i.e multiplying elements by positive numbers result in another element in the set). Now consider the case where the indexing set for `MinimalProduct` is empty. By definition, elements of `MinimalProduct` consist of finite sums of product tensors. In this degenerate case, each product tensor reduces to the multiplicative identity 1 of the underlying ring. Therefore, `MinimalProduct` becomes the set of all natural numbers. Since our ring is the real numbers, `MinimalProduct` will not form a cone under multiplication of positive real numbers. This is why `Nonempty` is explicitly required. This represents an unusual case where an implicit mathematical assumption must be made explicit due to Lean's type system. The `Nonempty` requirement is similarly imposed on `MaximalProduct` for the following reason: to establish the existence of a distinguished element in its interior, one first demonstrates its presence in the interior of `MinimalProduct`. The desired conclusion then follows from the inclusion `MinimalProduct` in `MaximalProduct`.

The distinguished element e of P_{min} and P_{max} can be naturally defined as

```
def RefTensor [Fintype S'] [Nonempty S'] := ⨂ₜ[R] i : S', (O i).ref
```

which is a product tensor of a family of distinguished elements. An interesting lemma is the fact that we can factorize a distinguished element from `RefTensor`:

```
lemma RefTensor.factorized {q} [hq : q ∉ S] :
  RefTensor (insert q S) O =
  tmulInsertEquiv hq (RefTensor S O ⊗ₜ[R] (O q).ref) :=
  EquivLike.inv_apply_eq_iff_eq_apply.mp (by simp [EquivLike.inv]; rfl)
```

Moreover, as previously claimed, we can prove that `MinimalProduct` is a subset of `MaximalProduct`

Lemma 2.3.3. *P_{min} is a subset of P_{max}*

Proof. Choose an arbitrary element v of P_{min}. The evaluation of an element of S_F against v is $v(s_F) = \sum_i \prod_{a \in S} s_a(v_a^i)$. This is nonnegative, because it is formed through multiplication and addition of nonnegative numbers. $\square$

Now, we are at the position to prove theorem 1.3.1 using the results from section 2.1, specifically `tmulInsertEquiv` for the induction on the length of tensor products. Firstly, note that tensor product of an element of `MinimalProduct` with an element of `OrderCone` result in an element of `MinimalProduct` over larger indexing set:

```
theorem extended_mem
  {q} [hq : q ∉ S]
  {x : ⨂[R] i : S, V i} {v : V q}
  (hx : x ∈ MinimalProduct S O) (hv : v ∈ O q) :
  tmulInsertEquiv hq (x ⊗ₜ[R] v) ∈ MinimalProduct (insert q S) O
```

Furthermore, if a set is `Fintype`, we can perfrom the following induction on it

```
theorem Set.induction_on
  {α : Type*} {motive : Set α → Prop}
  {s : Set α} (hs: Fintype s) (empty : motive ∅)
  (insert : ∀ {q : α} {s : Set α},
  q ∉ s → motive s → motive (Set.insert q s)) : motive s

theorem refTensor_mem_core : RefTensor S O ∈ core (MinimalProduct S O)
```

The use of `Set.induction_on` in conjunction with `tmulInsertEquiv` facilitates induction over the length of `PiTensorProduct`. Upon applying `Set.induction_on`, the initial goal in proving the theorem `refTensor_mem_core` reduces to establishing the proposition for the empty set. However, since the indexing set is assumed to be nonempty, this case resolves by contradiction. The subsequent goal entails proving the proposition for a set augmented by an arbitrary element q. Here, two scenarios arise: either the set was empty prior to the insertion of q, in which case the proposition follows via `uniqueEquiv` (as referenced in Section 2.1) due to the resulting singleton indexing set; or the set was already `Nonempty`, allowing the induction hypothesis to be invoked. The induction hypothesis asserts the validity of the statement for the set before the insertion of q, enabling the application of `tmulInsertEquiv` to decompose

q and define the elements a, b, c, and d in accordance with Theorem 1.3.1. The conclusion is then derived using `extended_mem`. This demonstrates that although `PiTensorProduct` and `TensorProduct` are defined independently, a proposition proven for `TensorProduct` can be translated into a proof for `PiTensorProduct` via the isomorphism `tmulInsertEquiv`. Finally, S_{max} and S_{min} are formalized as

```
def MaxPosDual : Set (Dual R (⨂[R] (i : ↑S'), V ↑i)) :=
  {dv | ∀ v ∈ MinimalProduct S' O, 0 ≤ dv v} ∩
  {dv| dv (RefTensor S' O) = 1}

def MinPosDual : Set (Dual R (⨂[R] (i : ↑S'), V ↑i)) :=
  {dv | ∀ v ∈ MaximalProduct S' O, 0 ≤ dv v} ∩
  {dv | dv (RefTensor S' O) = 1}
```

The compactness and convexity of these sets can be proven by following the same steps as `PosDual`. We can also prove S_{min} is a subset of S_{max}

Lemma 2.3.4. $S_{min} \subseteq S_{max}$

Proof. An element dv of S_{min} evaluates to 1 when paired with the distinguished element, a property that S_{max} also satisfies. Moreover, it yields a nonnegative number against any element of P_{max}. But since $P_{min} \subseteq P_{max}$, this nonnegativity extends to all elements of P_{min}, thereby satisfying two required properties of S_{max}. □

Chapter 3

Conclusion

This project was initially motivated by formalizing De-Finetti's theorem within Lean. However, because essential mathematical infrastructure was absent, the objective shifted to developing the foundational basis required for such a formalization. To move toward this goal, certain challenges had to be addressed. The choice of mathematical framework had to strike a balance between competing requirements:

1. It needed to be sufficiently minimal to permit a clean and manageable formalization in Lean,
2. yet sufficiently general to avoid restricting later extensions or obstructing the formalization of more advanced results.

This issue was addressed by using a general framework proposed by [1]. Another challenge was selecting an appropriate implementation of these structures in Lean, since the system often allows several distinct formalizations of a single mathematical object. To manage this, the design decisions were mostly guided by conventions already established in Mathlib4, with emphasis on prioritizing the introduction of as few new definitions as possible and favoring proof strategies that remain concise and maintainable.

The results obtained here establish core definitions and properties of ordered vector spaces, which can form the groundwork necessary for infinite tensor products and De-Finetti's theorem for future works.

Chapter 4

Miscellaneous

This section outlines the challenges encountered during the formalization of order cones and the isomorphisms of `PiTensorProducts`. In particular, it explains the specific issues that arise when an initial topology is assumed on the vector space underlying the order cone, and how the distinction between propositional and definitional equality creates difficulties when formalizing mathematical structures in Lean.

4.1 Order Cone Definition

The starting point was the definition of order cones. Note that from here onward we use effect and order cone interchangeably. According to [15], a state space is a convex set that is closed and bounded in the Euclidean topology. An effect space is then defined as the set of dual vectors acting on elements of the state space whose values lie between 0 and 1. In addition, the effect space possesses a distinguished element, referred to as the identity, whose pairing with every element of the state space yields the value 1. Since there is no compelling reason to treat the state space as the primordial object, and objections to that choice have been noted in section 2.2, one may instead begin with the definition of an order cone and define the state space relative to it.

Therefore, for the purpose of formalization, the order cone was taken as the primary object. It can be defined as a closed convex cone equipped with a distinguished element located in its topological interior. The state space is then the set of all dual vectors that take nonnegative values on the elements of the order cone and take the value 1 on the distinguished element of the order cone. If probabilities are concerned, one can further define the effect space to be the intersection of the order cone with its negation. The negation of the order cone can be defined by subtracting each element of the cone from the distinguished element. The pairing between elements of the state space and effect space will be automatically between 0 and 1 as desired.

But here lies the issue: If this definition is extended to vector spaces equipped with arbitrary topologies, then for bipartite systems no unique choice for combining elements of two different order cone remains would exist, because the resulting topology of the bipartite system would depend on the particular manner in which the component systems are combined. For instance, in the case of C* algebras, two type of norms are often considered [16]:

If A_1 and A_2 are two C* algebras and $\pi_{1/2}$ is a nondegenerate representation of the elements of C* algebra as bounded linear operators on a Hilbert space, then we have:

$$|x|_{min} = sup\{\sum_j \pi_1(a) \otimes \pi_2(b) | \pi_i : A_i \to B(H)\} \tag{4.1}$$

$$|x|_{max} = sup\{\pi(x) | \pi : A_1 \otimes A_2 \to B(H)\} \tag{4.2}$$

These two norms do not necessarily coincide, resulting in different topologies for bipartite systems.

Initially, an attempt was made to define a general class for all conceivable bipartite systems as

```
class BiPartite
    (e₁: EffectCone N) (e₂: EffectCone M) (e:  EffectCone P)
    (B: (N × M) → P): Prop where
  id: B (e₁.id, e₂.id) = e.id
  embedding: ∀v₁ ∈ e₁, ∀v₂ ∈ e₂, B (v₁, v₂) ∈ e
  linear_left: ∀v₂, IsLinearMap R (fun v₁ => B (v₁, v₂))
  linear_right: ∀v₁, IsLinearMap R (fun v₂ => B (v₁, v₂))
  continuous_left: ∀v₂, Continuous (fun v₁ => B (v₁, v₂))
  continuous_right: ∀v₁, Continuous (fun v₂ => B (v₁, v₂))
  injective_left: ∀v₂, v₂ ≠ 0 →
    Function.Injective (fun v₁ => B (v₁, v₂))
  injective_right: ∀v₁, v₁ ≠ 0 →
    Function.Injective (fun v₂ => B (v₁, v₂))
```

a bi-continuous bilinear embedding from two effect cones to a larger one that is injective in each argument. Note that N, M, P correspond to different vector spaces and id refers to the distinguished element. Although this definition keeps the choice of how elements are combined flexible, it is difficult to use in practice because establishing that the larger effect cone e is indeed an effect cone, in particular that its distinguished element lies in its topological interior, is not a straightforward task in general. In comparison, by not assuming any initial topology on the vector spaces and using algebraic interior, one can prove in general that the algebraic tensor product of distinguished elements lies within algebraic interior 1.3.1, removing the burden of a proof for arbitrary multipartite systems. For this reason we chose to rely on [1] for this project.

4.2 Cast hell

While Lean's proof system closely resembles pen and paper proofs, there are certain places in which it diverges the expectations. For example, in `tmulUnifyEquiv` 2.1, one might expect `tmulUnionEquiv disjoint_sdiff_right` proves the goal right away, since `disjoint_sdiff_right` proves that two sets `S` and `T-S` are disjoint and therefore by `tmulUnionEquiv` conclude that `tmulUnifyEquiv` holds. However, the kernel does not know `S ∪ (T-S)` and `T` are equal. More precisely, their definition syntactically does not match. Therefore, using `tmulUnionEquiv` alone does not type check.

```
#check tmulUnionEquiv (R := R) (V := V)
    (disjoint_sdiff_right (s := S) (t := T))
/-- shows: (⨂[R] (i₁ : S), V i₁) ⊗[R]
    (⨂[R] (i₂ : (T \ S)), V i₂) ≃ₗ[R]
    ⨂[R] (i : (S ∪ T \ S)), V i --/
```

Note that the index set on the right-hand side is not `T`, even though it is mathematically equal to `T`. One must prove that `S ∪ (T - S)` $=$ `T` by using `union_diff_cancel`, which establishes this identity when `S` $\subseteq$ `T`. However, somewhat counterintuitively, the manner in which this equality is used matters for the proof of `tmulUnifyEquiv`. Consider the following:

```
def tmulUnifyEquiv' :
  (⨂[R] i₁ : S, V i₁) ⊗[R] (⨂[R] i₂ : (T \ S), V i₂)
  ≃ₗ[R] ⨂[R] i : T, V i := by
  have h := tmulUnionEquiv
    (R := R) (V := V) (disjoint_sdiff_right (s := S) (t := T))
  rw [union_diff_cancel hsub] at h
  exact h
```

Here, `S ∪ (T - S)` is being replaced by `T` by `rw` tactic and kernel no longer complains. However, the way the `rw` produces the correct type is through `casting` the unwanted isomorphism (right hand side indexed by `S ∪ (T - S)`) to the expected isomorphism tmulUnifyEquiv. This leads to certain difficulties that are explained shortly. Before addressing them, it is helpful to examine how `cast` operates. In Lean, `cast` is defined as follows:

```
def cast {α β : Sort u} (h : Eq α β) (a : α) : β :=
  h.rec a
```

Here, if two types α and β are equal, one can convert an object a with type α to type β. This is done through recursor of equality. Note that the equality itself, is defined as an inductive type:

```
inductive Eq : α → α → Prop where
  | refl (a : α) : Eq a a
```

It has only one constructor that takes an element and shows that it is equal with itself. For any inductive type, a recursor is automatically generated (for more information, check chapter 4 of [17]). In the case of `Eq`, it is:

```
#check Eq.rec
/-- Eq.rec.{u, u_1}
    {α : Sort u_1} {b : α}
    {motive : (a : α) → b = a → Sort u}
    (refl : motive b _) {c : α}
    (t : b = c) : motive c t --/
```

It may appear intimidating at first, but its role is simply to eliminate the equality in favor of an arbitrary type specified by the chosen motive. In the case of `cast`, the motive is `fun (b :  Sort u) (g :  a = b) => b`, which ignores the equality and returns the target type. In the definition of `cast`, `refl` is supplied by `a` (the proof of `a = a` is generated automatically for the motive), and `t` is instantiated with the provided equality h. The type checker accepts the proof of `tmulUnifyEquiv'` through

```
cast
 (h : ((⨂[R] (i₁ : ↑S), s ↑i₁) ⊗[R] (⨂[R] (i₂ : ↑(T \ S)), s ↑i₂)
 ≃ₗ[R] ⨂[R] (i : ↑(S ∪ T \ S)), s ↑i) =
 ((⨂[R] (i₁ : ↑S), s ↑i₁) ⊗[R] (⨂[R] (i₂ : ↑(T \ S)), s ↑i₂)
 ≃ₗ[R] ⨂[R] (i : ↑T), s ↑i))
 tmulUnionEquiv disjoint_sdiff_right
```

that produces the expected isomorphism. Note that `rw` tactic in this case is clever enough to prove `h` and use `cast` to finish the goal. However, `cast` makes reasoning about terms very complicated. For example, consider the following theorem:

```
theorem tmulUnionEquiv_tprod (hdisj : Disjoint S₁ S₂)
    (lv : (i : S₁) → s i) (rv : (i : S₂) → s i) :
    (tmulUnionEquiv hdisj)
        ((⨂ₜ[R] i : S₁, lv i) ⊗ₜ (⨂ₜ[R] i : S₂, rv i)) =
      ⨂ₜ[R] j : (S₁ ∪ S₂),
        if h : ↑j ∈ S₁ then lv ⟨j, h⟩ else rv ⟨j, by aesop⟩
```

It states what is the result of application of `tmulUnionEquiv` on tensor product of two product tensors. One would naturally expect the following theorem to hold:

```
theorem tmulUnifyEquiv_tprod' (hsub : S ⊆ T)
    (lv : (i : S) → V i) (rv : (i : ↑(T \ S)) → V i) :
    tmulUnifyEquiv' hsub ((⨂ₜ[R] i, lv i) ⊗ₜ (⨂ₜ[R] i, rv i)) =
    ⨂ₜ[R] i : T,
        if h : ↑i ∈ S then lv ⟨↑i, by aesop⟩ else rv ⟨↑i, by aesop⟩
```

However, `tmulUnionEquiv` is wrapped inside a `cast`, that is, `(cast _ tmulUnionEquiv _) ((⨂ₜ[R] i, lv i) ⊗ₜ (⨂ₜ[R] i, rv i))`, and there is no simple way to eliminate the `cast`, because the types on the two sides do not match. So proving `tmulUnifyEquiv_tprod'` becomes complicated, although it should follow immediately from `tmulUnionEquiv_tprod`. This issue can be avoided by refraining from changing the index set of the isomorphism through an equality and instead reindexing the tensors, as illustrated in 2.1. Note that `tmulUnifyEquiv_tprod` is provable for `tmulUnifyEquiv`, since after unfolding, the goal becomes `(reindex R _ _) ((tmulUnionEquiv _) ((⨂ₜ[R] i, lv i) ⊗ₜ (⨂ₜ[R] i, rv i))` which is simply application of composition of linear equivalences.

It is also helpful to explain how the use of `reindex` avoids the `cast` issue. As explained previously, `Equiv.subtypeEquivProp` establishes the equivalence between $(S \cup (T - S)) \approx T$ and this equivalence is consumed by `reindex` to produce the expected types. More precisely, $(S \cup (T - S)) \approx T$ is the equivalence between two `Subtypes`, that is, the type of elements that are members $S \cup (T - S))$ and the type of elements that are members of T. In Lean, `Subtypes` are defined as:

```
structure Subtype {α : Sort u} (p : α → Prop) where
  /-- The value in the underlying type that satisfies the predicate.-/
  val : α
  /-- The proof that `val` satisfies the predicate `p`.-/
  property : p val
```

In other words, a `Subtype` is a `Type` that isolates an object of an underlying `Type` from its additional properties. For example, `{ n : Nat // n % 2 = 0 }` is a `Subtype` of `Nat` that represents even numbers. We can create an object of a subtype by supplying a value and the proof that value satisfies the `Subtype` property:

```
let b : { n : Nat // n % 2 = 0 } := <2, Nat.mod_self>
```

Furthermore, every `PiTensorProduct` with an index set is actually indexed by a `Subtype`, namely a `Subtype` that represents the elements belonging to a specified set. In `tmulUnifyEquiv` 2.1, after `(tmulUnionEquiv (disjoint_sdiff_right)).trans` the goal becomes proving

```
    (⨂[R] (i : (S ∪ T \ S)), V i) ≃ₗ[R] ⨂[R] (i : T), V i}
```

However, the type `reindex` provides is

```
(⨂[R] (i : (S ∪ T \ S)), V i)
≃ₗ[R] ⨂[R] (i : T), V ((Equiv.subtypeEquivProp _).symm i)
```

One might expect the type checker to complain, since `((Equiv.subtypeEquivProp _).symm i` and `i` do not coincide syntactically. However, note that `V : ι → Type*` expects type ι, so automatically, `((Equiv.subtypeEquivProp _).symm i` which has the type of `Subtype T` is unpacked to its underlying value `i` by calling `Subtype.val`. Since all `reindex` is doing is changing a `Subtype` to another through an equivalence without modifying the underlying value, unpacking `((Equiv.subtypeEquivProp _).symm i` results in `i` itself that kernel expects. In other words, type casting in this case is avoidable through the isolation that `Subtypes` provide.

Chapter 5

Acknowledgments

I would like to express my deepest gratitude to Professor David Gross, whose guidance and insight have shaped every stage of this thesis. His intellectual rigor, critical feedback, and consistent support were indispensable in refining my research. Working under his supervision has been both a challenge and a privilege, and his standards have left a lasting impact on how I approach scientific inquiry.

I am also grateful to Lionel J. Dmello for our many conversations, particularly his thoughts on generelized probabilistic theorems, which proved helpful. I thank all my peers and colleagues who contributed through discussions, critiques, or simply by questioning assumptions—each contribution played a role in bringing this thesis to completion.

Bibliography

[1] A. Hulanicki & R. R. Phelps. Some applications of tensor products of partially-ordered linear spaces. *Journal of Functional Analysis*, (2):177–201, 1968.

[2] J. McCarthy. *Programs with common sense.* Stanford University, 1959.

[3] Deepmind. *AI achieves silver-medal standard solving International Mathematical Olympiad problems.* Google, 2024.

[4] R. P. Brent. Some instructive mathematical errors. *arXiv.*

[5] Docs. *Mathlib Documentation.*

[6] W. A. Howard. The formulae-as-types notion of construction. *Academic Press*, page 479–490, September 1980.

[7] M. H. Sørensen & P. Urzyczyn. Lectures on the curry-howard isomorphism. *Elsevier Science*, 149, 2006.

[8] Expressions in Lean. Lean meta programming.

[9] Docs. *Tactics.*

[10] A. Guichardet. *Infinite tensor product of C*-Algebras Part II. Infinite Tensor Products.* Aarhus Universitet, 1969.

[11] M.F. Atiyah & I.G. Macdonald. *Introduction to commutative algebra PP.32-33.* University of Oxford, 1969.

[12] Martin Plávala & Mário Ziman. Popescu-rohrlich box implementation in general probabilistic theory of processes. *Phys. Lett. A 384, 16, 126323 (2020)*, 2020.

[13] J. M. Borwein & A. S. Lewis. *Convex analysis and nonlinear optimization: Theory and examples.* CMS Books in Mathematics, 2006.

[14] I. Namioka & R. R. Phelps. Tensor products of compact convex sets. *Pacific Journal of Mathematics*, 31(2):469–480, 1969.

[15] Martin Plávala. General probabilistic theories: An introduction. *arXiv*, (2103.07469), 2021.

[16] L. Ligthart. *Semidefinite Programming Techniques for the Quantum Causal Compatibility Problem.* 2023.

[17] Docs. *Lean Lecture.*