Formalizing the Gelfand-Naimark-Segal Construction in Lean

Gregory Wickham

Lucas Bang, Advisor

Michael Orrison, Reader

Department of Mathematics

May, 2026

Abstract

I formalized the Gelfand-Naimark-Segal (GNS) construction using the proof assistant Lean. The GNS construction is a method for constructing a Hilbert space and a *-homomorphism from a C^*-algebra into the algebra of bounded operators on that Hilbert space. The GNS construction is an essential step in the proof the Gelfand-Naimark theorem, which is a foundational theorem in the theory of C^*-algebras. This formalization has been "peer-reviewed" in the sense in which Lean formalization projects are reviewed, meaning that it has been merged into Mathlib, the canonical, open-source library of mathematics in Lean.

Contents

Acknowledgments

I would like to thank my advisor, Prof. Lucas Bang, the math thesis director, Prof. Heather Zinn-Brooks, and my second reader, Prof. Michael Orrison, for their support and guidance while writing this thesis. I would also like to thank DruAnn Thomas and Melissa Hernandez-Alvarez for their support and thank Prof. Jireh Loreaux and Monica Omar for their extensive feedback and (where noted) direct contributions to this formalization project. Finally, I would like to thank Prof. Konrad Aguilar for teaching me about C^*-algebras in Math 138, and for his support for this project and my other work in C^*-algebras, and for providing feedback on the earlier drafts of this thesis.

Chapter 1

Introduction

Matrices are essential structures in mathematics because of their importance in linear algebra. Matrices can represent information about graphs, derivatives, stability, and countless other properties of a wide variety of mathematical structures, but fundamentally, a matrix is only a representation of a linear transformation on some vector space. Unfortunately, matrices can only be properly defined if they are operating on a finite-dimensional vector space. To extend the idea of matrices to infinite-dimensional vector spaces, we must turn to operator theory.

In operator theory, one might consider individual operators and study their properties, but what is perhaps more interesting is the study of operator algebras. The set of linear operators on some fixed vector space forms a vector space itself, because linear operators, just like matrices, can be added to one another and scaled by scalars from $\mathbb{R}$ or $\mathbb{C}$. The vector space that those linear operators form is also an algebra if we introduce composition of linear operators as a sort of multiplication-like operation. In the case where the linear operators are operating on, for example, $\mathbb{R}^n$, we know that the operators can be represented as matrices, and matrix multiplication is exactly composition of those operators.

The Gelfand-Naimark-Segal (GNS) construction deals with a particular class of operator algebras called C^*-algebras. In fact, the GNS construction is used to prove that all C^*-algebras are (up to isomorphism) operator algebras. The definition of a C^*-algebra makes no reference to operators or any vector space that they might act upon, so the fact that every C^*-algebra is an operator algebra is surprising, and requires a non-trivial proof. The theorem that states that all C^*-algebras are operator algebras is the Gelfand-Naimark theorem, and its proof uses the GNS construction to construct an

appropriate vector space for the operators to act upon and to construct an explicit isomorphism from the C^*-algebra to a subalgebra of the algebra of operators on that vector space (1).

This thesis will explain, in extensive detail, every step of the GNS construction, and it will do so in order to explain the newly formalized proof that it is presenting. A formalized proof is a proof of a theorem that is expressed in such a way that its correctness can be verified by a computer program. Typically, this means that the proof is written in some programming language that has a specialized kernel that only compiles programs that are both syntactically ***and logically*** correct (2).

The programming language used in this thesis to formalize the GNS construction is called Lean. There are many programming languages that can be used to write formal proofs, and such programming languages are often called proof assistants or interactive theorem provers. Other proof assistants like Isabelle/HOL (3), Metamath (4), and Rocq (5) also have significant user bases, but due to Lean's dedicated and well-funded team of developers and maintainers (6), the recent increase in popularity of generative AI research for mathematics (7), the creation of a unified open-source library of formalized mathematics in Lean (8), and the advocacy of some of its high-profile supporters like Terence Tao (9) and Kevin Buzzard (10), Lean has emerged as the go-to proof assistant for those interested in formalizing mathematics.

Lean's open source library of formally verified mathematics is called Mathlib (11) and is one of the main reasons that many mathematicians decide to use Lean when they formalize mathematics (8). As of March 1st, 2026, Lean contains 125,328 definitions and 260,371 theorems which have been contributed by 759 contributors (12). Consisting of over 2 million lines of code (12), Mathlib is the foundation upon which rest nearly all other mathematical formalization projects that use Lean (13).

For the purposes of this project to formalize the Gelfand-Naimark-Segal construction, the main mathematical structure with which we must work is the C^*-algebra.

1.1 C^*-algebras

A C^*-algebra can be thought of as a generalization of the algebra of square $n \times n$ matrices with complex entries, denoted $M_n(\mathbb{C})$. Fundamentally, it is often best to think of a C^*-algebra as a vector space with some additional

structure. Most of the examples and definitions presented in this section are from course notes prepared by Aguilar (14).

Definition 1.1.1. A vector space V over $\mathbb{C}$ is an **algebra over** $\mathbb{C}$ if there exists a map, $m : V \times V \to V$, that we call multiplication and denote $m(a, b) = ab$, which is

- associative, meaning $(ab)c = a(bc)$ for all $a, b, c \in V$, and
- linear over $\mathbb{C}$ in both coordinates, so that for all $r \in \mathbb{C}$, and $a, b, c \in V$
  - $r(ab) = (ra)b = a(rb) = rab$, and
  - $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.

Definition 1.1.2. A **subalgebra** is a subset of an algebra that is also an algebra over the same field and with respect to the same multiplication as the original algebra.

Example 1.1.3. For example, the vector space of polynomials of x with complex coefficients, $\mathbb{C}[x]$, is an algebra over $\mathbb{C}$, because it is a vector space with $\{1, x, x^2, x^3, \cdots\}$ as a basis, and the multiplication that makes it an algebra is the usual multiplication map which is associative and linear over scalars from $\mathbb{C}$.

Example 1.1.4. As another example, square $n \times n$ matrices with complex entries, $M_n(\mathbb{C})$ also form an algebra. $M_n(\mathbb{C})$ is a vector space over $\mathbb{C}$ with dimension n^2. A basis in the $n = 2$ case could be

$$\left\{ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\}.$$

The multiplication that makes $M_n(\mathbb{C})$ an algebra over $\mathbb{C}$ is the usual matrix multiplication, because it is associative and linear over scalars from $\mathbb{C}$.

Now we can define a C^*-algebra.

Definition 1.1.5. An algebra A that has a norm $\|\cdot\|_A$ and is closed with respect to that norm is a **C^*-algebra** if there exists a map $^* : A \to A$ called the **adjoint** such that for all $c \in \mathbb{C}$, and $a, b \in A$, the map *

- is conjugate linear: $(ca + b)^* = \overline{c}a^* + b^*$,
- is anti-multiplicative: $(ab)^* = b^*a^*$,

- is involutive: $(a^*)^* = a$, and
- satisfies the C^*-identity: $\|a\|_A^2 = \|a^*a\|_A$.

Definition 1.1.6. An element a of a C^*-algebra is called **self-adjoint** if $a^* = a$.

Definition 1.1.7. A C^*-algebra A is **unital** if it contains an element, denoted 1_A, for which $1_A a = a 1_A = a$ for all $a \in A$, $a \neq 0$.

Example 1.1.8. $M_n(\mathbb{C})$ is a C^*-algebra if we choose the right norm and adjoint operation. The appropriate adjoint operation is the conjugate transpose, often denoted † or sometimes just *. The appropriate norm is the operator norm, which in this case can be defined for all $M \in M_n(\mathbb{C})$ as $\|M\|_{op} = \sqrt{r(M^\dagger M)}$, where r denotes the spectral radius (greatest absolute value of the eigenvalues) and † denotes the conjugate transpose.

Example 1.1.9. As a consequence of the previous example, the complex numbers themselves are also a C^*-algebra. This is because $M_1(\mathbb{C})$ is simply the set of 1×1 matrices with a complex entry, which is isomorphic to the set of complex numbers themselves. In this $n = 1$ case, the adjoint function, which was the conjugate transpose, becomes simply the complex conjugate. So, we can define our norm for any $c \in \mathbb{C}$ as $\|c\| = \sqrt{|\overline{c}c|} = |c|$, where $|\cdot|$ is the typical modulus on $\mathbb{C}$.

Example 1.1.10. Another example of a C^*-algebra is the set of continuous functions on a closed real interval, $C([a,b]) = \{f : [a,b] \to \mathbb{C} \mid f \text{ is continuous}\}$. This set of functions, which is already an algebra because the functions can be scaled, added, and multiplied pointwise, becomes a C^*-algebra by defining the adjoint $f^*(x) = \overline{f(x)}$, and defining the norm $\|f\| = \sup\{|f(x)| : x \in [a,b]\}$.

Definition 1.1.11. Let A and B be C^*-algebras. The map $\pi : A \to B$ is a ***-homomorphism**, if, for all $c_1, c_2 \in \mathbb{C}$ and $a, b \in A$, π is

- linear: $\pi(c_1 a + c_2 b) = c_1 \pi(a) + c_2 \pi(b)$,
- multiplicative: $\pi(ab) = \pi(a)\pi(b)$, and
- *-preserving: $\pi(a^*) = \pi(a)^*$.

If $\pi(1_A) = 1_B$, then π is a **unital *-homomorphism**. If π is bijective, then π is a ***-isomorphism**. If there exists a *-isomorphism between A and B, then A and B are ***-isomorphic**, denoted $A \cong B$.

Example 1.1.12. An example of a unital *-homomorphism is

$$c \in \mathbb{C} \longmapsto \begin{pmatrix} c & 0 \\ 0 & c \end{pmatrix} \in M_2(\mathbb{C}),$$

because the identity of $\mathbb{C}$ maps to the identity of $M_2(\mathbb{C})$.

Example 1.1.13. An example of a non-unital *-homomorphism is

$$c \in \mathbb{C} \longmapsto \begin{pmatrix} c & 0 \\ 0 & 0 \end{pmatrix} \in M_2(\mathbb{C}),$$

because the identity of $\mathbb{C}$ does not map to the identity of $M_2(\mathbb{C})$. In fact, the identity of $M_2(\mathbb{C})$ is not in the range of the *-homomorphism at all.

Example 1.1.14. Another example of a non-unital *-homomorphism is

$$c \in \mathbb{C} \longmapsto -c \in \mathbb{C},$$

because the identity 1 is mapped to -1, which is not the identity. In this case, the identity is in the range of the *-homomorphism however, because -1 maps to 1.

Definition 1.1.15. The **dual space** of a A, denoted A', is defined

$$A' = \{f : A \to \mathbb{C} \mid f \text{ is continuous and linear}\}.$$

Definition 1.1.16. A (continuous) **linear functional** on A is an element of A'. That is, a linear functional $f : A \to \mathbb{C}$ is a function that is continuous and linear.

1.2 The Gelfand-Naimark-Segal Construction

To state the GNS construction as a theorem, we must first define Hilbert spaces and the bounded operators upon those Hilbert spaces.

Definition 1.2.1. An **semi-inner product** over a vector space V is a mapping $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{C}$ which, for all $c_1, c_2 \in \mathbb{C}$ and $x, y, z \in V$ is

- conjugate symmetric: $\langle x, y \rangle = \overline{\langle y, x \rangle}$
- conjugate linear in its first coordinate: $\langle c_1 x + c_2 y, z \rangle = \overline{c_1}\langle x, z \rangle + \overline{c_2}\langle y, z \rangle$

- linear in its second coordinate: $\langle x, c_1 y + c_2 z\rangle = c_1\langle x, y\rangle + c_2\langle x, z\rangle$

Definition 1.2.2. An **inner product** is a semi-inner product which has the additional property that $\langle\cdot,\cdot\rangle$ is positive definite, meaning if $x \neq 0$ then $\langle x, x\rangle > 0$, and $\langle 0, 0\rangle = 0$.

Note: Some sources may use an alternate convention in which the inner product/semi-inner product is linear in the first coordinate and conjugate linear in the second coordinate. The convention used in this thesis is chosen in order to be consistent with the convention used in Mathlib (15).

Definition 1.2.3. A vector space with a semi-inner product is called a **pre-inner product space**.

Definition 1.2.4. A vector space with an inner product is called an **inner product space**.

Definition 1.2.5. The (semi-)norm induced by an inner product is defined

$$\|x\| = \sqrt{\langle x, x\rangle}.$$

If $\langle\cdot,\cdot\rangle$, is a semi-inner product, it induces a semi-norm. If $\langle\cdot,\cdot\rangle$ is an inner product, it induces a norm.

Definition 1.2.6. An inner product space H is a **Hilbert space** if it is closed with respect to its inner product-induced norm.

Example 1.2.7. An example of a Hilbert space is $\mathbb{R}^N$, where the inner product is the dot product. Then, the inner product-induced norm is the usual Euclidean norm on $\mathbb{R}^N$.

Definition 1.2.8. Let V and W be vector spaces. Then, a map $T : V \to W$ is a **linear operator** if T is a linear function.

Definition 1.2.9. The set of **bounded (linear) operators** on a Hilbert space H, denoted $B(H)$, is the set of continuous linear operators from H to H. That is,

$$B(H) = \{T : H \to H \mid T \text{ is linear and continuous}\}.$$

Note: When describing a linear operator, the terms bounded, continuous, and uniformly continuous are all equivalent (16).

A special kind of linear operator that we will use extensively is the positive linear functional, but to define a positive linear functional, we must first define a positive element of a C^*-algebra.

Definition 1.2.10. Let A be a C^*-algebra. An element $a \in A$ is **positive** if $a = b^*b$ for some $b \in A$.

Example 1.2.11. The positive elements of the C^*-algebra $\mathbb{C}$ are the positive real numbers. This is because for any $c \in \mathbb{C}$, $c^*c = \overline{c}c$ is a positive real number, and any positive real number $r \in \mathbb{R}$ can be expressed in the form $r = \sqrt{r}\sqrt{r} = \overline{\sqrt{r}}\sqrt{r}$ and $\sqrt{r} \in \mathbb{C}$.

Prop 1.2.12. Positive elements of a C^*-algebra are self-adjoint.

Proof. If $a \in A$ is positive, $a = b^*b$ for some $b \in A$. This means $a^* = (b^*b)^*$, so by the properties of the adjoint, $a^* = b^*b = a$. □

Notation 1.2.13. For a self-adjoint $a \in A$, if a is positive we write $0 \leq a$, and for some $a, b \in A$, $a \leq b$ means $0 \leq b - a$.

Now we can define a positive linear functional.

Definition 1.2.14. A linear functional (element of A' (1.1.15)) is **positive** if $0 \leq a$ implies $0 \leq f(a)$. This can equivalently be stated as the condition that for self-adjoint $a, b \in A$, $a \leq b$ implies $f(a) \leq f(b)$.

The alternative definition that $a \leq b \implies f(a) \leq f(b)$, means that a positive linear functional is an order homomorphism. That is, it is an order-preserving map.

Theorem 1.2.15. Positive linear functionals are *-preserving. That is, for all $a \in A$, $f(a^*) = \overline{f(a)}$.

In Mathlib, this theorem is `PositiveLinearMap.instStarHomClass OfLinearMapClassComplexOfOrderHomClass`. See Proposition 9.5 in (17) for a proof of this fact.

Definition 1.2.16. Let H be a Hilbert space with inner product $\langle \cdot, \cdot \rangle$, and let $T \in B(H)$. Then, the **adjoint** of T, denoted T^*, is the unique element of $B(H)$ for which

$$\langle T(x), y \rangle = \langle x, T^*(y) \rangle$$

for all $x, y \in H$.

See any standard reference that discusses Hilbert spaces, such as (16), for a proof of the existence and uniqueness of such an adjoint.

Prop 1.2.17. With the adjoint defined as above, $B(H)$ is a $*$-algebra with function composition serving as multiplication and the adjoint serving as the * operation.

With these definition established, we can now state the GNS construction as a theorem.

Theorem 1.2.18 (The GNS Construction)**.** Let A be a C^*-algebra. Let $f \in A'$ (1.1.15) be positive.

There exists a pre-inner product space (1.2.3) A_f, whose elements are the elements of A, and whose semi-inner product (1.2.1) is defined

$$\langle x, y \rangle_f = f(x^* y).$$

There is also a function $\pi_f : A \to B(A_f)$ defined by

$$\pi_f(a)(x) = ax,$$

where $a \in A$ and $x \in A_f$.

Then, if we define H_f to be the Hausdorff completion (3.1.7) of A_f with respect to its semi-inner product-induced semi-norm (1.2.5), H_f is a Hilbert space (1.2.6), and there exists a *-homomorphism (1.1.11) $\overline{\pi}_f(a) : H_f \to H_f$ that is the completion of $\pi_f(a) : A_f \to A_f$.

It is important to note that this is not the standard statement of the GNS construction. The version stated here is consistent with the version I implemented and which is in Mathlib, and it is fully equivalent to the traditional definition. The differences between my/Mathlib's implementation and the standard version are discussed in Section 4.1.

Also note that there is typically a additional statement included as part of the GNS construction that states that there exists an $x_f \in H_f$ such that $\|x_f\|^2_{H_f} = \|f\|$ and $f(a) = \langle \pi_f(a)(x_f), x_f \rangle_f$ for all $a \in A$. This part of the GNS construction is important because it provides a key property that makes the GNS construction useful in the proof of the Gelfand-Naimark theorem, but due to time constraints, I have not yet formalized this statement, so I have not included it as part of this theorem.

1.2.1 Uses of the GNS Construction

The most notable use of the GNS construction is in the proof of the Gelfand-Naimark theorem, as that is the context in which it was originally developed (18). The Gelfand-Naimark theorem states that every C^*-algebra is

*-isomorphic to an algebra of bounded operators on some Hilbert space. To prove the Gelfand-Naimark theorem, such a Hilbert space can be constructed as the direct sum of all H_f where the f range over a set of linear functionals generated in a particular way by the self-adjoint elements of A. We denote this Hilbert space H_A. Then, the direct sum of the corresponding π_f, which we may call π_A, becomes an injective *-homomorphism so that A is isomorphic to $\pi_A(A)$, which is a subalgebra of the algebra of bounded operators on H_A.

The Gelfand-Naimark theorem is a theorem so essential to the study of C^*-algebras that it is often invoked implicitly to justify treating a C^*-algebra as an operator algebra, but the GNS construction is also frequently used independently of its relevance to the Gelfand-Naimark theorem. This is because the GNS construction provides an easy way to construct representations of C^*-algebra which can be used for various purposes depending on the context. For example, the GNS construction is often invoked in the field of noncommutative geometry, in which representations of C^*-algebras and operators related to those representations are often central objects of study (19).

Further, as Segal notes in section 6 of his original paper (20), a consequence of the GNS construction is that an algebra of quantum states is generated by its observables. This extended a prior result by von Neumann (21) to a more general setting, and now the GNS construction and GNS representations (and their generalizations) are have become a useful tool in algebraic quantum field theory (22). For an undergraduate-level overview of the use of C^*-algebras in quantum theory, see (23). The GNS construction is an important theorem in quantum theory

1.3 Mathematical Formalization in Lean

In this thesis I formalize the Gelfand-Naimark-Segal construction in Lean. Lean is a type of programming language called a proof assistant or interactive theorem prover, because it verifies the correctness of proofs. The process of writing a proof in Lean so that it can be formally verified is called formalization.

In "Why formalize mathematics?" (24), Massot asserts five main reasons that people formalize mathematics:

Correctness

The first is that it confirms correctness of results. Where conventional mathematical proofs rely on the understanding of a human reader to interpret and be convinced by a proof, formal proofs are machine verified by a program that will only accept a proof that is 100% correct and complete. This completely eliminates any potential issues from missing edge cases, references to inaccessible or incomprehensible literature, hand-waving explanations assuming prior reader familiarity, and proofs being too large or complex for one person to reasonably verify themselves.

Explainability

The second reason is that formal proofs can help people to explain their mathematics. Formal proofs have the potential to enable readers of proofs to control the level of detail at which they attempt to understand the proof. Whereas mathematics papers are typically written for a small audience already familiar with most of the relevant background material and thus often exclude explicit references to such material, a formal proof in a proof assistant like Lean can always explicitly identify each underlying result it uses. This allows readers with less expertise and immersion in the field to trace any chain of reasoning to a point of familiarity upon which they can build their new understanding.

Teaching

The third reason is that proof assistants could help to teach proof-writing to mathematics students. A proper discussion of this is far beyond the scope of this thesis however, so I recommend referring to "Proof Assistants for Teaching: a Survey" (25) for a perhaps unduly optimistic introduction to the topic.

Creating Mathematics

The fourth reason is that proof assistants can help researchers to create new mathematics. Proof assistants can help to provide confidence in intermediate results that a researcher may be questioning. It can also help with making iterative modifications to a theorem or its proof by providing instant feedback on what fails if a theorem statement or proof is slightly modified. Throughout my work, I have made significant use of that particular feature.

It allows me to modify results with complete confidence and without having to re-write an entirely new proof. Proof assistants also have the potential to help prove new theorems, either by searching a database of already formalized results that may be able to prove all or part of a desired result, or by applying some AI system to try to generate a proof for a statement.

Collaboration

The final reason is that formal proofs facilitate collaboration. Having multiple people working on different parts of a proof is easier when each part is formally specified and can be verified with out any additional work from any other person. It is also easier to subdivide and delegate tasks in a formal proof to be worked on in parallel and modified without risk of impacting the work of others. Some very large projects are currently in progress or have been completed in Lean, demonstrating its potential to facilitate collaboration. See the project to formalize Fermat's Last Theorem (26) with 58 contributors so far (27) or the Liquid Tensor Experiment (28) with 29 contributors (29). Lean-specific tools like Blueprint have even been created to help facilitate collaboration on large Lean formalization projects (30).

1.3.1 Formal Mathematics for Creating Math

Search Engines for Formal Mathematics

As argued by Thum (31), who was the last HMC math thesis student to work with Lean, formalization can make it easier for mathematicians to search for a proof of what they believe to be a known result. Currently, the tactic `exact?` in Lean tries to prove one's specified statement by searching for a theorem in available files and libraries that will yield your desired result as a direct consequence.

More sophisticated search engines have also been developed to help find results that have already been formalized. For example LeanDex (32) and LeanExplore (33), Lean State Search (34), Lean Search (35), Loogle (36), and the Mathlib documentation search (37) all allow one to search for formalized results, using, respectively, LLM-enhanced natural language search, proof state-based search, natural language search, pattern-matching of exact statement/theorem name, and exact search of theorem/statement name.

For example, searching using LeanSearch with the query "a complex number that is its own conjugate is real" produces results like

```
theorem Complex.conj_ofReal (r : ℝ) : conj (r : ℂ) = r
```

and

```
theorem Complex.conj_eq_iff_real{z : ℂ} :
(starRingEnd ℂ) z = z ↔ ∃ (r : ℝ), z = ↑r
```

among other, similar theorems that state similar results in different ways. Ignore the syntax for now. Lean's syntax, structure, and theory will be explained in more detail in Chapter 2.

There is also the possibility that the usefulness of Lean-based search engines could eventually extend beyond already-formalized mathematics. As Thum (31) wrote, "Were we to formalize the statements of the results of, say, every paper on arXiv.org [...], a Lean-enabled search engine could search for your desired result based on its actual mathematical content rather than the particular language used to describe it." Such an initiative is currently underway, being led by Kevin Buzzard and funded by Renaissance Philanthropy's AI for Math Fund, and will likely encourage the formalization of, at the very least, the definitions essential to the selected theorems that are being proven by modern research mathematicians (38).

Generative AI for Formal Mathematics

The use of AI for theorem proving is beyond the scope of this thesis, but recent notable successes of AI models generating formal proofs in Lean have captured that attention and imagination of many, so for further information I suggest referring to (39) for information on Google's DeepSeek, (40) for information on Harmonic's Aristotle, and (41) for information on Lean copilot.

I have used Harmonic's Aristotle model. After formalizing the proofs of some of the first theorems and lemmas for this project, I created a duplicate file that contained the theorem statements but not the proofs. Aristotle is designed to fill in these missing proofs, and it was able to do so in about an hour. For comparison, I had spent at least 40 hours on formalizing those proofs. However, almost all of the proofs that it produced were either the same length as mine or longer. They were also more difficult to understand, and for the theorem which had the most complicated proof and which had taken the majority of my time on that section of the project, the proof it produced could not be verified in a Lean environment with the default settings. That is, the proof took too long to compile. So, by the standards typically used to verify Lean proofs, it was unsuccessful.

Because Aristotle's proofs are longer and more difficult to interpret and sometimes do not compile, I am not using any Aristotle-produced code in my codebase nor do I include any other AI-generated code.

1.4 Reading this Thesis

We have now defined some of the essential mathematical concepts that we will need in order to understand the GNS construction, and we have established the motivation behind formalization in general and behind formalizing the GNS construction in particular. The next chapter will explain the basics of how to mentally parse Lean code so that you can understand the structure of the code snippets throughout this thesis. This text will aim to explain all of the code as comprehensively as is feasible, but if you wish to truly follow along with the code, rather than just the explanations of the code, you must install Lean yourself and download a copy of Mathlib to follow along with. This is because, as will be explained in Chapter 2, most of the information needed to understand Lean code can only be accessed when viewing the code in an appropriate development environment, typically Visual Studio Code with an extension for using Lean. That is to say, because Lean is an *interactive* theorem prover, one must interact with it to reap its full benefits. For instructions on how to install Lean locally, visit https://lean-lang.org/install/. To use Lean online, visit https://live.lean-lang.org/.

Throughout this thesis, I will frequently refer to theorems, definitions, tactics, and files in Mathlib without explicit citation. This is because it would be incredibly unwieldy to cite what would amount to every word in every code snippet. Additionally, Mathlib is an active project which is updated and changed frequently, so locations and names of theorems and definitions change regularly, which makes direct citations to particular results unstable. The code described in this thesis is all current as of Mathlib commit #e9ba370, although some if it, where noted, was only current as of an earlier commit, #7e1b73e. To find the current version of any code in Mathlib, visit https://leanprover-community.github.io/mathlib4_docs/Mathlib.html. In Mathlib, specific files are identified by what is effectively a path, but which uses a period instead of a slash. So, the Mathlib file located in the mathlib4 repository at "Mathlib/Analysis/CStarAlgebra /GelfandNaimarkSegal.lean", is written as "Mathlib.Analysis.CStarAlgebra.GelfandNaimarkSegal". This formatting is used because it is the same as the notation used in a Lean file to import a file or directory.

To find theorems, lemmas, and definitions in Mathlib when the file in which they reside is unspecified, you can visit the Mathlib documentation at https://leanprover-community.github.io/mathlib4_docs/Mathlib.html and paste the theorem name in the search bar at the top right corner of the page. Certain theorem names may return multiple results with different prefixes that indicate the namespace in which they reside, or the type of structure with which they deal. Sometimes you may be able to infer which result is the one you are looking for from context, but the only way to know for sure which theorem is being referenced is by opening the relevant code in Visual Studio Code and placing your cursor over the relevant term to show its full identifier and type.

Chapter 3 explains some background material about semilinearity and Hausdorff completions and explains the formalization of the fact that a continuous semilinear map between modules can be extended to a continuous semilinear map between the Hausdorff completions of the modules. That is, Chapter 3 explains the content of "Mathlib.Topology.Algebra.LinearMapCompletion".

Chapter 4 explains the GNS construction and its formalization, up to defining the 'GNS space', H_f / `f.GNS`, as described in Theorem 1.2.18. Chapter 5 them completes the explanation of the GNS construction and its formalization, beginning by defining the map π_f / `leftMulMapPreGNS`, as described in Theorem 1.2.18. Together, Chapters 4 and 5 cover the content of "Mathlib.Analysis.CStarAlgebra.GelfandNaimarkSegal".

Chapters 4 and 5 are both structured to begin with a strictly mathematical overview of their content, which does not contain significant Lean code, but which does explain the relevant mathematics informally in extensive detail. In both of those chapters, that first section is then followed by an in-depth explanation of the formalization of that mathematics, as it exists/existed in Mathlib. Each chapter then concludes with a brief discussion of some of the design choices made in the formalization and other comments on the formalization process.

In this thesis there are many mathematical structures and functions which are referred to via different names depending on whether they are being referenced informally or in Lean code. I try to make the correspondences clear in the text, but Appendix A also contains a glossary for easier reference.

Chapter 2

Lean

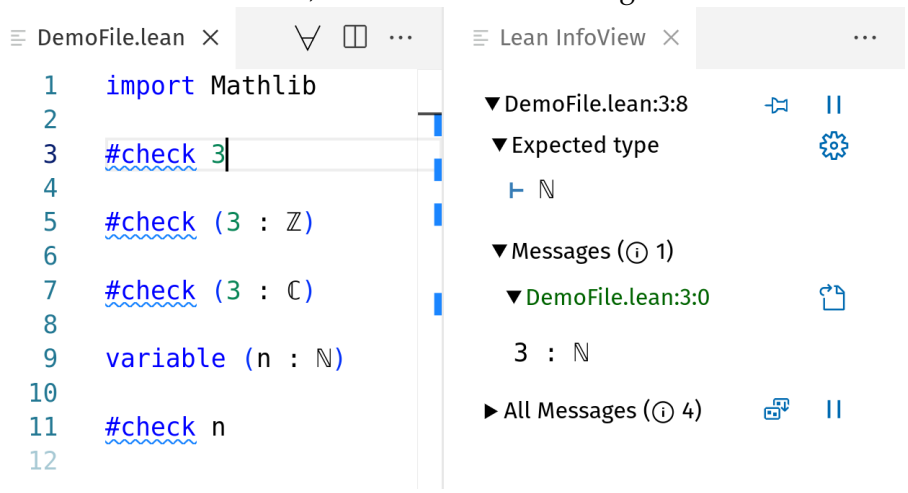
2.1 Structure & Syntax

2.1.1 Types

Lean is based on dependent type theory (42). What this means in practice is that every expression has type. For example, the expression `3` might have the type `ℕ`, meaning that it is a natural number.

In Lean, users can check the type of an expression using the `#check` command, so when we write `#check 3` and place our cursor at the end of the line, the Lean InfoView, under where it says "Messages," tells us `3 : ℕ`, which we read as "3 is of type ℕ", "3 has type ℕ", or in this particular case "3 is a natural number".

In Visual Studio Code, this will look something like:



This syntax that has an expression before the colon and has its type after the colon is ubiquitous in Lean, and will be used frequently in this thesis. It can also be used to specify the type of a value, so we could also write, for example, `(3 : ℤ)` to specify that we are referring to 3 as an integer, or we could write `(3 : ℂ)` to specify that it is a complex number.

We can specify a variable of a certain type without knowing its exact value. For example, if we want to fix an arbitrary natural number called `n`, we can write

```
variable (n : ℕ)
```

and we can check that, just like the number 3, the type of `n` is $\mathbb{N}$. That is, when we write `#check n`, Lean tells us `n : ℕ` (`n` has type $\mathbb{N}$).

This shows why types can generally be thought of as sets of all expressions of a certain form. When we write `n : ℕ`, we are essentially saying $n \in \mathbb{N}$.

2.1.2 Functions and Implications

Functions also have types. For example, a function g that takes a complex number and returns a complex number can be written `g : ℂ → ℂ` and we can define some arbitrary complex number c as `c : ℂ`. We can declare these variables by writing

```
variable (g : ℂ → ℂ) (c : ℂ)
```

and if we like, we can verify that they have the specified types by using the `#check` command.

The important thing to observe with these definitions is what happens when we want to evaluate $g(c)$. We would expect that $g(c)$ will be a complex number because g maps a complex number to a complex number. This is precisely what we see in Lean. That is, when we write

```
#check g c
```

Lean tells us that `g c : ℂ`, which means $g(c) \in \mathbb{C}$, just as we would hope.

You can think of this as us passing a complex number to g so that the resulting expression is a complex number. This idea is core to all mathematics in Lean. In fact, this is exactly how logical implication works.

For example, consider the theorem `Int.lt_succ_self`. In the Mathlib documentation, we can see the theorem's full statement is

```
theorem Int.lt_succ_self (a : ℤ) : a < a.succ
```

Note that `.succ` is short for successor, which means the next integer, so `a.succ` is just another way of writing `a + 1`. The part of this theorem that comes after the last colon (`a < a.succ`) indicates the type that the term will have after what is specified to the left of the colon `(a : ℤ)` is provided.

This means that we could, for example, pass the number 26 to the theorem, by writing

```
#check Int.lt_succ_self 26
```

and the Lean InfoView will tell us

```
Int.lt_succ_self 26 : 26 < Int.succ 26
```

This means that we have constructed a term of the type `26 < 26 + 1`, which means we have proven that `26 < 26 + 1`.

If we now compare our theorem:

```
Int.lt_succ_self (a : ℤ) : a < a.succ
```

to the function we defined earlier, `g : ℂ → ℂ`, we will see that they act very similarly. That is, they both have a type and when we pass a term of a certain pre-specified type, the result has a new type specified by the definition of the original term.

The key to understanding Lean lies in understanding that theorems and functions are really the same thing. The theorem

```
Int.lt_succ_self (a : ℤ) : a < a.succ
```

could just as easily be written

```
Int.lt_succ_self (a : ℤ) → a < a.succ
```

because it really is just a function that takes an integer and returns a term of the type `a < a.succ` where `a` is the original integer. The reason that the parameter is written in the form `(a : A)` is that specifying the name `a` for the input allows the value to be referenced later by the resulting output type.

Contrast this with `g : ℂ → ℂ`, where all we know is that it takes a complex number and returns a complex number. The number that we provide as input never receives a name, and the resulting output is just an arbitrary complex number that makes no reference to the input we provide.

Using both named inputs and →(arrow)-based inputs is common in theorems. For example, the theorem

```
theorem Complex.conj_eq_iff_im {z : ℂ} :
  (starRingEnd ℂ) z = z ↔ z.im = 0
```

states that, for some complex number z, $\overline{z} = z$ if and only if $\mathrm{Im}(z) = 0$. (The function `(starRingEnd ℂ)` denotes complex conjugation.)

We can see that before the type specification of the theorem, there is a `{z : ℂ}` to indicate that there is some complex number z that needs to be referenced by the output type. The use of curly brackets here instead of parentheses indicates that Lean will usually try to infer the value of z, although we can specify it explicitly. For example, if we write

```
#check Complex.conj_eq_iff_im (z := 7)
```

we are explicitly specifying the value of z as 7, and Lean shows us that

```
conj_eq_iff_im : (starRingEnd ℂ) 7 = 7 ↔ im 7 = 0
```

This means that the type of our expression is

```
(starRingEnd ℂ) 7 = 7 ↔ im 7 = 0
```

and so we have a proof that $\overline{7} = 7 \iff \mathrm{Im}(7) = 0$.

This type is actually two functions expressed as one, because the bi-directional `↔` indicates that we can use this logical implication in either direction. We can specify that we want to use the implication from left to right by adding the suffix `.mp` (for modus ponens) to the end of the expression, like so

```
#check (Complex.conj_eq_iff_im (z := 7)).mp
```

which produces a term of the type

```
(starRingEnd ℂ) 7 = 7 → im 7 = 0
```

which is a function that takes an input of type `(starRingEnd ℂ) 7 = 7` and returns an output of type `im 7 = 0`. Importantly, this arrow is also denoting logical implication, because it is telling us that `(starRingEnd ℂ) 7 = 7` implies `im 7 = 0`.

We can now use this result to complete a proof of the fact that `im 7 = 0`, by constructing a term of type (i.e. proving that) `(starRingEnd ℂ) 7 = 7`, which we can provide as the input to

```
(Complex.conj_eq_iff_im (z := 7)).mp
```

To construct a term of the type `(starRingEnd ℂ) 7 = 7`, we will use the theorem

```
theorem Complex.conj_ofReal (r : ℝ) :
  (starRingEnd ℂ) ↑r = ↑r
```

which when provided with an $r \in \mathbb{R}$ produces a proof that (term of type) $\overline{r} = r$. Note that the ↑s indicate the the real number r is being "coerced" (essentially converted) to a complex number.

When we write

```
Complex.conj_ofReal 7
```

we get a proof that (term of the type)

```
(starRingEnd ℂ) ↑7 = ↑7
```

We can then pass this result to our expression of type

```
(starRingEnd ℂ) 7 = 7 → im 7 = 0
```

which should produce a term of the type `im 7 = 0`. To do this, we write

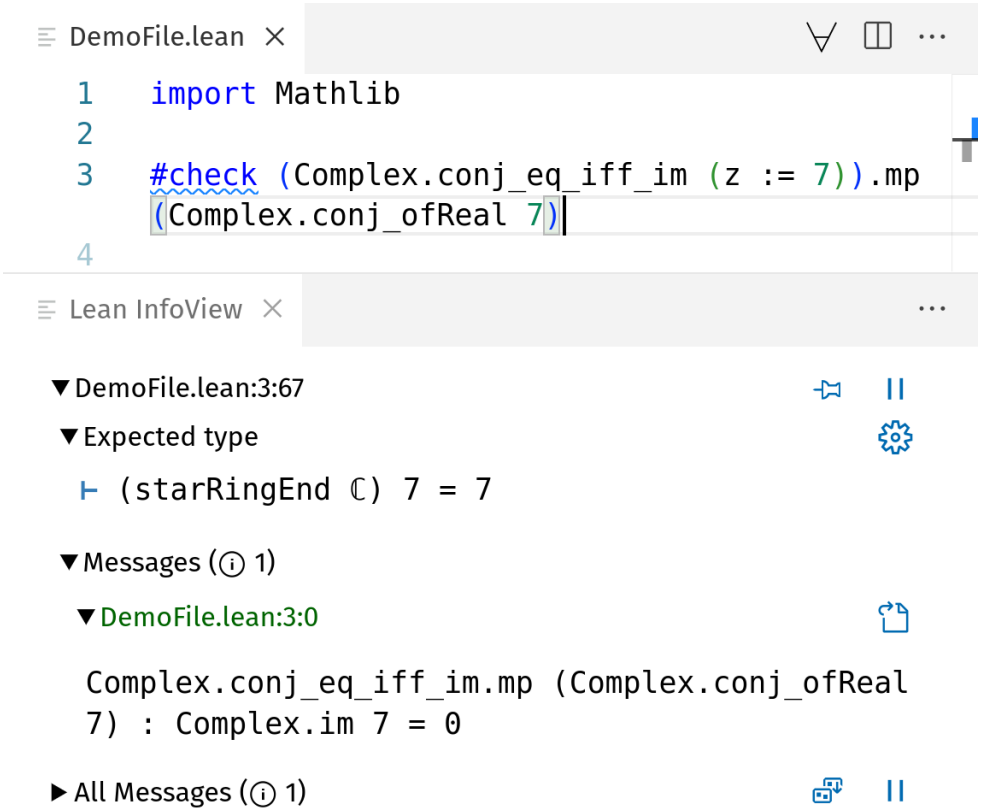
```
#check (Complex.conj_eq_iff_im (z := 7)).mp
    (Complex.conj_ofReal 7)
```

And the InfoView tells us that this term has the type

```
Complex.im 7 = 0
```

so we have proven that $\mathrm{Im}(7) = 0$.

At this point, our development environment should look something like the image below: (Note that the Lean InfoView is typically on the right by default. It is simply placed on the bottom in the screenshot below to conform the vertical format of the page.)



Thus, we have completed our formal proof that $\mathrm{Im}(7) = 0$.

To summarize what we have learned in this section:

1. Every expression has a type.
2. Passing an expression of the correct type to the right expression produces a new term of a new type.
3. We prove a statement by constructing a term whose type is the desired statement.
4. The `→` character denotes functions as well as logical implication, because in Lean there is no distinction between the two.
5. Notation of the form `(a : A)` specifies that an input value should have type `A` and can be referred to as `a`.
6. Notation of the form `A →` indicates that an input should have type `A`, but will not be explicitly referenced again.

2.1.3 Definitions, Theorems, and Lemmas

In the last section we used theorems that had particular types as though they were black boxes handed to us by the gods of Lean. However, each theorem, because it has a particular type, needed to be constructed from other terms in much the same way that we constructed our proof that $\text{Im}(7) = 0$. This is how Lean can ensure that proofs are logically correct: there is a small set of axioms that are declared to be true, and all other terms and theorems are built by using those axioms (and the other terms built using those same axioms) to construct theorems that state our desired results (have the types we want).

For the purposes of this explanation, we will not explain the basic axioms in Lean, but we will show how a theorem is defined/proven.

We will call our theorem `Im7Is0`, and it will have type `im 7 = 0`. We can define this theorem by writing

```
import Mathlib
open Complex

theorem Im7Is0 : im 7 = 0 :=
  (conj_eq_iff_im (z := 7)).mp (conj_ofReal 7)
```

We can the check that `Im7Is0 : im 7 = 0`, by writing

```
#check Im7Is0
```

This notation to define/prove a theorem has 4 components:

1. The **`theorem`** keyword indicates that we are about to define a named theorem
2. The name `Im7Is0` assigns an identifier to this theorem that we can use to refer to it later.
3. The `im 7 = 0` specifies the type of the theorem, which is the statement that it proves.
4. The part after the `:=` is the definition, which uses the term we constructed in the prior section to construct a term of the type `im 7 = 0`, as specified before the `:=`.

We can see that this allows us to use a single, short name to refer to a result by abstracting away all of the steps we took to produce a term of

the type `im 7 = 0`. Once we construct a term of a particular type, we can name it by making it a theorem, and conveniently refer back to it and reuse it, without worrying about how it was constructed. This is called proof irrelevance (42) and it is part of what makes Lean so useful: as long as a term has a particular type, the steps taken to produce that type are irrelevant and will have no bearing on how the term is used.

We could have equivalently written

```
lemma Im7Is0 : im 7 = 0 :=
  (conj_eq_iff_im (z := 7)).mp (conj_ofReal 7)
```

because in Lean, there is no difference between the keywords **`theorem`** and **`lemma`**. They are both provided simply for the convenience of those accustomed to the mathematical convention of designating more minor theorems as lemmas.

We can define functions in a similar way. For example, we can define the function `plusTwo` by writing

```
def plusTwo (n : ℤ) := n + 2
```

We can then see that `#check plusTwo` tells us that

```
plusTwo (n : ℤ) : ℤ
```

and when we apply the function to an actual number, by writing

```
#check plusTwo 17
```

we see that `plusTwo 17 : ℤ`, as we would expect. We can also see that if we write

```
#check (plusTwo : ℤ → ℤ)
```

we get that `plusTwo : ℤ → ℤ`. What we have done here is simply use a different notation to express the type of `plusTwo`. Because `plusTwo` is defined with a named input, `n : ℤ`, Lean will express its type as `plusTwo (n : ℤ) : ℤ` by default. But, as shown by writing `#check (plusTwo : ℤ → ℤ)`, it is also entirely valid to write the type of `plusTwo` as `ℤ → ℤ` because it is a function that takes and integer and returns an integer.

If we wanted to make the output type of `plusTwo` extra explicit, we could have written

```
def plusTwo (n : ℤ) : ℤ := n + 2
```

which specifically indicates that the output of `plusTwo` is an integer. At this point, you may notice that this definition looks very similar to the way we define theorems.

And that is because the only difference between a theorem and a definition is that the output type of a theorem must be a proposition, that is, an assertion that some fact is true. That is to say, a theorem/lemma is a special case of a definition, where the output type is a proposition and must be stated explicitly.

We can see that theorems/lemmas can be expressed as definitions, by returning to `Im7Is0`. We could just as easily have written

```
def Im7Is0 : im 7 = 0 :=
    (conj_eq_iff_im (z := 7)).mp (conj_ofReal 7)
```

and the fact that Lean accepts this statement and `#check Im7Is0` confirms that `Im7Is0` has type `im 7 = 0`, shows that this is fully equivalent to the version of this statement that had **`theorem`** in place of **`def`**.

Another difference between the **`def`** keyword and **`theorem`**/**`lemma`** is that the output type must be stated explicitly. That is why we could also write

```
def Im7Is0 :=
    (conj_eq_iff_im (z := 7)).mp (conj_ofReal 7)
```

which doesn't explicitly state `im 7 = 0` in the statement, but Lean can infer the type from the type of the term that is constructed after the `:=`. In this definition, we could not change **`def`** to **`theorem`** without adding back the explicit type annotation.

At the top level of a program, **`def`**, **`theorem`**, and **`lemma`** are the keywords that can be used to define new expressions.

2.2 Tactics

Although direct composition of theorems to produce an expression of one's desired type is a logically and computationally sound way to prove a statement, it is often impractical to explicitly construct those so-called "proof terms" in that way. Instead, Lean programmers often use other programs called "tactics" that combine theorems in the background to construct a proof, while allowing the person writing the code to write in a way that is much more natural to both programmers and mathematicians.

To use tactics to prove a theorem, one must enter tactic mode by using the keyword **by** after the := that starts a definition. For example, we can prove Im7Is0 by writing

```
theorem Im7Is0 : im 7 = 0 := by
  simp
```

where simp is short for simplify, and is one of Mathlib's most powerful tactics. It accesses a huge library of theorems and automatically decides which ones to apply in order to prove the goal, which is specified as the type annotation between the : and the :=, which in this case is im 7 = 0 . We can see that Lean recognizes that our goal is to prove im 7 = 0, by placing our cursor at the end of the line with **by** on it. We can then look to the InfoView where it states "1 goal", followed by

```
⊢ im 7 = 0
```

We can actually investigate what simp is doing, by re-writing it to simp?. Doing this causes the InfoView to suggest

```
Try this:
  [apply] simp only [im_ofNat]
```

When we click [apply], it replaces our simp? with the suggested content: simp only [im_ofNat], which only uses the theorem im_ofNat, which states that the imaginary part of a natural number is 0, which is a sufficient theorem to prove that $\mathrm{Im}(7) = 0$, because 7 is a natural number.

2.2.1 Changing the Goal

The main purpose of tactic mode is to modify the goal into a statement which can be proven to be true immediately by some theorem. In the case of proving im 7 = 0, that theorem could be immediately identified as im_ofNat by the simp tactic, but not all proofs are so simple.

Suppose instead, that we wanted to prove that $2+3 = 3+(2+0)$. Again, the simp tactic could easily prove this, invoking only two simple theorems, but we will demonstrate other common uses of tactic mode by ignoring the simp tactic and instead using the tactics rewrite and rfl to prove this goal.

To prove that goal using only the tactics rewrite and rfl (which is short for reflexivity) we could write

```
theorem twoThree : 2 + 3 = 3 + (2 + 0) := by
  rewrite [add_zero, add_comm]
  rfl
```

To see what these tactics do, first place your cursor at the end of the first line, right after `by`. You will see that the InfoView states 1 goal:

```
⊢ 2 + 3 = 3 + (2 + 0)
```

If you now move your cursor to the beginning of `add_zero` (right before the `a`), the `(2 + 0)` in the goal will be highlighted red, which indicates that `add_zero` is about to change that part of the expression. If you then move your cursor to the end of `add_zero`, right before the comma, the `(2 + 0)` will be replaced by a `2`. Thus, the new goal is to prove

```
⊢ 2 + 3 = 3 + 2
```

Investigating `add_comm` in the same way, we see that it re-writes the goal to

```
⊢ 3 + 2 = 3 + 2
```

Because we now have exactly the same expression on both sides of the equals sign, we have a statement that must be true by the reflexivity of equality. That means, we can complete the proof with the tactic, `rfl`. We can see that `rfl` completes the proof by placing the cursor at the end of `rfl`, and seeing that the InfoView tells us that our tactic state is "No goals".

In real Lean proofs, the tactic `rewrite` is almost never used. Instead we use the tactic `rw`, which is an abbreviation of rewrite, but which also automatically tries to use `rfl` when it is done rewriting the goal. Changing `rewrite` to `rw` in this proof allows it to be shortened to

```
theorem twoThree : 2 + 3 = 3 + (2 + 0) := by
  rw [add_zero, add_comm]
```

The are many other tactics and ways of manipulating the goal and the information we have available to help prove the goal, but all other tactics will be explained as they are introduced later on in this thesis. If you wish to see a list of all tactics in Lean/Mathlib, see the Tactic Reference of the Lean Language Reference (43).

2.3 Summary

In this chapter, we have learned that, in Lean, every expression has a type. Everything is an expression with a type. The type of a number might be `ℝ` or `ℤ`. The type of a theorem specifies its mathematical content. For example, the theorem `Im7Is0` has the type `im 7 = 0`. This is why, to prove a certain mathematical statement, we construct a term whose type is that statement.

We also learned to how to use existing terms with certain types to construct new terms with new types. The methods and syntax involved are explained in section 2.1 and summarized at the end of section 2.1.2.

We then saw in section 2.2 that tactics provide a more convenient way to prove statements (i.e. construct a term of a certain type). In particular, we learned how to use the tactic `rw` to re-write our goal (the statement we are trying to prove). We saw that we can complete proofs by re-writing our goal (if it is an equation) so that both sides of the equation are identical.

The methods and tactics introduced in this chapter are provided so that you can get a better sense of the basic mechanics of using Lean. In later chapters, many new tactics and methods will be introduced, and they will be explained as they become relevant. Most of those explanations will assume familiarity with the content of this chapter.

Chapter 3

Lifting Continuous Semilinear Maps

3.1 Mathematics & Formalization

Before we get into the details of the formalization of the GNS construction that we contributed to Mathlib, there is an important concept that must be explained first. It is a definition which occupies its own file in Mathlib that we added as part of our formalization of the GNS construction, but which was sufficiently general that it was placed in a different location from the rest of the code.

To understand this definition, we must first define semilinear maps and Hausdorff completions.

Definition 3.1.1. Let A and B be vector spaces (or more generally modules) with ring (or more generally semi-ring) of scalars R_A and R_B respectively. Let $\sigma : R_A \to R_B$ be a ring (or semi-ring) homomorphism. We call a map $f : A \to B$ **σ-semilinear** if for all $x, y \in A$ and $c \in R_A$

- $f(x + y) = f(x) + f(y)$, and
- $f(cx) = \sigma(c) f(x)$.

Definition 3.1.2. If A and B are vector spaces over $\mathbb{C}$, and σ is complex conjugation, then a σ-semilinear $f : A \to B$ is called **conjugate-linear**.

Notation 3.1.3. When using Mathlib, continuous σ-semilinear maps from A to B have type `A →SL[σ] B`.

Definition 3.1.4. If f is σ-semilinear and σ is the identity function, then f is **linear**.

This means that linear and conjugate-linear maps are special cases of semilinear maps where σ is the identity map or complex conjugation, respectively.

Definition 3.1.5. The **separation quotient** (also called Kolmogorov quotient) of a topological space X is the set of equivalence classes of points of X, where two points are in the same equivalence class if they are inseparable (also called topologically indistinguishable).

Prop 3.1.6. The separation quotient of a topological space must be Hausdorff (also called separable, T_0).

Definition 3.1.7. The **Hausdorff completion** of a uniform space A is the separation quotient of the union of A with the limits of all of its Cauchy filters.

Example 3.1.8. A special case of the Hausdorff completion is the Cauchy completion of a metric space. This is because metric spaces are uniform spaces and Cauchy sequences are Cauchy filters. That is why the next two examples are examples of Hausdorff completions.

Example 3.1.9. The real numbers $\mathbb{R}$, are the Hausdorff completion of the rational numbers $\mathbb{Q}$ with respect to the usual topology induced by the absolute value metric.

Example 3.1.10. The p-adic numbers $\mathbb{Q}_p$, are the Hausdorff completion of the rational numbers $\mathbb{Q}$ with respect to the topology induced by the p-adic absolute value.

Now, we can understand the definition

```
ContinuousLinearMap.completion
```

This definition takes as input a continuous function $f : A \to B$ is σ-semilinear, and produces term whose type is a continuous function g that agrees with f for all inputs in A, but which is also a σ-semilinear map from the Hausdorff completion of A to the Hausdorff completion of B.

The main definition in Mathlib.Topology.Algebra.LinearMapCompletion is `ContinuousLinearMap.completion`. As we can see in the official Mathlib documentation, this definition's full signature is

```
def ContinuousLinearMap.completion {α : Type u_1}
    {β : Type u_2} {R₁ : Type u_3} {R₂ : Type u_4}
    [UniformSpace α] [AddCommGroup α]
    [IsUniformAddGroup α] [Semiring R₁] [Module R₁ α]
    [UniformContinuousConstSMul R₁ α] [Semiring R₂]
    [UniformSpace β] [AddCommGroup β]
    [IsUniformAddGroup β] [Module R₂ β]
    [UniformContinuousConstSMul R₂ β]
    {σ : R₁ →+* R₂} (f : α →SL[σ] β) :
  UniformSpace.Completion α →SL[σ]
    UniformSpace.Completion β
```

Because the only parameter in parentheses is `(f : α →SL[σ] β)`, we can see that if we pass some function `f` that is a semi-linear map from `α` to `β` to `ContinuousLinearMap.completion`, the result is a σ-semilinear map from the Hausdorff completion of `α` to the Hausdorff completion `β`. Because we opened the namespace `ContinuousLinearMap`, in which this definition is defined, we can refer to `ContinuousLinearMap.completion` as `completion`, so that is how we will refer to this definition for the rest of this section.

The various assumptions listed in the Mathlib documentation are the minimal assumptions sufficient to ensure that semi-linear maps can be defined between `α` and `β`, that the Hausdorff completions of `α` and `β` can be defined, and that the linear structure on `α` and `β` (i.e. the operations of addition and scalar multiplication) respect the underlying structure that enables the Hausdorff completion. Specifically

- `{α : Type u_1} {β : Type u_2} {R₁ : Type u_3} {R₂ : Type u_4}` defines four sets upon which structure is added by the subsequent assumptions.
- `[UniformSpace α]` and `[UniformSpace β]` ensure that the spaces `α` and `β` have the necessary topological properties to ensure that their Hausdorff completion can be defined.
- `[AddCommGroup α]` and `[AddCommGroup β]` ensure that `α` and `β` each have a commutative addition operation under which each is a group.
- `[IsUniformAddGroup α]` and `[IsUniformAddGroup β]` ensure that addition and negation on `α` and `β` are uniformly continuous with

respect to their structure as a `UniformSpace`. This is ensures that `α` and `β` are topological additive groups.

- `[Semiring R₁] [Module R₁ α]` specifies that `α` is a module with scalars from the semi-ring `R₁`. `[Semiring R₂] [Module R₂ β]` similarly specifies that `β` is a module with scalars from the semi-ring `R₂` In this thesis, the ring of scalars for our modules will always be the field of complex numbers $\mathbb{C}$, so our modules are vector spaces.
- By assuming a module structure, a scalar multiplication operation is inferred, and so the remaining assumptions, `[UniformContinuousConstSMul R₁ α]` and `[UniformContinuousConstSMul R₂ β]` specify that the scalar multiplication must also be uniformly continuous.

Now, we can tell that `(f : α →SL[σ] β)` is the only parameter that must be explicitly specified both because it is the only parameter listed in parentheses — rather than square or curly brackets — and because it is the only named parameter other than `σ`, and `σ` does not need to be specified explicitly because it can be inferred from `f`. When we pass `(f : α →SL[σ] β)` to `completion`, we can see from the Mathlib documentation and the definition in the source code that we obtain a term of the form

```
UniformSpace.Completion α →SL[σ]
  UniformSpace.Completion β
```

That is, it produces a continuous σ-semilinear function from the Hausdorff completion of `α` to the Hausdorff completion of `β`. The Lean kernel will infer `R₁, R₂, α, σ`, and `β` from whatever `f` it receives, and it will throw an error if it cannot verify that the that `R₁, R₂, α, σ`, and `β` satisfy all of the necessary assumptions.

Those familiar with category theory may prefer to think of `completion` as a functor. That is, it is a function that takes a morphism, in this case a σ-semilinear map from α to β, and it returns a new morphism, which in this case is another σ-semilinear map from the Hausdorff completion of α to the Hausdorff completion of β.

In order to define `completion (f : α →SL[σ] β)` as a continuous semilinear map from `Completion α` to `Completion β`, we must define a function from `Completion α` to `Completion β`, and only then can we prove that it is continuous and σ-semilinear. This requires four parts: defining a function f, proving that $f(x+y) = f(x) + f(y)$, proving that $f(cx) = \sigma(c)f(x)$ for a scalar c, and proving that f is continuous.

Those first two steps, defining a function and proving that it is an additive homomorphism can both be done explicitly, but there is also a way to do both concisely in a single line. This method was suggested by Monica Omar (44) and allows us to use the fact that Mathlib already has a way to extend additive monoid homomorphisms to the completions of the domain and codomain (a group is a kind of monoid). So, in place of defining a function and proving it is an additive homomorphism in separate lines, the first two lines of our definition become

```
noncomputable def completion (f : α →SL[σ] β) :
    Completion α →SL[σ] Completion β where
  __ := f.toAddMonoidHom.completion f.continuous
```

which specifies that we are defining our new continuous semilinear map to be the function that we get when we treat `f`, which is already a continuous additive homomorphism by definition, as a monoid homomorphism and lift it to the completions of its domain and codmain. The double underscore notation essentially tells Lean to fill in the missing fields with the corresponding fields from the structure that follows the `:=`.

The **`noncomputable`** flag before **`def`** is a mostly irrelevant designation that we will not explain because it is not relevant to any of the content of this thesis. See (43) Section 7.1 for more information on noncomputability in Lean.

That structure `f.toAddMonoidHom.completion f.continuous` can be broken down into the expression `AddMonoidHom.completion` and the parameters `f` and `f.continuous` that we pass to it. `AddMonoidHom.completion` does the same thing as the `completion` that we are defining, but only for the additive component of our map. That is, when we pass it an additive homomorphism, denoted `(f : α →+ β)` and a proof that that additive homomorphism is continuous, `(hf : Continuous ⇑f)`, it returns a new additive homomorphism `Completion α →+ Completion β` that is defined on the Hausdorff completions of its domain and codomain. Because a continuous semilinear map, `(f : α →SL[σ] β)` is an additive homomorphism that also respects scalar multiplication (meaning $f(cx) = \sigma(c)f(x)$), the only properties that we still need to prove are that the additive homomorphism produced by `AddMonoidHom.completion` does, in fact, respect scalar multiplication, and that it is continuous. Those two properties are called `map_smul'` and `cont`, respectively.

To complete proof that $f(rx) = \sigma(r)f(x)$, we write

```
map_smul' r x := by
```

```
induction x using Completion.induction_on with
| hp =>
  exact isClosed_eq
    (continuous_map.comp <| continuous_const_smul r)
    (continuous_map.const_smul _)
| ih x => simp [← Completion.coe_smul]
```

This proof is the first instance in which we use an induction principle to complete our proof. An induction principle is a theorem that allows us to prove that a proposition about a type is true by first proving that the proposition holds from some related type or element of that type and then proving that the proposition "extends" in the correct way to the main type for which we wish to prove the proposition holds. The induction principle we use here is `Completion.induction_on`, which lets us prove a proposition about the completion of α.

That is, because x is in the completion of α and not necessarily in α itself, we cannot directly use the definition of f to simplify. Instead, we have to re-write x as an element of α in such a way that proving $f(rx) = \sigma(r)f(x)$ for $x \in \alpha$ will imply that $f(rx) = \sigma(r)f(x)$ for any x in the completion of α. To do that, we use the following theorem:

Theorem 3.1.11. (`UniformSpace.Completion.induction_on`) Let $\overline{\alpha}$ be the Hausdorff completion of α, and let $P(a)$ be a proposition defined on all $a \in \overline{\alpha}$. If

- $\{a \in \overline{\alpha} : P(a)\}$ is closed in $\overline{\alpha}$, and
- for all $b \in \alpha$, $P(b)$,

then $P(a)$ for all $a \in \overline{\alpha}$.

In this case, the proposition that we want to prove is that $f(rx) = \sigma(r)f(x)$ for all `x : Completion α`, and we will prove that by showing that the set $\{a : f(ra) = \sigma(r)f(a)\}$ is closed in `Completion α` and that $f(rx) = \sigma(r)f(x)$ for all for all `x : α`.

To prove that $\{a : f(ra) = \sigma(r)f(a)\}$ is closed in `Completion α`, we use the following theorem:

Theorem 3.1.12. (`isClosed_eq`) Let X and Y be topological spaces, and let X be Hausdorff (T_2/separable). If $f : X \to Y$ and $g : X \to Y$ are continuous functions, then $\{y \in Y : f(y) = g(y)\}$ is a closed set.

For our purposes in this proof, this theorem tells us that we must prove that $a \mapsto f(ra)$ is continuous and that $a \mapsto \sigma(r)f(a)$ is continuous. This is where we use our assumptions involving `UniformContinuousConstSMul`. Because f is continuous by assumption, we can show that $a \mapsto f(ra)$ and $a \mapsto \sigma(r)f(a)$ are continuous by showing that they are both compositions of continuous functions, and to do that, we must show that out scalar multiplication is continuous, which is what `UniformContinuousConstSMul` guarantees.

This allows us to write our proof that $\{a : f(ra) = \sigma(r)f(a)\}$ is closed in a single statement:

```
hp =>
  exact isClosed_eq (continuous_map.comp
    <| continuous_const_smul r)
      (continuous_map.const_smul (σ r))
```

which uses that fact that scalar multiplication by a constant (`const_smul`) is continuous and that composing a continuous function with a continuous function produces a continuous function. In the version of this code that is in Mathlib, the `(σ r)` is replaced by a `_`, because lean can infer the correct term based on the proof state. The same could just as easily be done for `r`, but it would not make the line any shorter.

Now, to prove our "inductive hypothesis", that $f(rx) = \sigma(r)f(x)$ for all `x : α`, we can use Mathlib's `simp` tactic to automatically prove the goal. We only need to additionally inform it that it can use the lemma `Completion.coe_smul`, which states that

```
((c • x) : Completion α) = c • (x : Completion α)
```

We also want simp to use this equality to rewrite terms in the form seen on the right in the form of the term on the left, so we include a `←` in front of the theorem name, making the entire proof

```
    ih x => simp [← Completion.coe_smul]
```

If we want to better understand how Lean constructs this proof, we can change `simp` to `simp?` and apply the suggestion that Lean generates to see that it becomes:

```
simp only [← Completion.coe_smul,
ZeroHom.toFun_eq_coe,
  AddMonoidHom.toZeroHom_coe,
```

```
  AddMonoidHom.completion_coe,
  LinearMap.toAddMonoidHom_coe, LinearMap.map_smul_sl,
  coe_coe]
```

showing us all of the theorems that `simp` is using. If you would like to inspect each step of the proof individually, see Appendix B.2 for a version of this proof that can be run in VSCode or lean-lang.org and uses re-writes instead of simplification. Here I am omitting a step-by-step explanation of that proof because it consists almost entirely of type coercions which are largely irrelevant to the underlying mathematics and only truly make sense when viewing the code interactively in an appropriate code editor.

After we have proven `map_smul'`, all that remains is to prove continuity, and continuity can be proven directly by the theorem

```
UniformSpace.Completion.continuous_map
```

which states that a map between completions of `UniformSpaces` that is generated by raising a function between the `UniformSpaces` is always continuous. So, by writing

```
cont := continuous_map
```

we have completed our definition of `completion`.

Now, to ensure that `completion` can be used conveniently, there are three additional lemmas included in `LinearMapCompletion.lean`.

Their full code is:

```
@[simp]
lemma toAddMonoidHom_completion (f : α →SL[σ] β) :
    f.completion.toAddMonoidHom =
    f.toAddMonoidHom.completion f.continuous := rfl

lemma coe_completion (f : α →SL[σ] β) :
    f.completion = Completion.map f := rfl

@[simp]
theorem completion_apply_coe (f : α →SL[σ] β) (a : α)
:
    f.completion a = f a := by
  simp [coe_completion, map_coe]
```

The first two lemmas simply establish some equivalences that follow directly from the definitions, hence the proofs by `rfl` (a tactic that proves statements by reflexivity of equality). The final theorem simply states that the new function `f.completion` agrees with the original `f` on all inputs in `α`. The `@[simp]` tags tell the simp tactic that it can use the designated lemmas and that terms will generally become more simple when expression like those on the left side of the equality are replaced by those on the right.

3.2 Design Considerations & Development Process

All of the important design choices in `LinearMapCompletion.lean`, including the file's existence were suggested to me by Omar or Loreaux (44).

3.2.1 Naming Conventions

In an earlier version of this code (44), I was defining the completion of π_f using an intermediate step and the theorem `Completion.map`, but Loreaux rightfully pointed out that the method I was using was not specific to C^*-algebras, and he suggested that I generalize it and put it in a more appropriate file, and name the new definition `UniformSpace.Completion.mapCLM` (44). Later when Omar was reviewing my edits, she suggested that I name it `ContinuousLinearMap.completion` (44).

These different naming suggestion were due to an inconsistency in Mathlib's naming conventions, which Loreaux brought to the attention of the Mathlib community on Zulip. The ensuing discussion ultimately led to the name used above: `ContinuousLinearMap.completion`.

3.2.2 Code Concision

Omar also provided multiple other suggestions that helped to make the code cleaner and more concise, including the use of `MonoidHom` to simplify the definition of `ContinuousLinearMap.completion` (44).

3.3 Summary

In this chapter, we defined the Hausdorff completion (3.1.7) of a space and the completion functor of a map between certain spaces. The completion functor takes a continuous σ-semilinear map $f : \alpha \to \beta$ and extends it

to a continuous σ-semilinear map $\overline{f} : \overline{\alpha} \to \overline{\beta}$, where $\overline{\alpha}$ and $\overline{\beta}$ denote the Hausdorff completions of α and β, and $\overline{f}$ is the notation we define for the newly "completed" map. The exact technical assumptions on α and β are explained earlier in the chapter, but it is typically most convenient to assume α and β are (semi-)normed vector spaces.

In Lean, we denote the completion of a `f : α →SL[σ] β` by writing `f.completion`.

In chapter 5 we will use this completion functor to raise the morphism $\pi_f(a) : A_f \to A_f$ to the morphism $\overline{\pi}_f(a) : H_f \to H_f$ where A_f is an pre-inner product space and $H_f = \overline{A}_f$.

Chapter 4

Constructing the Hilbert Space

4.1 Mathematical Overview

For standard explanations of the GNS construction, refer to the sources referenced in the introduction. In this section, I will explain, in purely mathematical terms (with no Lean code) the steps of the GNS construction in the way that best mimics the formalization as it currently exists in Mathlib. Then I will explain the various ways in which it differs from other methods. In the next section, I will explain the formalization and Lean code.

Although the Mathlib approach to the GNS construction and the traditional approach will seem largely equivalent, and in terms of their results they are completely equivalent, the Mathlib approach is slightly more direct in terms of what needs to be proven when we define the *-homomorphism.

4.1.1 Mathlib Approach

Suppose A is a (possibly non-unital) C^*-algebra. Fix a positive linear functional, $f : A \to \mathbb{C}$. Then,

Definition 4.1.1. For $a, b \in A$, define $\langle a, b \rangle_f = f(a^* b)$.

We then prove that $\langle a, b \rangle_f$ is a semi-inner product (1.2.1) on A. Here, we repeat the definition of the semi-inner product in a way that more closely follows its definition in Mathlib:

Definition 4.1.2. A **semi-inner product** on a vector space A is a map $\langle \cdot, \cdot \rangle : A \times A \to \mathbb{C}$ that it is

- conjugate symmetric (also called Hermitian): $\langle x, y \rangle = \overline{\langle y, x \rangle}$

- positive semi-definite (also called nonnegative-definite): $0 \leq \langle x, x \rangle$
- additive in its first coordinate: $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$, and
- conjugate linear in its first coordinate: $\langle cx, z \rangle = \overline{c} \langle x, z \rangle$,

where $c \in \mathbb{C}$ and $x, y, z \in A$.

For the map we have defined above, $\langle \cdot, \cdot \rangle_f$ (4.1.1), we prove each of these properties fairly directly.

To prove conjugate symmetry we write

$$\begin{aligned}
\langle x, y \rangle_f &= f(x^* y) \\
&= f((y^* x)^*) && \text{because } {}^* \text{ is anti-multiplicative (1.1.5)} \\
&= \overline{f(y^* x)} && \text{because } f \text{ is } {}^*\text{-preserving (1.2.15)} \\
&= \overline{\langle y, x \rangle_f}.
\end{aligned}$$

To prove positive semi-definiteness, we observe that $\langle x, x \rangle_f = f(x^* x)$. Then, because elements of the form $x^* x$ are positive, we have that $0 \leq x^* x$. Because f is positive (i.e. it is an order-preserving homomorphism) (1.2.14), this implies $f(0) \leq f(x^* x)$, which simplifies to $0 \leq f(x^* x) = \langle x, x \rangle_f$.

To prove additivity in the first coordinate, we write

$$\begin{aligned}
\langle x + y, z \rangle_f &= f((x + y)^* z) \\
&= f((x^* + y^*) z) && \text{because } {}^* \text{ is additive (1.1.5)} \\
&= f(x^* z + y^* z) && \text{by distributivity} \\
&= f(x^* z) + f(y^* z) && \text{because } f \text{ is linear (1.2.14)} \\
&= \langle x, z \rangle_f + \langle y, z \rangle_f.
\end{aligned}$$

And, to prove conjugate-linearity in the first coordinate, we write

$$\begin{aligned}
\langle cx, y \rangle_f &= f((cx)^* y) \\
&= f(\overline{c} x^* y) && \text{because } {}^* \text{ is conjugate-linear (1.1.5)} \\
&= \overline{c} f(x^* y) && \text{because } f \text{ is linear (1.2.14)} \\
&= \overline{c} \langle x, y \rangle_f.
\end{aligned}$$

Now that this semi-inner product is defined, there is an induced semi-norm, $\| \cdot \|_f$ (1.2.5) on the elements of A such that

$$\|a\|_f^2 = \langle a, a \rangle_f,$$

for all $a \in A$.

This semi-norm allows us to define H_f to be the Hausdorff completion (3.1.7) of the semi-normed pre-inner product space $(A, \langle \cdot, \cdot \rangle_f)$.

Then, H_f is a Hilbert space because it is complete by construction, it inherits (a well-defined extension of) the inner product from $(A, \langle \cdot, \cdot \rangle_f)$, and it has a norm defined as $\| \cdot \|_f$ extended to H_f, which is a norm (rather than only being a semi-norm) because the Hausdorff completion collapses topologically inseparable points into a single point, so for any $a, b \in H_f$ such that $\|a\| = \|b\| = 0$, it must follow that $a = b = 0 \in H_f$.

4.1.2 Traditional Approach

To the best of my knowledge, the traditional approach is the method used by all sources on the topic, including (14), (45), (46), and (1), because those sources are recounting the method used by Gelfand and Naimark (18) and Segal (20).

The traditional approach has the same setup as the Mathlib approach, but then quickly diverges from it. Begin by letting A be a (possibly non-unital) C^*-algebra, and fix a positive linear functional, $f : A \to \mathbb{C}$.

Then, define a $\| \cdot \|_A$-closed left ideal of A,

$$N_f = \{a \in A \mid f(a^* a) = 0\}$$

Although I will not include here the proof that N_f is a left ideal, it is important to note that the proof requires the inequality

$$|f(b^* a)|^2 \leq f(b^* b) f(a^* a),$$

which is the Cauchy-Schwarz inequality if you first construct the pre-inner product space described in the Mathlib approach, but which can also be derived without any reference to (semi-)inner products, simply by the properties of f. The latter method is used by (1), while the former is much more concise and favored by most other sources.

Regardless of how N_f is proven to be a norm-closed left ideal of A, the next step is to define an inner product on A/N_f as

$$\langle a + N_f, b + N_f \rangle = \langle a, b \rangle_f = f(b^* a)$$

for $a + N_f, b + N_f \in A/N_f$. At this point, one then proves that this function is well-defined (i.e. that it does not depend on choice of representative).

Because this function is defined on A/N_f rather than A, it is already an inner product, not just a semi-inner product. In the Mathlib method, this is achieved by taking the Hausdorff completion of A with respect to the semi-inner product on A.

Once we have defined the inner product on A/N_f, the traditional method then defines H_f as the completion of A/N_f with respect to its inner product. This means that H_f is a Hilbert space, because H_f is a vector space that is complete by construction and inherits an inner product from A/N_f.

4.2 Formalization

4.2.1 Setup & Assumptions

The entire formalization of the GNS construction is included in the single file "Mathlib.Analysis.CStarAlgebra.GelfandNaimarkSegal" / `GelfandNaimarkSegal.lean` (11). After some imports and documentation, the first important code content begins with the lines

```
open scoped ComplexOrder InnerProductSpace
open Complex ContinuousLinearMap UniformSpace
Completion
```

These lines open a few namespaces to make it easier to use certain theorems, structures, and notation. Specifically, opening

- `ComplexOrder` puts an order on the Complex numbers (that coincides with the usual order on $\mathbb{R}$) which allows us to define a positive linear functional,
- `InnerProductSpace` allows us to use the notation `⟪a, b⟫_ℂ` for the inner product,
- `Complex` allows us to refer to multiple theorems about complex numbers without prefixing them with `Complex`,
- `ContinuousLinearMap` allows us to reference various theorems about continuous linear maps without using the prefix `ContinuousLinearMap`,
- `UniformSpace` allows us to refer to the `UniformSpace.Completion` as just the `Completion`, and
- `Completion` allows us to refer to `UniformSpace.Completion.induction_on` as `induction_on`.

Next, the assumptions define the essential structure behind our construction: the positive linear functional, f.

- `{A : Type*}` declares that we have a set A.
- `[NonUnitalCStarAlgebra A]` states that A is a C^*-algebra which is not necessarily unital.
- `[PartialOrder A]` states that there is a partial order on the elements of A, so that the notation $a \leq b$ for $a, b \in A$ can be meaningful.
- `(f : A →p[ℂ] ℂ)` is the only named variable, because we will refer to it explicitly many times throughout the file. `f` is a positive homomorphism from the C^*-algebra A to the C^*-algebra $\mathbb{C}$. This is our positive linear functional. This statement requires `ComplexOrder`, which was opened earlier, and the `[PartialOrder A]` in order for it to be possible to define a positive (order-preserving) map.

Because `f` is a `PositiveLinearMap`, we explicitly put the rest of the code under `namespace PositiveLinearMap`, so that every declaration will receive the prefix `PositiveLinearMap` and so that we can use dot notation to refer to structures like `f.GNS` instead of writing `GNS f`. Why this is useful may become apparent as we discuss the code, but it will also be explained further in the next section.

4.2.2 Defining the Pre-GNS Space

Now the actual construction can begin. We first define the pre-GNS space, which is simply composed of the elements of A together with a positive linear functional, f. The declaration

```
def PreGNS (f : A →p[ℂ] ℂ) := A
```

defines the pre-GNS space as a type whose elements are the elements of A. However, the pre-GNS space does not yet have any of the additional structure that makes the elements of A into a C^*-algebra. This is deliberate, because we do not want the pre-GNS space to be a C^*-algebra. Instead, we want it to become a pre-inner product space. So, we specify that we only want the pre-GNS space to inherit the commutative additive group structure of A and the vector space/module structure of A, by writing

```
instance : AddCommGroup f.PreGNS :=
    inferInstanceAs (AddCommGroup A)
instance : Module ℂ f.PreGNS :=
    inferInstanceAs (Module ℂ A)
```

The `inferInstanceAs` command tells Lean exactly what structures `f.PreGNS` should inherit from `A`. Being able to refer to the pre-GNS space generated by f as `f.PreGNS` instead of `PreGNS f`, is one of the perks of working the in the `PositiveLinearMap` namespace.

Next, as suggested by Loreaux (44), we include some lemmas to streamline our work and the work that will be done by the `simp` tactic when dealing with `f.PreGNS`. First, we explicitly name the maps from A to the pre-GNS space and from the pre-GNS space to A.

We name those maps `toPreGNS` and `ofPreGNS` respectively. Because we defined `f.PreGNS` to be equal to `A`, we can define `toPreGNS` as the identity map by writing

```
def toPreGNS : A ≃ₗ[ℂ] f.PreGNS := LinearEquiv.refl ℂ
_
```

The notation `≃ₗ[ℂ]` specifies that `toPreGNS` is a linear equivalence (called `LinearEquiv` in Mathlib) with complex scalars. Then, because a linear equivalence is defined to be an invertible linear map we can then define the inverse function, `ofPreGNS`, as `f.toPreGNS.symm` (which is necessarily also a linear equivalence) by writing

```
def ofPreGNS : f.PreGNS ≃ₗ[ℂ] A := f.toPreGNS.symm
```

Because `f.toPreGNS` and `f.ofPreGNS` are defined to be inverses of one another, we also introduce two other lemmas which make that relationship more explicit:

```
@[simp]
lemma toPreGNS_ofPreGNS (a : f.PreGNS) :
    f.toPreGNS (f.ofPreGNS a) = a := rfl

@[simp]
lemma ofPreGNS_toPreGNS (a : A) :
    f.ofPreGNS (f.toPreGNS a) = a := rfl
```

Together, these lemmas state that wherever `f.ofPreGNS` and `f.toPreGNS` are composed with one another, they effectively cancel out. The `@[simp]`

tags tell Lean's `simp` tactic that these lemmas can be used to simplify expressions, and that expressions generally become simpler when expressions of the form seen on the left of the equality are replaced by those on the right.

4.2.3 Constructing the Pre-Inner Product Space

Now we can begin the process of constructing a pre-inner product space (1.2.3) structure on the elements of A. Some of the mathematics in this section requires an additional assumption, so we add the line

```
variable [StarOrderedRing A]
```

to specify that A is a ordered *-ring. This means that the non-negative elements of A are exactly the elements which can be expressed in the form a^*a for some $a \in A$. This puts additional structure on the partial order that we assumed initially, but because our initial definitions and lemmas did not require this assumption, we did not include this assumption in our initial assumptions.

This assumption is justified because the partial order that is usually defined on a C^*-algebra induces a ordered *-ring structure, and so most of the theory of C^*-algebras assumes that C^*-algebras are ordered *-rings (45).

Now we define our pre-inner product space. We name it `preGNSpreInnerProdSpace` and we define it using the `PreInnerProductSpace.Core` structure so that we can define our own pre-inner product.

The first step of the process is to specify the name, elements, and scalars of our pre-inner product space and provide a function to use as the semi-inner product. Because we are defining our inner product as $\langle a, b\rangle = f(a^*b)$, we write

```
abbrev preGNSpreInnerProdSpace :
    PreInnerProductSpace.Core ℂ f.PreGNS where
  inner a b := f (star (f.ofPreGNS a) * f.ofPreGNS b)
```

Because we are defining our inner product on elements of `f.PreGNS` instead of elements of `A`, we have to use the function `f.ofPreGNS` to map our input elements back to `A` in order to multiply them, apply the star operation, and pass them to the function `f`, which is only defined on elements of `A`, not on elements of `f.PreGNS`.

The next step is to prove that the function we have defined satisfies all of the requirements to be a semi-inner product (4.1.2). Those properties are that the semi-inner product should be

- `conj_inner_symm` : conjugate-symmetric/Hermitian,
- `re_inner_nonneg` : positive semi-definite,
- `add_left` : additive in the first coordinate, and
- `smul_left` : conjugate-linear in the first coordinate.

and if any of these properties is missing, the Lean InfoView will make sure to remind us to include it.

Proving Conjugate-Symmetry

To prove conjugate-symmetry, we must prove that

```
∀ (x y : f.PreGNS),
(starRingEnd ℂ) (f (star (f.ofPreGNS y) * f.ofPreGNS
x))
= f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

When we remind ourselves that `(starRingEnd ℂ)` is simply complex conjugation, which is the star operation on the complex numbers, we see that this is exactly the usual mathematical definition of conjugate symmetry (1.2.2).

In fact, Lean needs that exact same reminder in addition to a reminder that $f(a^*) = \overline{f(a)}$ (where the overline denotes complex conjugation) because f is *-preserving (1.2.15). In Mathlib, these two facts are are the lemmas `Complex.star_def` and `map_star f` which we pass to `simp`, preceded by a `←` to indicate that, in this case, we want these relationships to be simplified in the opposite direction from how they are listed in Mathlib. The `simp` tactic is then able to construct the rest of the proof automatically, so we can write the entire proof as

```
conj_inner_symm := by
  simp [← Complex.star_def, ← map_star f]
```

If we investigate in more detail what Lean is doing, by replacing `simp` with `simp?`, we can see the other lemmas that are used in the simplification. This check shows that this simplification uses the lemmas `← star_def`, `← map_star f`, `star_mul`, `star_star` and `, implies_true`. Although we cannot see the exact order and ways in which these lemmas are applied, it is likely that they are used in a way similar to this:

First, `star_def : star = ⇑(starRingEnd ℂ)`, which we can tell is used in reverse due to the `←` preceding it, rewrites the goal to

```
star (f (star (f.ofPreGNS y) * f.ofPreGNS x))
= f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

Then, `map_star f : f (star r) = star (f r)` also used in reverse, rewrites the goal to

```
f (star (star (f.ofPreGNS y) * f.ofPreGNS x))
= f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

Next, `star_mul : star (r * s) = star s * star r` rewrites the goal to

```
f (star (f.ofPreGNS x) * star (star (f.ofPreGNS y)))
= f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

and `star_star : star (star r) = r` simplifies it to

```
f (star (f.ofPreGNS x) * f.ofPreGNS y)
= f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

which is true by reflexivity. Finally, the proof is concluded via the lemma `implies_true : (∀ (a : α), True) = True`, which states that the ultimate goal which includes the quantifier `∀ (x y : f.PreGNS)`, must be true because we have shown the predicate to be true.

Proving Positive Semi-Definiteness

To prove that the function we provided is positive semi-definite, we must prove that

```
∀ (x : f.PreGNS),
  0 ≤ RCLike.re (f (star (f.ofPreGNS x) * f.ofPreGNS
  x))
```

As Omar pointed out (44), we can prove this concisely by writing:

```
re_inner_nonneg _ :=
    RCLike.nonneg_iff.mp
        (f.map_nonneg (star_mul_self_nonneg _)) |>.1
```

The first thing to notice about this line is the underscore before the `:=`. Because our goal begins with the quantifier `∀ (x : f.PreGNS)`, the underscore introduces an arbitrary `x` of type `f.PreGNS`, which simplifies our goal to

```
0 ≤ RCLike.re (f (star (f.ofPreGNS x) * f.ofPreGNS x))
```

without the quantifier. In place of the underscore, we could have chosen a specific name for the variable, but because we do not explicitly refer to the variable in the proof, we can simply name it with an underscore to allow Lean to automatically-generate a name for it to use internally.

This proof has many different steps included in it, so let us unpack the proof one piece at a time, starting with the inner-most expression and working our way out.

Our first step is to produce a term of the type

```
0 ≤ f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

that we will then pass to

```
RCLike.nonneg_iff.mp : 0 ≤ z →
    0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

To construct a term of that type, we combine the fact that `f` is positive (1.2.14) with the fact that, because A is a ordered *-ring, elements of the form a^*a are positive/non-negative.

The theorem which states that, in a ordered *-ring, elements of the form `star a * a` are non-negative is

```
star_mul_self_nonneg : 0 ≤ star r * r
```

In this proof, we let `r` be `(f.ofPreGNS x)`. Due to the form of the goal, Lean can infer that the element being passed to `star_mul_self_nonneg` is `(f.ofPreGNS x)`, so instead of passing it explicitly, we can just refer to it using an underscore to tell Lean to infer the correct input. This means that the statement `star_mul_self_nonneg _` has the type:

```
0 ≤ star (f.ofPreGNS x) * f.ofPreGNS x
```

which is exactly what we need to prove that

```
0 ≤ f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

by passing it to

```
PositiveLinearMap.map_nonneg (hx : 0 ≤ x) : 0 ≤ f x
```

Thus, the statement

```
(f.map_nonneg (star_mul_self_nonneg _))
```

has the type

```
0 ≤ f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

which we now pass to

```
RCLike.nonneg_iff.mp : 0 ≤ z →
    0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

The theorem `RCLike.nonneg_iff.mp` that we are using is constructed from the theorem

```
RCLike.nonneg_iff : 0 ≤ z ↔
    0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

which states that a real or complex number is non-negative if and only if its real part is non-negative and its imaginary part is 0.

For our proof, our goal is of the form

```
0 ≤ RCLike.re z
```

for a complex `z`, which in this case is

```
f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

This means that we want to match our goal with the part of the result of `RCLike.nonneg_iff` with the type

```
0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

Because this is the right part of `RCLike.nonneg_iff` if we apply it left to right (we could in principle use this theorem in either direction because it is an if and only if statement), we need to tell Lean to use the theorem with the implication going from left or right. To do this, we append `.mp` the the name of the theorem, so the theorem becomes

```
RCLike.nonneg_iff.mp : RCLike.nonneg_iff : 0 ≤ z →
    0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

The suffix `.mp` stand for modus ponens and specifies that we want to use an if and only if statement in the form where the left side implies the right side (See (42), Section 3.3.4 for more information). If we wanted the converse, we would use `.mpr` for modus ponens reverse.

Once we pass a term of type `0 ≤ z` to `RCLike.nonneg_iff.mp`, we will get a result of the form

```
0 ≤ RCLike.re z ∧ RCLike.im z = 0
```

Our goal only requires the left result, so we append `|>.1` which returns the first part of whatever conjunction precedes it, so in this case, it returns `0 ≤ RCLike.re z`.

This means that when we pass the result we constructed above

```
f.map_nonneg (star_mul_self_nonneg _) :
0 ≤ f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

to `RCLike.nonneg_iff.mp` and extract the left clause of the resulting conjunction, the resulting expression is

```
RCLike.nonneg_iff.mp
    (f.map_nonneg (star_mul_self_nonneg _)) |>.1
```

and has type

```
0 ≤ RCLike.re (f (star (f.ofPreGNS x) * f.ofPreGNS x))
```

which completes the proof.

Proving First Coordinate Additivity

To prove that the function we provided as the semi-inner product is additive in the first coordinate, we must prove that

```
∀ (x y z : f.PreGNS),
  f (star (f.ofPreGNS (x + y)) * f.ofPreGNS z) =
    f (star (f.ofPreGNS x) * f.ofPreGNS z) +
    f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

Like in the prior proof, we can introduce the variables `x`, `y`, and `z` by placing underscores after `add_left` and right before the `:=`, so that Lean will generate names to use internally, which we will not need to reference.

This proof consists of a single sequence of re-writes, which makes it easier to follow in the InfoView. If you place your cursor at the start of the theorem name, the part of the expression which that theorem will re-write will be highlighted, and if you place it at the end of a theorem name, the newly re-written expression will be highlighted.

This lets us see that our initial goal, after we introduced our variables is

```
f (star (f.ofPreGNS (x + y)) * f.ofPreGNS z) =
  f (star (f.ofPreGNS x) * f.ofPreGNS z) +
  f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

Then, when we use the fact that `f.ofPreGNS` is additive/an additive homomorphism (because it is linear) to use the theorem `map_add` to re-write our goal to

```
f (star (f.ofPreGNS x + f.ofPreGNS y) * f.ofPreGNS z)
=
  f (star (f.ofPreGNS x) * f.ofPreGNS z) +
  f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

Next, the additivity of the star operation means we can use

```
star_add : star (r + s) = star r + star s
```

to re-write the goal to

```
f ((star (f.ofPreGNS x) + star (f.ofPreGNS y)) *
    f.ofPreGNS z) =
  f (star (f.ofPreGNS x) * f.ofPreGNS z) +
  f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

Then, `add_mul : (a + b) * c = a * c + b * c`, which is simply the distributive property, re-writes to goal to

```
f (star (f.ofPreGNS x) * f.ofPreGNS z +
  star (f.ofPreGNS y) * f.ofPreGNS z) =
    f (star (f.ofPreGNS x) * f.ofPreGNS z) +
    f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

Finally, we use that fact that `f` is an additive homomorphism (because it is linear), by using `map_add : f (x + y) = f x + f y` again to re-write our goal to

```
f (star (f.ofPreGNS x) * f.ofPreGNS z) +
  f (star (f.ofPreGNS y) * f.ofPreGNS z) =
    f (star (f.ofPreGNS x) * f.ofPreGNS z) +
    f (star (f.ofPreGNS y) * f.ofPreGNS z)
```

which is true by reflexivity of equality, and thus completes our proof.

Proving First Coordinate Conjugate-Linearity

To prove that the function we provided for the semi-inner product is conjugate-linear in the first coordinate, we must prove

```
∀ (x y : f.PreGNS) (r : ℂ),
  f (star (f.ofPreGNS (r • x)) * f.ofPreGNS y) =
  (starRingEnd ℂ) r *
    f (star (f.ofPreGNS x) * f.ofPreGNS y)
```

The entire proof can be written as

```
simp [smul_mul_assoc]
```

which, when we change the `simp` and `simp?`, we can see can be written more explicitly as

```
simp only [map_smul, star_smul, RCLike.star_def,
  smul_mul_assoc, smul_eq_mul, implies_true]
```

Like the proof of conjugate-symmetry, because this is a proof by `simp`, we cannot easily inspect the exact steps used to complete the proof, but the list of theorems provided by `simp?` can give us a general outline.

Because `simp` can manipulate goals that include quantifiers but does not change the quantifiers, we can focus on the portion of the goal that is

```
f (star (f.ofPreGNS (r • x)) * f.ofPreGNS y) =
  (starRingEnd ℂ) r * f (star (f.ofPreGNS x) *
    f.ofPreGNS y)
```

because, by the theorem

```
implies_true : (∀ (a : α), True) = True
```

once we prove that the equation is true, the whole expression with the quantifiers must also be true. Informally, this means that our goal is to prove that $f((r \cdot x)^* y) = \overline{r} f(x^* y)$, where $\cdot$ denotes scalar multiplication.

For this proof, because we cannot inspect the exact steps used to construct the proof, the clearest explanation I can provide is to simply duplicate the informal proof, and specify each step's justification with the relevant Mathlib theorem, like so:

$$
\begin{aligned}
\langle rx, y\rangle_f &= f((r \cdot x)^* y) && \\
&= f((r^* \cdot x^*) y) && \text{by } \texttt{star\_smul} \\
&= f((\overline{r} \cdot x^*) y) && \text{by } \texttt{RCLike.star\_def} \\
&= f(\overline{r} \cdot (x^* y)) && \text{by } \texttt{smul\_mul\_assoc} \\
&= \overline{r} \cdot f(x^* y) && \text{by } \texttt{map\_smul} \\
&= \overline{r} f(x^* y) && \text{by } \texttt{smul\_eq\_mul} \\
&= \overline{r} \langle x, y\rangle_f .
\end{aligned}
$$

The explanation shows a plausible way that all the theorems suggested by `simp?` (except for the theorem `implies_true`, whose purpose was explained previously) are used in the proof.

Constructing the Semi-Norm and Semi-Inner Product

We have almost constructed a pre-inner product space over the elements of `f.PreGNS`. Specifically, we constructed `preGNSpreInnerProdSpace` which has the type

```
PreInnerProductSpace.Core ℂ f.PreGNS
```

A `PreInnerProductSpace.Core` is a structure with a function that satisfies the properties of a semi-inner product, but it not actually a space with a semi-inner product. To construct a space with a semi-inner product we first use the function

```
InnerProductSpace.Core.toSeminormedAddCommGroup
```

to construct a semi-norm from the (soon-to-be) semi-inner product that we used to construct `preGNSpreInnerProdSpace`. To do this, we do not have to declare a new structure. Instead, we simply write

```
noncomputable
instance : SeminormedAddCommGroup f.PreGNS :=
  InnerProductSpace.Core.toSeminormedAddCommGroup
    (c := f.preGNSpreInnerProdSpace)
```

so that `f.PreGNS` can now be treated as an additive commutative group with a semi-norm, where the semi-norm is induced by the (soon-to-be) semi-inner product.

Once the semi-norm is defined, we can now define a true semi-inner product space in the sense defined in Mathlib. This is because Mathlib defines a pre-inner product space (which is somewhat confusingly called an `InnerProductSpace`) as a vector space which has a semi-inner product which is compatible with its semi-norm. This is why we have to construct the semi-norm before we can declare that there is a pre-inner product space structure on the elements of `f.PreGNS`.

To finally construct this pre-inner product space structure on `f.PreGNS`, we write

```
noncomputable
instance : InnerProductSpace ℂ f.PreGNS :=
  InnerProductSpace.ofCore f.preGNSpreInnerProdSpace
```

Now that we have defined a true pre-inner product space, we introduce a few helpful lemmas, the first two of which were provided by Omar and renamed by Loreaux.

The first is a lemma that directly states the definition of our inner product:

```
lemma preGNS_inner_def (a b : f.PreGNS) :
  ⟪a, b⟫_ℂ = f (star (f.ofPreGNS a) * f.ofPreGNS b) :=
    rfl
```

The notation used here was suggested by Loreaux. The second lemma directly states the definition of the semi-inner product-induced semi-norm:

```
lemma preGNS_norm_def (a : f.PreGNS) :
  ‖a‖ = √(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re
  :=
    rfl
```

Both of these lemmas are proved by `rfl` because they are true by definition.

The final lemma is a corollary of the definition of the semi-norm which essentially states that $\|a\|^2 = \langle a, a \rangle$, which in this case means that

```
‖a‖ ^ 2 = f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

The full proof and statement of this lemma, as written by Loreaux is

```
lemma preGNS_norm_sq (a : f.PreGNS) :
    ‖a‖ ^ 2 =
    f (star (f.ofPreGNS a) * f.ofPreGNS a) := by
  have : 0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a) :=
    map_nonneg f <| star_mul_self_nonneg _
  rw [preGNS_norm_def, ← ofReal_pow, Real.sq_sqrt]
  · rw [conj_eq_iff_re.mp this.star_eq]
  · rwa [re_nonneg_iff_nonneg this.isSelfAdjoint]
```

Note that this version of the proof is not the version currently in Mathlib. This version of the proof is from an earlier version of the code in Mathlib (commit #7e1b73e), which I later made more concise but also more opaque. Because this proof is more explicit, it is easier to explain, and so it is the

version I explain here. The updated version of this proof is conceptually similar but relies mostly on the `simp` tactic.

Informally, proving this statement requires a single trivial step that follows from the definitions. That is, we simply square both sides of the expression from the prior lemma. However, because Lean requires complete rigor and it considers the output of `f` to be a complex number, the formal proof is slightly more involved. Specifically, Lean sees this lemma as stating the the semi-norm of a, when treated as a complex number and squared, is equal to $f(a^*a)$, which it also considers to be a complex number. Treating these values as complex makes it much less obvious that the square of a value is a non-negative real and that the output of `f` is real. Of course, we can still prove all of these facts, but that is why the first step of the proof is

```
have : 0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a) :=
  map_nonneg f <| star_mul_self_nonneg _
```

which specifies that $f(a^*a)$ is non-negative and real (has imaginary part 0). That is, this step uses the fact that because `A` is a star-ordered ring, we have that

```
star_mul_self_nonneg : 0 ≤ star a * a
```

for any `a : A`, which in this case is `f.ofPreGNS a`. Then, using the fact that `f` is an order homomorphism/positive (1.2.14), we can use

```
f.map_nonneg (hx : 0 ≤ x) : 0 ≤ f x
```

to obtain the result that

```
0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

Because there is no name specified after **have**, this result can only be referred to as `this`, until some other unnamed theorem is defined.

The rest of the proof is a sequence of re-writes. Recall that the goal is to show that

```
↑‖a‖ ^ 2 = f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

Also observe that the `↑` in this expression indicates that `‖a‖` is being coerced to a complex number, so although `‖a‖ : ℝ`, `↑‖a‖ : ℂ`. We first use

```
preGNS_norm_def :
    ‖a‖ = √(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re
```

to re-write the goal to

```
↑√(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re ^ 2 =
  f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

Then, rewriting with `ofReal_pow : ↑(r ^ n) = ↑r ^ n` from right to left moves the type coercion so it is applied after the square, changing the goal to

```
↑(√(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re ^ 2) =
  f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

Now, re-writing with

```
Real.sq_sqrt {x : ℝ} (h : 0 ≤ x) : √x ^ 2 = x
```

splits our goal into two cases, because in order to apply the re-write, we must show that the value under the radical is real and non-negative. That is, we must first prove that the assumptions of `Real.sq_sqrt` are satisfied, so we must prove that `x : ℝ` and `0 ≤ x`, for our `x`, which, in this case, is

```
(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re
```

To prove the first assumption, we must prove that

```
↑(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re =
f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

This equation states that taking the real part of

```
(f (star (f.ofPreGNS a) * f.ofPreGNS a))
```

and then coercing that to a complex number, produces the same value as

```
f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

That is, we are proving that the value is real by proving that it is equal to its real part.

The first step to prove this requires us to use the fact that in a ordered *-ring, non-negative elements are self-adjoint. That is, for any `x` in a ordered *-ring, we have

```
LE.le.star_eq (hx : 0 ≤ x) : star x = x
```

We can then pass the fact proven earlier, that

```
0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

to this theorem by writing `this.star_eq`. Observe that we can use this dot notation instead of writing `LE.le.star_eq this` because `this` is term whose type is less than or equal to (that is, `LE`) and `LE.le.star_eq` is in the `LE` namespace. Writing `this.star_eq` produces a term of the type

```
star (f (star (f.ofPreGNS a) * f.ofPreGNS a)) =
  f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

Now, to prove that `(star (f.ofPreGNS a) * f.ofPreGNS a)` is real, we use the fact that a complex number is real if and only if it is equal to its conjugate. That is, we use the theorem

```
conj_eq_iff_re : (starRingEnd ℂ) z = z ↔ ↑z.re = z
```

And we use it in the forward direction, from left to right, so we append `.mp` to the theorem name. This means that the term

```
conj_eq_iff_re.mp this.star_eq
```

has the type

```
↑(f (star (f.ofPreGNS a) * f.ofPreGNS a)).re =
  f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

which is exactly our goal, so we can prove our desired result be writing

```
exact conj_eq_iff_re.mp this.star_eq
```

Now that we have proven that

```
(star (f.ofPreGNS a) * f.ofPreGNS a)
```

is real, to complete our proof that it is equal to the square of its square root, we must additionally prove that it is non-negative.

To do this, we use the fact that non-negative elements of ordered *-rings are self-adjoint, which is stated by the theorem

```
LE.le.isSelfAdjoint : (hx : 0 ≤ x) : IsSelfAdjoint x
```

When we write `this.isSelfAdjoint` to use the fact the our desired value is non-negative with this theorem, we get the result that

```
IsSelfAdjoint (f (star (f.ofPreGNS a) * f.ofPreGNS a))
```

We then pass this result as an assumption to the theorem

```
re_nonneg_iff_nonneg :
    (hx : IsSelfAdjoint x) : 0 ≤ x.re ↔ 0 ≤ x
```

to get a term of the type

```
0 ≤ (f (star (f.ofPreGNS a) * f.ofPreGNS a)).re ↔
  0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

The left side of this statement is exactly the goal we wish to prove, so we can use this statement, by writing

```
rwa [re_nonneg_iff_nonneg this.isSelfAdjoint]
```

to re-write our goal to be exactly

```
0 ≤ f (star (f.ofPreGNS a) * f.ofPreGNS a)
```

which is exactly a statement that we already proved, so our proof is complete.

Defining the GNS Hilbert Space

Now the we have a pre-inner product space defined over `f.PreGNS`, we can construct the GNS Hilbert space with a single line:

```
abbrev GNS := Completion f.PreGNS
```

This defines `f.GNS` to be the Hausdorff completion of the pre-GNS space generated by f. Because a Hilbert space is a complete, normed, inner-product space, the fact that `f.GNS` is a Hilbert space may not be immediately obvious. That is, it is clear that it must be complete by construction, and it is a semi-inner product space because it inherits the semi-inner product from the semi-inner product on the pre-GNS space. However, it is not clear that the semi-inner product is an inner product (that is, that it is positive definite, rather than positive semi-definite) and that the semi-norm it inherits is now a norm.

Once we recall the definition of the Hausdorff completion, however, both of those facts will become clear. Because the Hausdorff completion first groups any points that are topologically inseparable into a single point (or equivalence class), the semi-inner product and semi-norm must become

an inner product and norm respectively. This is because two points are grouped together under the Hausdorff completion exactly when the semi-norm of their difference is 0. This means that for any a, b in the pre-GNS space where $a \neq b$ and $\|a\| = \|b\| = 0$, after applying the Hausdorff completion $a = b$, so $\|a\| = 0$ implies $a = 0$, meaning the norm is positive definite. This suffices to show that the norm on the GNS space is a norm, rather than only a semi-norm. In the language of Mathlib, this means that `f.GNS` is a `NormedAddCommGroup`.

That fact that `f.GNS` satisfies all of the properties of a Hilbert space is already inferred by Lean, as we can see if we write

```
instance : HilbertSpace ℂ f.GNS where
```

Because Lean is not suggesting any fields to include or properties to prove after the **`where`**, we can see that Lean can construct for itself all the results it needs to prove that `f.GNS` is a Hilbert space.

4.3 Design Considerations & Development Process

4.3.1 Choosing to use the Hausdorff Completion

The most notable difference between the Mathlib implementation of the GNS construction and the traditional approach is that the Mathlib approach does not define an ideal/subspace/submodule and quotient space in order to define π_f. It is entirely possible to implement the traditional approach in Lean, and I did, in fact, implement it that way initially, but while reviewing my code, Loreaux suggested that I use a separation quotient (3.1.5) instead (44). That is, instead of grouping elements of A into equivalence classes by modding out by an ideal, like in the traditional approach and completing the resulting space with respect to its metric, I would immediately group points into equivalence classes based on the semi inner-product induced semi-norm, using the separation quotient, and complete that space with respect to its metric to produce H_f.

So, I re-worked the code to use that approach, but it was still needlessly complicated and required using two induction principles: one for the separation quotient and one for the metric (Hausdorff) completion. After I did that, Loreaux then pointed out that using both the separation quotient and the Hausdorff completion was redundant (44), because the Hausdorff completion is already defined to produce a separation quotient (3.1.7). This allowed me to make the code much more concise than it was when using

either the traditional approach or the separation quotient + Hausdorff completion approach.

Because the Hausdorff completion approach allowed me to avoid proving that a $\pi_f(a)$ was well defined on a quotient space, and made the code generally cleaner and more concise, I agreed with Loreaux that this was the best option for formalizing the GNS construction for Mathlib.

4.3.2 Other Design Choices

Dot Notation

Another design choice which I did not make initially, but was suggested to me be reviewers was the choice to use dot notation. Specifically Monica Omar suggested that I use dot notation for everything which is parametrized by the linear functional f (44). That is what allows us to write `f.GNS` instead of `GNS f`. Although the default behavior in Lean is that terms take parameters as inputs listed on the right, separated by spaces, because the definition of almost every structure and term involved in the GNS construction depends on f, in this case it makes the code significantly more readable to write f as a prefix on each term, instead of always listing it as the first parameter being passed to an expression.

This also makes the Lean code more similar to the informal ways of writing these functions, because informally, we choose to write f as a subscript, rather than pass it as a parameter, even though that would have the equivalent meaning. This is why all of the code for the GNS construction is in the `PositiveLinearMap` namespace: being in that namespace allows us to pass positive linear map parameters, like the positive linear function f, to expressions via dot notation.

Using abbrev instead of def

Recall that we defined the GNS space by writing.

```
abbrev GNS := Completion f.PreGNS
```

We use **`abbrev`** instead of **`def`** here so that we do not have to explicitly tell Lean to unfold the definition of `f.GNS` when we refer to it in subsequent proofs. This helps to keep proofs more concise and readable. If you open the file in VSCode and change the **`abbrev`** to **`def`**, Lean will immediately highlight many locations in the code that no longer compile, indicating that

if we had chosen to use **def** instead of **abbrev**, each of those locations in would have required additional code, which would make the file needlessly complicated.

4.4 Summary

In this chapter we constructed the Hilbert space H_f / `f.GNS`. To do this, we defined A_f / `f.PreGNS` to have the same elements as our C^*-algebra A, but we added an semi-inner product that we defined as $\langle x, y \rangle_f = f(x^* y)$ so that A_f / `f.PreGNS` is a pre-inner product space. We then defined H_f / `f.GNS` to be the Hausdorff completion (3.1.7) of A_f / `f.PreGNS`. This approach is slightly different from the approach most sources use, but it is slightly more direct and more suitable for Mathlib.

Now that this Hilbert space is defined, we proceed in the next chapter to define a continuous linear operator $\pi_f(a) : A_f \to A_f$ and construct $\overline{\pi}_f(a) : H_f \to H_f$. This will allow us to prove that $\overline{\pi}_f : A \to B(H_f)$ is a *-homomorphism, which will complete our proof of the GNS construction.

Chapter 5

Constructing the *-Homomorphism

Now that we have defined the Hilbert space H_f, we can define the homomorphism from our C^*-algebra, A, to the algebra of bounded linear operators, $B(H_f)$ (notation from 1.2.9), on the Hilbert space, H_f. To do this, we first define a function from our C^*-algebra, A, to the algebra of bounded linear operators, $B(A_f)$, on the Pre-GNS space, A_f / `f.PreGNS`.

5.1 Mathematical Overview

5.1.1 Defining a *-homomorphism on A_f

The first task is to define a function

$$\pi_f : A \longmapsto B(A_f),$$

where the domain A is our C^*-algebra, and the codomain $B(A_f)$ is the algebra of bounded linear operators on our Pre-GNS space generated by f, whose elements are the elements of A, but which is a pre-inner product space (1.2.3), rather than a C^*-algebra.

Because the output of π_f when passed an element of A is a function from A_f to A_f, we could also write

$$\pi_f(a) : A_f \to A_f$$

where both A_fs refer to our Pre-GNS space.

Once we have defined this $\pi_f(a)$, we will then be able to apply our completion functor (Chapter 3) to make it a function from the completion of A_f to the completion of A_f, which, by definition, is H_f. That is how we will produce the bounded linear operator $\pi_f(a) \in B(H_f)$.

Because the elements of A_f are the elements of A, we can define π_f like so:

Definition 5.1.1. We define the map π_f, where $f : A \to \mathbb{C}$ is a positive linear functional as

$$\pi_f(a) : b \in A_f \longmapsto ab \in A_f.$$

That is, $\pi_f(a)$ multiplies its input b, which is an element of the Pre-GNS space, by a on the left. This is linear because multiplication is linear, but proving that it is bounded (continuous) takes some more work.

Proving Continuity of $\pi_f(a)$

To show that $\pi_f(a)$ is bounded we prove that for all $x \in A_f$,

$$\|\pi_f(a)(x)\|_f \leq \|a\|_A \|x\|_f.$$

To prove this, first use the fact that norms are non-negative to re-write this statement to

$$\|\pi_f(a)(x)\|_f^2 \leq \|a\|_A^2 \|x\|_f^2.$$

Next, we can apply the definition of $\pi_f(a)$ to re-write this to

$$\|ax\|_f^2 \leq \|a\|_A^2 \|x\|_f^2.$$

Then, using the fact that $\|\cdot\|_f$ is, by definition, the semi-norm induced by $\langle \cdot, \cdot \rangle_f$ (1.2.5), we can re-write this to

$$f((ax)^*(ax)) \leq \|a\|_A^2 f(x^*x),$$

which by the anti-multiplicativity of * (1.1.5) and the linearity of f (1.1.16), simplifies to

$$f(x^*a^*ax) \leq f(\|a\|_A^2 x^*x).$$

Now, because f is positive/order preserving (1.2.14), this statement will be proven if we prove that

$$x^*a^*ax \leq \|a\|_A^2 x^*x.$$

This statement is then proven directly from the fact that a^*a is self-adjoint. That is, we use the following theorem, stated in Mathlib as:

```
CStarAlgebra.star_left_conjugate_le_norm_smul :
    star a * b * a ≤ ‖b‖ • (star a * a)
```

which tells us that for a self-adjoint $b \in A$, $x^*bx \leq \|b\|_A(x^*x)$, so

$$x^*a^*ax \leq \|a^*a\|_A x^*x,$$

which simplifies to

$$x^*a^*ax \leq \|a\|_A^2 x^*x,$$

via the C^*-identity (1.1.5), which completes our proof.

This means that we have proven that $\pi_f(a)$ is continuous and linear, so $\pi_f(a) \in B(A_f)$. Now, all that remains is to prove that π_f is a *-homomorphism, and we will do that in the next section when we also raise it to H_f from A_f.

5.1.2 Defining the *-homomorphism to H_f

In Mathlib, a *-homomorphism (`NonUnitalStarAlgHom`) is defined as a map $\pi : A \to B$ where A and B are (possibly non-unital) *-algebras and for all $c \in \mathbb{C}$, $a, b \in A$, we have that π

- maps scalars: $\pi(ca) = c\pi(a)$
- maps zero: $\pi(0_A) = 0_B$
- is additive: $\pi(a + b) = \pi(a) + \pi(b)$
- is multiplicative: $\pi(ab) = \pi(a)\pi(b)$
- is *-preserving: $\pi(a^*) = \pi(a)^*$.

The attentive reader may have noticed that if π maps scalars, then it must also map 0, and so requiring that $\pi(0_A) = 0_B$ is redundant. This is certainly true in our case, but in Mathlib, these homomorphisms are defined so generally that they do not assume there is a scalar 0. That is, they assume

that the set of scalars in only a monoid, rather than a ring. However, in our formalization, we do take advantage of the fact that $\mathbb{C}$ is a ring and so mapping scalars implies mapping zero.

Before we prove any of these properties, however, we first must lift $\pi_f(a) : A_f \to A_f$ to $\pi_f(a) : H_f \to H_f$, so that $\pi_f(a) \in B(H_f)$. That is, we need to ensure that $\pi_f(a)$ is an operator on our Hilbert space H_f, not just our pre-inner product space A_f.

To do this, we apply the completion functor discussed in Chapter 3, and so from this point on, $\overline{\pi}_f(a)$ will refer to the lifted $\overline{\pi}_f(a) \in B(H_f)$ and $\pi_f(a)$ will refer to the original $\pi_f(a) \in B(A_f)$.

These functions are thus related by the property that for all $a \in A_f \subseteq H_f$, $\pi_f(a) = \overline{\pi}_f(a)$, and we have that $\overline{\pi}_f$ is bounded/continuous because, as proven in section 5.1.1, $\pi_f(a)$ is bounded and as proven in Chapter 3, the completion of a continuous linear map is continuous and linear.

Proving Scalar Mapping

The first property we prove is that if $c \in \mathbb{C}$ and $a \in A$, then $\overline{\pi}_f(ca) = c\overline{\pi}_f(a)$. To prove this, consider an arbitrary input $x \in H_f$ so that our goal is to prove

$$\overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)$$

To prove this, we use Theorem 3.1.11 in a very similar manner to how we used it in Chapter 3. In this case, because we want to prove that for all $x \in H_f$, $\overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)$, we must prove that

- $\{x \in H_f : \overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)\}$ is closed, and
- for all $x \in A_f$, $\overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)$.

This first statement, that $\{x \in H_f : \overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)\}$ is closed, can be proven via Theorem 3.1.12. That is, we can conclude that $\{x \in H_f : \overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)\}$ is closed by the continuity of $\overline{\pi}_f(ca)$ and of $c\overline{\pi}_f(a)$. Those functions are both continuous because, as we proved in Section 5.1.1, $\pi_f(a)$ is continuous for all $a \in A$, and we know that scalar multiplication is continuous, so $\overline{\pi}_f(ca)$ and $c\overline{\pi}_f(a)$ are both continuous because they are compositions of continuous functions.

This means that we next must prove that for all $x \in A_f$, $\overline{\pi}_f(ca)(x) = c\overline{\pi}_f(a)(x)$. Luckily, in this case, because $\pi_f(a)(x) = \overline{\pi}_f(a)(x)$ for all $x \in A_f$, we can re-write our goal as

$$\pi_f(ca)(x) = c\pi_f(a)(x),$$

and apply the definition of π_f to simplify this to

$$(ca)x = c(ax),$$

which is evidently true because multiplication is associative.

Proving Zero Mapping

Now that we have proven that $\overline{\pi}_f(ca) = c\overline{\pi}_f(a)$ for all $a \in A$, we can prove that $\overline{\pi}_f(0_A) = 0_{B(H_f)}$ by observing that

$$\begin{aligned}\overline{\pi}_f(0_A) &= \overline{\pi}_f(0 \cdot 0_A) \\ &= 0\overline{\pi}_f(0_A) \\ &= 0_{B(H_f)}.\end{aligned}$$

That is, we have proven that $\overline{\pi}_f(0_A)$ is the linear operator that maps every element of H_f to $0_{H_f} \in H_f$.

Proving Additivity

Now we prove that $\overline{\pi}_f$ is additive, which means we must prove that for all $a, b \in A$,

$$\overline{\pi}_f(a + b) = \overline{\pi}_f(a) + \overline{\pi}_f(b).$$

To prove this statement, we first pass an arbitrary element $c \in H_f$ to the functions on each side of the inequality so that our goal becomes

$$\overline{\pi}_f(a + b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c).$$

Now to complete the proof, we use Theorem 3.1.11 in essentially the same way as we used it to prove scalar mapping. That is, in order to treat c as though it is an element of A_f instead of H_f, we apply Theorem 3.1.11.

In order to apply Theorem 3.1.11, we first have to prove its hypotheses, which are that

- $\{x \in H_f : \overline{\pi}_f(a + b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c)\}$ is closed in H_f, and
- for all $c \in A_f$, $\overline{\pi}_f(a + b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c)$.

To prove the first hypothesis, we again use Theorem 3.1.12, which allows us to prove that $\{x \in H_f : \overline{\pi}_f(a+b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c)\}$ is closed in H_f by proving that $\overline{\pi}_f(a+b)$ and $(\overline{\pi}_f(a) + \overline{\pi}_f(b))$ are both continuous functions. That $\overline{\pi}_f(a+b)$ is continuous follows from that fact that $\pi_f(a+b)$ is continuous, and so $\overline{\pi}_f(a+b)$ must be continuous. (Recall that we proved that the completion of a continuous linear function is continuous and linear in Chapter 3). Similarly, $(\overline{\pi}_f(a) + \overline{\pi}_f(b))$ is continuous because addition is continuous and each $\overline{\pi}_f(a)$ is continuous.

Next, to prove the second hypothesis of Theorem 3.1.11, we observe that because $c \in A_f$, $\overline{\pi}_f(a)(c) = \pi_f(a)(c)$ for all $a \in A$. This lets us simplify our goal like so:

$$\begin{aligned}
&\overline{\pi}_f(a+b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c) && \\
&\pi_f(a+b)(c) = (\pi_f(a) + \pi_f(b))(c) && \text{because } c \in A_f \\
&\pi_f(a+b)(c) = \pi_f(a)(c) + \pi_f(b)(c) && \\
&\quad (a+b)c = ac + bc && \text{by the definition of } \pi_f \text{ (5.1.1)} \\
&\quad ac + bc = ac + bc, &&
\end{aligned}$$

which completes the proof.

Proving Multiplicativity

Next, we prove that $\overline{\pi}_f$ is multiplicative. That is, that $\overline{\pi}_f(ab) = \overline{\pi}_f(a)\overline{\pi}_f(b)$. First, observe that because $\overline{\pi}_f(a), \overline{\pi}_f(b) \in B(H_f)$, multiplying these functions together, really means composing them. So, our goal is really to prove that $\overline{\pi}_f(ab) = \overline{\pi}_f(a) \circ \overline{\pi}_f(b)$.

To do this, we first prove a related lemma:

Lemma 5.1.2. For $a, b \in A$, $\pi_f(ab) = \pi_f(a) \circ \pi_f(b)$.

Proof. To prove this, we pass an arbitrary $c \in A_f$ to the functions to change our goal to

$$\pi_f(ab)(c) = (\pi_f(a) \circ \pi_f(b))(c)$$

We can then simplify this to

$$\pi_f(ab)(c) = \pi_f(a)(\pi_f(b)(c)),$$

which, by repeatedly applying the definition of π_f simplifies to

$$\begin{aligned}
abc &= \pi_f(a)(bc) \\
abc &= abc,
\end{aligned}$$

completing the proof. □

Now, to prove that $\overline{\pi}_f(ab) = \overline{\pi}_f(a)\overline{\pi}_f(b)$, we can let $c \in H_f$ and pass it to both functions, so our goal becomes

$$\overline{\pi}_f(ab)(c) = (\overline{\pi}_f(a)\overline{\pi}_f(b))(c).$$

Then, we again use Theorem 3.1.11 with Theorem 3.1.12, which tells us that we can complete our proof by showing that

1. $\{c \in H_f : \overline{\pi}_f(ab)(c) = (\overline{\pi}_f(a)\overline{\pi}_f(b))(c)\}$ is closed in H_f, which, by Theorem 3.1.12, can be proven by showing that
   - $\overline{\pi}_f(ab)$ is continuous and
   - $(\overline{\pi}_f(a)\overline{\pi}_f(b))$ is continuous, and that
2. for all $c \in A_f$, $\overline{\pi}_f(ab)(c) = (\overline{\pi}_f(a)\overline{\pi}_f(b))(c)$.

The second statement is proven immediately by Lemma 5.1.2 because in $B(A_f)$ multiplication is exactly function composition, so

$$\overline{\pi}_f(a)\overline{\pi}_f(b) = \overline{\pi}_f(a) \circ \overline{\pi}_f(b)$$

and we have that

$$\overline{\pi}_f(ab)(c) = \pi_f(ab)(c)$$

and

$$(\overline{\pi}_f(a)\overline{\pi}_f(b))(c) = (\pi_f(a)\pi_f(b))(c)$$

because $c \in A_f$.

To prove the first statement, observe that continuity of $\overline{\pi}_f(ab)$ follows immediately from the continuity of $\pi_f(ab)$ by the result in Chapter 3, and that continuity of $(\overline{\pi}_f(a)\overline{\pi}_f(b)) = \overline{\pi}_f(a) \circ \overline{\pi}_f(b)$ follows from the fact that a composition of continuous functions is continuous.

Proving *-Preserving

Finally, we prove that $\overline{\pi}_f$ is *-preserving. That is, that $\overline{\pi}_f(a^*) = \overline{\pi}_f(a)^*$, where the adjoint of $\overline{\pi}_f(a) \in B(H_f)$ is defined as in 1.2.16.

Recall that by the definition of the adjoint, what we are trying to prove is that for all $x, y \in H_f$,

$$\langle \overline{\pi}_f(a^*)(x), y\rangle_f = \langle x, \overline{\pi}_f(a)(y)\rangle_f$$

To do this, we again use Theorems 3.1.11 and 3.1.12 to treat x and y as elements of A_f intead of H_f.

To do this, the first thing we have to prove is that

$$\{(x,y) \in H_f \times H_f : \langle \overline{\pi}_f(a^*)(x), y\rangle_f = \langle x, \overline{\pi}_f(a)(y)\rangle_f\}$$

is a closed set. To do this, we can use Theorem 3.1.12, which states that it suffices to prove that

$$\begin{aligned}(x,y) &\mapsto \langle \overline{\pi}_f(a^*)(x), y\rangle_f, \text{ and}\\ (x,y) &\mapsto \langle x, \overline{\pi}_f(a)(y)\rangle_f\end{aligned}$$

are both continuous. That both those functions are continuous follows from the continuity of the map $(x,y) \mapsto \langle x, y\rangle_f$ and the continuity of $\overline{\pi}_f(a)$ and $\overline{\pi}_f(a^*)$.

Now, all that is left is to prove that

$$\langle \overline{\pi}_f(a^*)(x), y\rangle_f = \langle x, \overline{\pi}_f(a)(y)\rangle_f$$

for all $x, y \in A_f$. We can prove this by applying definitions as follows

$$\begin{aligned}
\langle \overline{\pi}_f(a^*)(x), y\rangle_f &= \langle x, \overline{\pi}_f(a)(y)\rangle_f && \\
\langle \pi_f(a^*)(x), y\rangle_f &= \langle x, \pi_f(a)(y)\rangle_f && \text{because } x, y \in A_f\\
\langle a^*x, y\rangle_f &= \langle x, ay\rangle_f && \text{by the definition of } \pi_f \text{ (5.1.1)}\\
f((a^*x)^*y) &= f(x^*(ay)) && \text{by the definition of } \langle\cdot,\cdot\rangle_f \text{ (4.1.1)}\\
f(x^*ay) &= f(x^*ay) && \text{by the properties of } ^* \text{ (1.1.5).}
\end{aligned}$$

Thus, we have proven that $\overline{\pi}_f$ is *-preserving.

5.1.3 The Unital Case

We have now defined a *-homomorphism, $\overline{\pi}_f : A \to B(H_f)$ on a non-unital C^*-algebra, A. If however, A is unital (1.1.7), we can prove that $\overline{\pi}_f$ is a unital homomorphism (1.1.11). That is, we can prove that $\overline{\pi}_f(1_A) = 1_{B(H_f)}$, where $1_{B(H_f)}$ is the identity function from H_f to H_f. To prove this, we first pass an arbitrary $c \in H_f$ to the functions to re-write our goal as

$$\overline{\pi}_f(1_A)(c) = 1_{B(H_f)}(c),$$

which simplifies to

$$\overline{\pi}_f(1_A)(c) = c.$$

Then by applying Theorems 3.1.11 and 3.1.12 just as in the previous proofs, we can re-write our goal to

$$\pi_f(1_A)(c) = c,$$

for $c \in A_f$. Then we apply the definition of π_f to simplify this to

$$1_A c = c.$$

Then, because 1_A is the identity in A, $1_A c = c$, and the proof is complete.

5.2 Formalization

5.2.1 Defining the *-homomorphism

First, I will explain the code that defines `leftMulMapPreGNS`, the Mathlib name for π_f. The code for `leftMulMapPreGNS` as it exists in Mathlib and which I explain below was written entirely by Loreaux (44) because his version of the proof was more general and more concise than the one I wrote. (See section 5.3.1 for more explanation of the differences between our proofs.)

The first thing to observe about `leftMulMapPreGNS` is its type. The first line of its declaration is

```
noncomputable def leftMulMapPreGNS (a : A) :
    f.PreGNS →L[ℂ] f.PreGNS :=
```

As usual, disregard the **`noncomputable`** designation, and focus on the type annotation: `f.PreGNS →L[ℂ] f.PreGNS`. This means that we are defining a function from the Pre-GNS space generated by f to the Pre-GNS space generated by f, and the `→L` denotes that the function is both linear and bounded, and the `[ℂ]` indicates that the function is linear with scalars from $\mathbb{C}$.

Also note the `(a : A)` that follows the declaration name, which indicates that we only get our desired function after providing some element of A to `leftMulMapPreGNS`. This matches the type of $\pi_f : A \to B(A_f)$ because $B(A_f)$ is the set of bounded linear maps from our Pre-GNS space to our Pre-GNS space, and we only get an output in $B(A_f)$ after providing an element of A as input.

The first line of the definition of `leftMulMapPreGNS` is dense, but explainable. The line

```
f.toPreGNS.toLinearMap ∘ₗ mul ℂ A a ∘ₗ
  f.ofPreGNS.toLinearMap
```

defines the function itself and proves that it is linear, and the rest of the content of the definition is a proof that the function is also continuous.

Each part of the line defining the function can be interpreted directly, with very little additional information. The `∘ₗ` operators denote composition of linear functions. That is, they compose two functions which are known to be linear to produce a single function, which must also be linear. So, to interpret this composition functions, we should begin with the rightmost function, because it is the function that is applied first to an input.

The first function is `f.ofPreGNS.toLinearMap`. Recall that the function `f.ofPreGNS` is defined as a linear equivalence that takes an element of the type `f.PreGNS` and returns an element of type `A` (link to explanation). The suffix `.toLinearMap` converts the linear equivalence into a `LinearMap` (linear function) so that it can be composed via `∘ₗ` with another linear function to produce a new linear function.

The linear function with which `f.ofPreGNS.toLinearMap` is composed is left multiplication by the element `a` of our C^*-algebra `A`. That multiplication is denoted `mul ℂ A a`, and can be treated as a linear map because multiplication is linear.

The functions that we have composed so far to produce

```
mul ℂ A a ∘ₗ f.ofPreGNS.toLinearMap
```

which takes an element of the Pre-GNS space, converts it to an element of A, and multiplies it on the left by $a \in A$. So, the final step must be to convert the resulting product back into an element of the Pre-GNS space. This is done by composing `mul ℂ A a ∘ₗ f.ofPreGNS.toLinearMap` with `f.toPreGNS.toLinearMap`, which is the function that converts an element of A into an element of the Pre-GNS space. Thus, we have defined a linear function that takes an element of the Pre-GNS space and returns an element of the Pre-GNS space. That is, we have defined a $\pi_f(a)$ that is linear, and now the final step in defining $\pi_f(a)$ is to prove that it is continuous.

To prove that $\pi_f(a)$`/leftMulMapPreGNS` is continuous, we use the theorem `LinearMap.mkContinuous` which takes a linear map and a proof that that linear map is bounded, and returns a continuous linear map (that is equal to the map that we provided to it). Specifically, that theorem has the signature

```
LinearMap.mkContinuous (f : E →ₛₗ[σ] F) (C : ℝ)
    (h : ∀ (x : E), ‖f x‖ ≤ C * ‖x‖) : E →SL[σ] F
```

(some parts are omitted for brevity). This means that we pass a (semi-)linear map f, and an explicit bound C, together with a term of the type

```
∀ (x : E), ‖f x‖ ≤ C * ‖x‖
```

that states that for all inputs x to f, $\|f(x)\| \leq C\|x\|$, which is what it means for a function f to be bounded/continuous.

In the case of `leftMulMapPreGNS`, `mkContinuous` is invoked by writing

```
f.toPreGNS.toLinearMap ∘ₗ mul ℂ A a ∘ₗ
  f.ofPreGNS.toLinearMap
    |>.mkContinuous ‖a‖ fun x ↦ by
```

The `|>` notation takes everything to its left and passes it as an input the the expression on its right. In this case, that means that it takes the linear function that we defined and passes it as the first parameter to `mkContinuous`.

We can then see that the next parameter passed to `mkContinuous` is `‖a‖`, which will serve as the explicit bound C in the proof that

```
∀ (x : E), ‖f x‖ ≤ C * ‖x‖
```

Now, recall that a term of the type `∀ (x : E), ‖f x‖ ≤ C * ‖x‖` is essentially a function that takes a `(x : E)` as input and produces a term of type `‖f x‖ ≤ C * ‖x‖`. This definition takes advantage of this fact by constructing a function explicitly, instead of defining a separate term of type `∀ (x : E), ‖f x‖ ≤ C * ‖x‖`, giving it a name, and then passing it to `mkContinuous`. That is, we explicitly begin defining a function that takes a `(x : E)` as input by writing `fun x ↦` and we provide a way to produce a term of type `‖f x‖ ≤ C * ‖x‖`, given that `x : E`. (Note that `E` is a general placeholder type, which in this case is `f.PreGNS`.)

Now, we can explain the proof that `‖f x‖ ≤ C * ‖x‖`. Looking at the exact proof state and goal after each step provides far more information than is useful to us, so I will explain this proof mostly with conventional mathematical notation, but with reference to the exact steps being taken by the Lean formalization.

We have already defined a linear $\pi_f(a)$ as

```
f.toPreGNS.toLinearMap ∘ₗ mul ℂ A a
  ∘ₗ f.ofPreGNS.toLinearMap
```

so our goal is now to prove that for an arbitrary $x \in A_f$,

$$\|\pi_f(a)(x)\|_f \leq \|a\|_A \|x\|_f,$$

where $\|\cdot\|_A$ denotes the norm of the C^*-algebra A, and $\|\cdot\|_f$ denotes the inner product-induced norm of A_f.

The first step of this proof is a sequence of re-writes:

```
rw [← sq_le_sq₀ (by positivity) (by positivity),
  mul_pow, ← RCLike.ofReal_le_ofReal (K := ℂ),
  RCLike.ofReal_pow,
  RCLike.ofReal_eq_complex_ofReal, preGNS_norm_sq]
```

The first theorem used is `sq_le_sq₀` which states that for non-negative a and b, $a \leq b$ implies $a^2 \leq b^2$. Essentially, this squares both sides of our goal, re-writing it to

$$\|\pi_f(a)(x)\|_f^2 \leq (\|a\|_A \|x\|_f)^2$$

The reason the theorem is written as

```
← sq_le_sq₀ (by positivity) (by positivity)
```

with a `←` is because the theorem is bi-directional (if and only if) and we are using it from right to left. The `(by positivity)` parameters specify that Lean should construct its own proofs of the fact that $\|\pi_f(a)(x)\|_f$ is non-negative and that $\|a\|_A \|x\|_f$ is non-negative. These proofs can be computed easily because Lean knows that absolute values are non-negative and that the product of non-negative values is non-negative.

The next theorem used is `mul_pow` which states that $(ab)^n = a^n b^n$, which re-writes our goal to

$$\|\pi_f(a)(x)\|_f^2 \leq \|a\|_A^2 \|x\|_f^2.$$

The next theorem is

```
← RCLike.ofReal_le_ofReal (K := ℂ)
```

which changes both sides of the inequality to be complex numbers instead of real numbers. This is possible because the Complex order agrees with the Real order on the Real line. The theorem `RCLike.ofReal_pow` then re-writes the goal from treating $\|\pi_f(a)(x)\|_f^2$ as a single complex number to treating $\|\pi_f(a)(x)\|_f$ as a complex number, with a natural number exponent,

2. The next theorem, `RCLike.ofReal_eq_complex_ofReal`, just specifies that the function that was implicitly being used to access the real part of each of these numbers (which could have been either real or complex) is specifically taking the real part of a complex number. This step is pointless mathematically. It is only included to deal with the quirks of Mathlib and the difference between a number that is like a Real or Complex number and a number that is actually a Complex number.

The last step in this re-write invokes `preGNS_norm_sq` which used the definition of $\|\cdot\|_f^2$ to re-write our goal to

$$f((\pi_f(a)(x))^*\pi_f(a)(x)) \leq \|a\|_A^2\|x\|_f^2$$

The proof now takes a break from re-writing our goal to prove a useful lemma that states that

$$x^*a^*(ax) \leq \|a\|_A^2 x^*x,$$

where $a \in A$ and $x \in A_f$.

It proves this via another sequence of re-writes. It begins by using `← mul_assoc` to re-write the goal to

$$x^*a^*ax \leq \|a\|_A^2 x^*x$$

and uses it again on a different part of the expression by writing `mul_assoc _ (star a)`, which re-writes the goal to

$$x^*(a^*a)x \leq \|a\|_A^2 x^*x$$

The theorem `sq` then re-writes $\|a\|_A^2$ to $\|a\|_A\|a\|_A$, making the goal

$$x^*(a^*a)x \leq \|a\|_A\|a\|_A x^*x.$$

It then invokes the C^*-identity (1.1.5) as

```
← CStarRing.norm_star_mul_self (x := a)
```

which re-writes the goal to

$$x^*(a^*a)x \leq \|a^*a\|_A x^*x.$$

and uses `smul_mul_assoc` to re-write the goal to

$$x^*(a^*a)x \leq \|a^*a\|_A (x^*x).$$

This goal can the be proven immediately by the theorem

```
CStarAlgebra.star_left_conjugate_le_norm_smul
```

because its type is `star a * b * a ≤ ‖b‖ • (star a * a)` for self-adjoint `b`, which, in this case is a^*a, which Lean can infer is self-adjoint.

Now, that we have as a lemma:

$$x^*a^*(ax) \leq \|a\|_A^2 x^*x$$

we can return to proving our main goal, which is that

$$f((\pi_f(a)(x))^*\pi_f(a)(x)) \leq \|a\|_A^2\|x\|_f^2.$$

To prove this, we use **`calc`** mode, which allows us to write our proof as

```
calc
 _ ≤ f (‖a‖ ^ 2 • star (f.ofPreGNS x) * f.ofPreGNS x)
  := by
    simpa using OrderHomClass.mono f this
 _ = _ := by simp [← Complex.coe_smul, preGNS_norm_sq,
    smul_mul_assoc]
```

which specifies a chain of calculations with appropriate justifications that ultimately prove our goal. The first and the last underscores match the initial and the final expressions with the left and right expressions in our goal, respectively. The middle underscore refers to the expression

```
f (‖a‖ ^ 2 • star (f.ofPreGNS x) * f.ofPreGNS x)
```

from the right side of the first inequality in the **`calc`** chain.

This means that the first step of our chain proves that

$$f((\pi_f(a)(x))^*\pi_f(a)(x)) \leq f(\|a\|^2 x^*x).$$

Loreaux's proof proves this in a single line

```
simpa using OrderHomClass.mono f this
```

which elides many details which we can investigate by re-writing `simpa` as `simpa?`, which shows that we could re-write this tactic as

```
simpa only [LinearMap.coe_comp, LinearEquiv.coe_coe,
  coe_coe, Function.comp_apply, mul_apply',
  ofPreGNS_toPreGNS, star_mul] using
    OrderHomClass.mono f this
```

Because this is a version of the `simp` tactic, we cannot directly see all of the steps that it takes, but based on the theorems it references, we can make a good guess as to what it is doing.

The part of the command that reads

```
using OrderHomClass.mono f this
```

states that after `simpa` has simplified the goal, it should have the same type as `OrderHomClass.mono f this`, and so the proof will be complete.

Note that `this` refers to the lemma proven above. When we write

```
#check OrderHomClass.mono f this
```

(`mono` here is short for monotone/monotonic) we can see that this expression has the type

```
f (star (f.ofPreGNS x) * star a * (a * f.ofPreGNS x))
≤
  f (‖a‖ ^ 2 • star (f.ofPreGNS x) * f.ofPreGNS x)
```

which, in conventional mathematical notation is

$$f(x^*a^*(ax)) \leq f(\|a\|_A^2 x^* x).$$

Comparing this with our goal, we see that their right sides are the same, so the `simpa` tactic must be simplifying the left side of our goal,

$$f((\pi_f(a)(x))^* \pi_f(a)(x))$$

to become

$$f(x^*a^*(ax)).$$

Specifically, the left side of our goal starts out as

```
f (star (f.ofPreGNS ((↑f.toPreGNS ∘ₗ
  ↑((ContinuousLinearMap.mul ℂ A) a) ∘ₗ
    ↑f.ofPreGNS) x)) *
  f.ofPreGNS ((↑f.toPreGNS ∘ₗ
    ↑((ContinuousLinearMap.mul ℂ A) a) ∘ₗ
      ↑f.ofPreGNS) x))
```

with full definition of $\pi_f(a)$/`leftMulMapPreGNS` included, along with the explicit maps between A/`A` and A_f/`f.PreGNS`.

The first theorem referenced by `simp`, `LinearMap.coe_comp` is likely used to convert the linear compositions, $\circ_l$ to normal function compositions, $\circ$. Next, the theorem `LinearEquiv.coe_coe` simplifies linear equivalences like `f.ofPreGNS` and `f.toPreGNS` that are being treated as linear maps, to simple being treated as functions. The next theorem, `coe_coe`, similarly converts continuous linear functions to functions.

The theorem `Function.comp_apply` then re-writes composed functions that look like $(f \circ g)(x)$ to look like $f(g(x))$. Next, `mul_apply'` re-writes any instance of `mul ℂ A a b` to simply be expressed as `a * b`.

After all of these manipulations, the left side of the expression likely looks more like

```
f (star (f.ofPreGNS (f.toPreGNS  (a * (f.ofPreGNS
x)))) * f.ofPreGNS (f.toPreGNS (a * (f.ofPreGNS x)))
```

The next theorem, `ofPreGNS_toPreGNS` eliminates the consecutive applications of `f.ofPreGNS` and `f.toPreGNS`, simplifying the goal to

```
f (star (a * (f.ofPreGNS x)) *
    (a * (f.ofPreGNS x)))
```

Finally, `star_mul` re-writes $(ab)^*$ to b^*a^*, changing the left side of the goal to

```
f (star (f.ofPreGNS x) * star a * (a * f.ofPreGNS x))
```

which completes the simplification. Thus, the proof of this step is completed.

The next step of the **`calc`** chain aims to prove that

```
f (‖a‖ ^ 2 • star (f.ofPreGNS x) * f.ofPreGNS x)
    = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

This is again done by a `simp`, so we re-write it to a `simp?` and see the full list of theorems it uses:

```
simp only [← Complex.coe_smul, ofReal_pow,
  smul_mul_assoc, map_smul, smul_eq_mul,
  ofReal_mul, preGNS_norm_sq]
```

The first theorem, `Complex.coe_smul`, when used in reverse as indicated by the $\leftarrow$, re-writes scalar multiplication of a real number to scalar multiplication of the real part of that number when it is treated as a complex number. This changes to goal to

```
f (↑(‖a‖ ^ 2) • star (f.ofPreGNS x) * f.ofPreGNS x)
  = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

which is nearly identical to the original goal, except for a `↑` in front of `(‖a‖ ^ 2)`, which takes the real part of `(‖a‖ ^ 2)` and treats it as a complex number.

The expression `↑(‖a‖ ^ 2)` can then be simplified by the next theorem, `ofReal_pow`, to become `↑‖a‖ ^ 2` where only `‖a‖` is treated as a complex number, and the exponent `2` is treated as a natural number. This changes the goal to

```
f (↑‖a‖ ^ 2 • star (f.ofPreGNS x) * f.ofPreGNS x)
  = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

The next theorem, `smul_mul_assoc` then associates the multiplication to the right, changing the goal to

```
f (↑‖a‖ ^ 2 • (star (f.ofPreGNS x) * f.ofPreGNS x))
  = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

The next theorem, `map_smul` uses the linearity of `f` to change the goal to

```
↑‖a‖ ^ 2 • f (star (f.ofPreGNS x) * f.ofPreGNS x)
  = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

by moving the scalar `↑‖a‖ ^ 2` outside of `f`. Then, `smul_eq_mul` changes the scalar multiplication to regular complex multiplication:

```
↑‖a‖ ^ 2 * f (star (f.ofPreGNS x) * f.ofPreGNS x)
  = ↑(‖a‖ ^ 2 * ‖x‖ ^ 2)
```

Then, `ofReal_mul` distributes the `↑` that was taking the real part of `(‖a‖ ^ 2 * ‖x‖ ^ 2)` onto each individual term, changing the goal to

```
↑‖a‖ ^ 2 * f (star (f.ofPreGNS x) * f.ofPreGNS x)
  = ↑‖a‖ ^ 2 * ↑‖x‖ ^ 2
```

Finally, `preGNS_norm_sq` applies the definition of the norm on the Pre-GNS space to change `↑‖x‖ ^ 2` to

```
f (star (f.ofPreGNS x) * f.ofPreGNS x)
```

making both sides equal and completing the proof.

Thus, we have formalized the proof that the map we defined by $\pi_f(a) : b \in A_f \mapsto ab \in A_f$ (5.1.1) is continuous/bounded.

5.2.2 Raising `leftMulMapPreGNS` to a *-Homomorphism on `f.GNS`

Proving `leftMulMapPreGNS_mul_eq_comp`

Before we lift `leftMulMapPreGNS` to `f.GNS` and prove that it is a non-unital *-algebra homomorphism (i.e. a *-homomorphism), we will first prove a brief lemma:

```
lemma leftMulMapPreGNS_mul_eq_comp (a b : A) :
  f.leftMulMapPreGNS (a * b) = f.leftMulMapPreGNS a
    ∘L f.leftMulMapPreGNS b := by
  ext c; simp [mul_assoc]
```

Mathematically, this states that $\pi_f(ab) = \pi_f(a) \circ \pi_f(b)$, which will be a helpful property when proving that π_f is multiplicative when lifted to `f.GNS`.

To prove this lemma, we first write `ext c`, which provides an input $c \in A_f$ to the functions on either side of the equality, changing our goal to

```
(f.leftMulMapPreGNS (a * b)) c =
((f.leftMulMapPreGNS a).comp (f.leftMulMapPreGNS b)) c
```

We then complete the proof by writing `simp [mul_assoc]`. When we change `simp` to `simp?` we see that it suggests

```
simp only [leftMulMapPreGNS_apply, mul_assoc,
coe_comp',
  Function.comp_apply, ofPreGNS_toPreGNS]
```

which we could replace with

```
rw [leftMulMapPreGNS_apply, coe_comp',
  Function.comp_apply, leftMulMapPreGNS_apply,
  leftMulMapPreGNS_apply, ofPreGNS_toPreGNS,
  mul_assoc]
```

to see how each individual theorem might be used. This lets us infer that `simp` effectively simplifies the goal as follows

$\pi_f(ab)(c) = (\pi_f(a) \circ \pi_f(b))(c)$

$\pi_f(ab)(c) = \pi_f(a)(\pi_f(b)(c))$ by `Function.comp_apply`

$abc = a(bc)$ by using `leftMulMapPreGNS_apply` 3 times

$abc = abc$ by `mul_assoc`.

Defining `gnsNonUnitalStarAlgHom`

Now we define

```
gnsNonUnitalStarAlgHom : A →⋆na[ℂ] f.GNS →L[ℂ] f.GNS
```

That is, we define a function called `gnsNonUnitalStarAlgHom` that takes an element of `A` and is a non-unital algebra *-homomorphism into `f.GNS →L[ℂ] f.GNS`. The notation `→⋆na` indicates that the function is a (⋆)-homomorphism that is (n)on-unital and acts between (a)lgebras.

The first step is to define the function itself, by providing a function in the `toFun` field, like so

```
noncomputable def gnsNonUnitalStarAlgHom :
    A →⋆na[ℂ] f.GNS →L[ℂ] f.GNS where
  toFun a := (f.leftMulMapPreGNS a).completion
```

Here, `.completion` refers to the `ContinuousLinearMap.completion` defined in Chapter 3.

Now, we prove one last lemma before each *-algebra homomorphism property is proven one at a time.

Proving completion_leftMulMapPreGNS_map_smul

We first prove a lemma that we mark as **`private`** because it should not be referenced outside this file and is only being used to make some later proofs more concise.

The theorem and its proof is stated as

```
@[simp]
private lemma completion_leftMulMapPreGNS_map_smul
  (m : ℂ) (x : A) :
    (f.leftMulMapPreGNS (m • x)).completion =
      m • (f.leftMulMapPreGNS x).completion := by
  ext a
  induction a using induction_on with
  | hp => apply isClosed_eq <;> fun_prop
  | ih a => simp [smul_mul_assoc]
```

This theorem states that for a scalar $m \in \mathbb{C}$, and an $x \in A$, $\pi(mx) = m\pi(x)$ which, as explained in the prior section, is one of the required properties of a homomorphism. The reason that we separate this out as its own lemma is

that, as mentioned in the prior section, the fact that π maps 0 to 0 is a direct consequence of π mapping scalars. So, proving it as a separate lemma that we can invoke both to prove that π maps scalars and that it maps 0 to 0 helps to make our code a little bit more concise.

This lemma also provides an opportunity to explain a basic principle that is used to prove that π satisfies most of the other properties of a *-homomorphism. Specifically, this proof uses function extensionality and an induction principle.

Function extensionality is the principle that two functions are equal if the always provide the same output when given the same input. Because this proof proves a statement that asserts that two functions are equal, it is natural for us to invoke function extensionality to complete the proof. To do this, we use the `ext` tactic, and we can write `ext a` to pass an arbitrary input `a` to the function on the left and the function on the right. Thus, after using this tactic, our goal changes from

```
(f.leftMulMapPreGNS (m • x)).completion =
  m • (f.leftMulMapPreGNS x).completion
```

to

```
(f.leftMulMapPreGNS (m • x)).completion a =
  (m • (f.leftMulMapPreGNS x).completion) a
```

Then, because `a : Completion f.PreGNS`, we can use induction on `a` so that we can treat it as though it has type `f.PreGNS`, while still proving the desired statement for a more general `a` of type `Completion f.PreGNS` (or equivalently `f.GNS`). In this case, we use Theorem 3.1.11, which requires us to prove that

- `IsClosed {a | (f.leftMulMapPreGNS (m • x)).completion a = (m • (f.leftMulMapPreGNS x).completion) a}`, and
- `(f.leftMulMapPreGNS (m • x)).completion ↑a = (m • (f.leftMulMapPreGNS x).completion) ↑a`

That is, we must prove

- $\{a \in H_f : \overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)\}$ is closed in H_f, and
- $\overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)$ for $a \in A_f$.

To prove the first goal, that $\{a \in H_f : \overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)\}$ is closed in H_f, we use the Theorem 3.1.12, which lets us prove that $\{a \in H_f : \overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)\}$ is closed by proving that our functions, $\overline{\pi}(mx)(a)$, and $m\overline{\pi}(x)(a)$ are both continuous.

As seen in the code mentioned above, we complete this part of the proof by writing

```
apply isClosed_eq <;> fun_prop
```

The purpose of the `apply` tactic is to take a theorem whose output type is your desired goal, and change your goal to proving the necessary assumptions for the theorem to hold. That is, if we have a theorem that states that $p \implies q$ and our goal is to prove q, then we apply that theorem, our new goal is to prove p, because by that theorem, proving p, proves q.

In this case, because `isClosed_eq` has two assumptions which are that each of the two functions is continuous, when we write

```
apply isClosed_eq
```

we are left with two goals:

1. prove that $\overline{\pi}(mx)(a)$ is continuous, and
2. prove that $m\overline{\pi}(x)(a)$ is continuous.

Then, because we are proving that two different functions are continuous, we use the notation `<;>` to indicate that we want to apply the same tactic to both goals. The tactic that we apply is `fun_prop` which is a tactic that automatically proves properties of functions (usually continuity) by finding theorems that state that the functions that compose a particular function are each continuous and using the fact that a composition of continuous functions is continuous. In our case, because we already proved that `leftMulMapPreGNS` is continuous (5.2.1), Lean can use that fact, along with the information it already has about scalar multiplication being continuous, to prove continuity of these functions.

With this section of the proof completed, we have proven that $\{a \in H_f : \overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)\}$ is closed in H_f. The next result we need to prove is that $\overline{\pi}(mx)(a) = m\overline{\pi}(x)(a)$ for $a \in A_f$. We complete this part of the proof by writing

```
simp [smul_mul_assoc]
```

When we change `simp` to `simp?`, we see that this tactic becomes

```
simp only [completion_apply_coe,
  leftMulMapPreGNS_apply, smul_mul_assoc,
  map_smul, Completion.coe_smul, coe_smul',
  Pi.smul_apply]
```

Although we cannot know the exact steps taken that use these theorems to prove the goal, we can guess at the process it uses, and to ensure that our guess is plausible, we can base it on an alternate version of the proof that invokes the same theorems, but uses the tactic `rw` instead of `simp`. Specifically, instead of `simp [smul_mul_assoc]`, we could instead write

```
rw [completion_apply_coe, leftMulMapPreGNS_apply,
  smul_mul_assoc, map_smul, Completion.coe_smul,
  coe_smul', Pi.smul_apply, completion_apply_coe,
  leftMulMapPreGNS_apply]
```

and if you truly wish to understand what each step here does, open the code in VSCode and step through the proof, or go to live.lean-lang.org and copy and paste the proof from B.1 to interact with this proof online. (See 2.2.1 for more explanation on how to interact with Lean proofs.)

Informally, the goal of this section of the proof is to prove that

$$\overline{\pi}(m \cdot x)(a) = m \cdot \overline{\pi}(x)(a),$$

where $a \in A_f$ (instead of H_f) because we applied our induction principle, but Lean is still treating a as an element of H_f. That is, it is being coerced to have the type `f.GNS`.

So, the first theorem we use is `completion_apply_coe`, which allows us to treat a as an element of A_f directly (change its type to `f.PreGNS`), at the expense of the overall output of $\overline{\pi}(x)(a)$ being an element of H_f.

Next, we use `leftMulMapPreGNS_apply`, which is a theorem that was automatically generated from the definition of `leftMulMapPreGNS` by the decorator `@[simps!]` which tells Lean to automatically generate lemmas we can use to simplify the definition of `leftMulMapPreGNS`. Specifically, `leftMulMapPreGNS_apply` states that $\pi_f(x)(a) = xa$, which is simply the definition of π_f (and thus why we let Lean generate this lemma automatically).

At this point, our goal is simplified to

$$m \cdot xa = m \cdot \overline{\pi}_f(x)(a).$$

Next, we use associativity of our multiplications (`smul_mul_assoc`) and the fact that the map (which is the identity map) between A and A_f is linear and thus maps scalars, to simplify our goal to

$$m \cdot xa = m \cdot \overline{\pi}_f(x)(a),$$

which looks the same as the prior state of our goal because this informal representation does not include the maps between A_f and A or between A_f and H_f. The next theorems, `coe_smul'`, `Pi.smul_apply` `completion_apply_coe`, similarly deal with type coercions and maps, which I will not explain here, but the final theorem, `leftMulMapPreGNS_apply` is what completes the proof by applying the definition of $\overline{\pi}_f(x)$ to simplify the right hand side to $m \cdot xa$, so the goal becomes

$$m \cdot xa = m \cdot xa,$$

which completes the proof.

Proving `map_smul'`

Now, we move into proving that $\overline{\pi}_f$ satisfies the properties of a *-homomorphism (1.1.11). Because we proved the lemma

```
completion_leftMulMapPreGNS_map_smul
```

previously (5.2.2), and we gave it the `@[simp]` decorator, we can prove the property `map_smul'`, that $\overline{\pi}_f(cx) = c\overline{\pi}_f(x)$, simply by writing `simp`.

Proving `map_zero'`

Similarly, because our ring of scalars $\mathbb{C}$ includes a scalar 0, `map_zero'`, the fact that $\overline{\pi}_f(0) = 0$ can be proven by writing

```
simpa using f.completion_leftMulMapPreGNS_map_smul 0 0
```

which tells Lean that it can use typical simplification rules on our goal and on the expression `f.completion_leftMulMapPreGNS_map_smul 0 0` in order to complete the proof. Because

```
f.completion_leftMulMapPreGNS_map_smul
```

is defined by

```
lemma completion_leftMulMapPreGNS_map_smul
    (m : ℂ) (x : A) :
    (f.leftMulMapPreGNS (m • x)).completion =
      m • (f.leftMulMapPreGNS x).completion
```

when we write `f.completion_leftMulMapPreGNS_map_smul 0 0` we are passing the complex number 0 and the element 0 of A to the theorem, so that the type of the resulting expression is

```
(f.leftMulMapPreGNS (0 • 0)).completion =
    0 • (f.leftMulMapPreGNS 0).completion
```

which states that $\overline{\pi}_f(0 \cdot 0_A) = 0 \cdot \overline{\pi}_f(0_A)$, which the `simpa` tactic can simplify to $\overline{\pi}_f(0_A) = 0$, which completes the proof.

As with `simp`, we can change `simpa` to `simpa?` to see what theorems it uses. In this proof, doing that lets us see that it uses the theorem `smul_zero` and `zero_smul` which state that $a \cdot 0 = 0$ and $0 \cdot a = 0$ respectively.

Proving `map_add'`

Now, our task is to prove that $\overline{\pi}_f(a + b) = \overline{\pi}_f(a) + \overline{\pi}_f(b)$, which in Lean, means our goal is to prove

```
(f.leftMulMapPreGNS (a + b)).completion =
    (f.leftMulMapPreGNS a).completion +
    (f.leftMulMapPreGNS b).completion
```

To prove this, we use a strategy almost identical to the one we used to prove `completion_leftMulMapPreGNS_map_smul` (5.2.2).

Our first step is to use the `ext` tactic to pass an arbitrary input to the function on both sides of the equality in our goal. That is, we write `ext c`, so goal becomes

```
(f.leftMulMapPreGNS (a + b)).completion c =
    ((f.leftMulMapPreGNS a).completion +
        (f.leftMulMapPreGNS b).completion) c
```

Then, we use the same induction principle we used in the proof of `completion_leftMulMapPreGNS_map_smul` (5.2.2). That is, we use Theorem 3.1.11.

The first thing we have to prove to use that induction principle is that the set

$$\{c \in H_f : \overline{\pi}_f(a + b)(c) = (\overline{\pi}_f(b) + \overline{\pi}_f(b))(c)\}$$

is closed in H_f. In the InfoView, we can see that this goal is stated as

```
IsClosed
  {a_1 | (f.leftMulMapPreGNS (a + b)).completion a_1 =
      ((f.leftMulMapPreGNS a).completion +
       (f.leftMulMapPreGNS b).completion) a_1}
```

Then, as before, we complete this step of the proof by writing

```
apply isClosed_eq <;> fun_prop
```

which uses theorem `isClosed_eq` (Theorem 3.1.12) and basic properties of the relevant functions to automatically complete the proof. In this case, `fun_prop` is again showing that $\overline{\pi}_f(a + b)$ and $(\overline{\pi}_f(b) + \overline{\pi}_f(b))$ are both continuous, which follows from the fact that (as we have already proven both formally 5.2.1 and informally 5.1.1) $\overline{\pi}_f$ is continuous, and the sum of continuous functions is continuous.

Next, to complete the proof, we must prove that for all $c \in A_f$, $\overline{\pi}_f(a + b)(c) = (\overline{\pi}_f(a) + \overline{\pi}_f(b))(c)$. This can be proven by appealing to the definition π_f, and can thus be completed by writing

```
simp [add_mul, Completion.coe_add]
```

which reminds `simp` to use the distributive property and the fact that the map between A_f and H_f is additive (because it is linear).

Changing `simp` to `simp?` shows us that the full list of theorems used by `simp` is

```
completion_apply_coe, leftMulMapPreGNS_apply,
add_mul, map_add, Completion.coe_add, add_apply
```

so we can see that `simp` invokes the definition of `leftMulMapPreGNS`/$\overline{\pi}_f$ via `leftMulMapPreGNS_apply` and uses a few other theorems that deal with type coercions and basic algebraic manipulations.

Proving `map_mul'`

The proof of the multiplicativity of $\overline{\pi}_f$, that $\overline{\pi}_f(ab) = \overline{\pi}_f(a)\overline{\pi}_f(b)$ is virtually identical to the proof of additivity. The whole proof reads

```
map_mul' _ _ := by
    ext c
    induction c using induction_on with
      | hp => apply isClosed_eq <;> fun_prop
      | ih c => simp
```

Every step is identical, except for the fact that the final `simp` does not require us to pass it any additional theorems. It is able to find all the theorems on its own.

We can see that the reason this `simp` can find all the relevant theorems is that, the `simp` can be replaced by

```
simp only [leftMulMapPreGNS_mul_eq_comp,
    completion_apply_coe, coe_comp',
    Function.comp_apply, leftMulMapPreGNS_apply,
    ofPreGNS_toPreGNS, ContinuousLinearMap.coe_mul]
```

which uses a theorem we defined earlier, `leftMulMapPreGNS_mul_eq_comp` (5.1.2/5.2.2), which is the key to this proof and can be accessed by `simp` because we added the designation `@[simp]` to the lemma's declaration.

All the other theorems that this `simp` uses deal with definitions and properties of various functions and type coercions.

If you are curious about those individual steps, replace the `simp` with

```
rw [leftMulMapPreGNS_mul_eq_comp,
completion_apply_coe,
    coe_comp', Function.comp_apply,
    leftMulMapPreGNS_apply,
    ContinuousLinearMap.coe_mul,
    Function.comp_apply, leftMulMapPreGNS_apply,
    ofPreGNS_toPreGNS, completion_apply_coe,
    completion_apply_coe, leftMulMapPreGNS_apply,
    leftMulMapPreGNS_apply, ofPreGNS_toPreGNS]
```

so you can inspect each step individually.

Proving `map_star'`

To prove that $\overline{\pi}_f$ is *-preserving means that we have to prove

```
(f.leftMulMapPreGNS (star a)).completion =
  star (f.leftMulMapPreGNS a).completion
```

which means

$$\overline{\pi}_f(a^*) = \overline{\pi}_f(a)^*$$

To do this, we use the definition of the adjoint (1.2.16). Specifically, we use the theorem `eq_adjoint_iff`, which essentially states that $a = b^* \iff$

$\forall x, y \in A_f, \langle ax, y\rangle_f = \langle x, by\rangle_f$. Specifically, the full type signature of `eq_adjoint_iff` is

```
ContinuousLinearMap.eq_adjoint_iff.{u_1, u_2, u_3}
    {𝕜 : Type u_1} {E : Type u_2} {F : Type u_3}
    [RCLike 𝕜] [NormedAddCommGroup E]
    [NormedAddCommGroup F] [InnerProductSpace 𝕜 E]
    [InnerProductSpace 𝕜 F] [CompleteSpace E]
    [CompleteSpace F] (A : E →L[𝕜] F) (B : F →L[𝕜] E)
    :
  A = adjoint B ↔
    ∀ (x : E) (y : F), ⟪A x, y⟫_𝕜 = ⟪x, B y⟫_𝕜
```

which shows that we should pass it two continuous linear maps in order to get a resulting expression of type

```
A = adjoint B ↔
  ∀ (x : E) (y : F), ⟪A x, y⟫_𝕜 = ⟪x, B y⟫_𝕜
```

that we can actually use as a theorem.

We can also see that, because `eq_adjoint_iff` is in the namespace `ContinuousLinearMap`, we can pass our first continuous linear map to `eq_adjoint_iff` by prepending it with dot notation. That is, because `((f.leftMulMapPreGNS _).completion` is a continuous linear map (we will address the underscore in a moment), we can write the expression

```
((f.leftMulMapPreGNS _).completion.eq_adjoint_iff _)
```

which Lean can infer has type

```
(f.leftMulMapPreGNS (star a)).completion
  = adjoint (f.leftMulMapPreGNS a).completion ↔
    ∀ (x y : Completion f.PreGNS),
      ⟪(f.leftMulMapPreGNS (star a)).completion x,
      y⟫_ℂ
      = ⟪x, (f.leftMulMapPreGNS a).completion y⟫_ℂ
```

The underscores in the original expressions are placeholders that make the expression shorter by allowing Lean to infer the values of the expressions that should replace the underscores. In this case, the first underscore represents `(star a)` and the second underscore represents `(f.leftMulMapPreGNS a).completion`.

Now because we have produced an expression that is a bi-directional if and only if statement, we add `.mpr` at the end of the expression to indicate that we want to use it as a theorem where the implication goes from right to left. Thus, we produce a term of the type

```
(∀ (x y : Completion f.PreGNS),
    ⟪(f.leftMulMapPreGNS (star a)).completion x, y⟫_ℂ
    =
      ⟪x, (f.leftMulMapPreGNS a).completion y⟫_ℂ) →
        (f.leftMulMapPreGNS (star a)).completion =
          adjoint (f.leftMulMapPreGNS a).completion
```

Then, because `adjoint` is equivalent to `star`, we can see that the conclusion of the implication is precisely the statement we are trying to prove. Thus, our next step is to prove the left (first) part of the implication.

To indicate to Lean that we will now be trying to prove the first part of this implication, we use the tactic `refine`. The behavior of this tactic is perhaps best described by its docstring (see Tactic Reference in (43)) which states:

> `refine e` behaves like `exact e`, except that named (`?x`) or unnamed (`?_`) holes in `e` that are not solved by unification with the main goal's target type are converted into new goals, using the hole's name, if any, as the goal case name.

In our case, we use just one unnamed goal that we represent as `?_`, and because of the type of the preceding expression, Lean can see that what we want to prove is that

```
(∀ (x y : Completion f.PreGNS),
    ⟪(f.leftMulMapPreGNS (star a)).completion x, y⟫_ℂ
    =
      ⟪x, (f.leftMulMapPreGNS a).completion y⟫_ℂ)
```

because, by the theorem we have constructed, this is sufficient to imply the statement which we are ultimately trying to prove.

This means that for the rest of the proof, our only goal is to prove

```
(∀ (x y : Completion f.PreGNS),
    ⟪(f.leftMulMapPreGNS (star a)).completion x, y⟫_ℂ
    =
      ⟪x, (f.leftMulMapPreGNS a).completion y⟫_ℂ)
```

To do this, the natural first step is to introduce arbitrary, fixed $x, y \in H_f$ using the tactic `intro`. That is, we write

```
intro x y
```

so that our new goal is

```
⟪(f.leftMulMapPreGNS (star a)).completion x, y⟫_ℂ =
    ⟪x, (f.leftMulMapPreGNS a).completion y⟫_ℂ
```

and we have arbitrary fixed `x` and `y` of type `Completion f.PreGNS` that we can work with and refer to.

Our next step is to use the same induction principle we have been using in all the prior proofs, Theorem 3.1.11, but this time, we will use it on x and on y. Instead of doing the induction on each variable separately, which would work and is completely rigorous and acceptable, we choose to induct on both variables at the same time for the sake of concision.

This means that, to finish the proof, we write

```
induction x, y using induction_on₂ with
| hp => apply isClosed_eq <;> fun_prop
| ih x y => simp [mul_assoc, preGNS_inner_def]
```

which is essentially just a more concise way to write

```
induction y using induction_on with
| hp => apply isClosed_eq <;> fun_prop
| ih y =>
induction x using induction_on with
| hp => apply isClosed_eq <;> fun_prop
| ih x => simp [mul_assoc, preGNS_inner_def]
```

However, for the sake of exactness, we should explain the theorem `induction_on₂`. In Mathlib, we can see that it is defined

```
theorem UniformSpace.Completion.induction_on₂{α : Type
u_1}
    [UniformSpace α] {β : Type u_2} [UniformSpace β]
    {p : Completion α → Completion β → Prop}
    (a : Completion α) (b : Completion β)
    (hp : IsClosed {x : Completion α × Completion β |
      p x.1 x.2})
    (ih : ∀ (a : α) (b : β), p ↑a ↑b) :
  p a b
```

This shows us that in order to use this theorem, the first assumption that we have to prove is the one named `hp` that states that

$$\{x \in \overline{\alpha} \times \overline{\beta} : P(x)\}$$

is a closed set. This is identical to the assumption required by `induction_on` except for the fact that, the variable `x` is now an ordered pair whose components are denoted `x.1` and `x.2`.

To prove this assumption we use the exact same tactic:

```
apply isClosed_eq <;> fun_prop
```

as used above, for the same purpose as in the prior proofs. However, because it is proving a slightly different goal here, it is worth explaining explicitly.

When we place our cursor at the start of the `apply`, we can see that the goal is to prove

```
IsClosed {x |
  ⟪(f.leftMulMapPreGNS (star a)).completion
      x.1, x.2⟫_ℂ
    = ⟪x.1, (f.leftMulMapPreGNS a).completion x.2⟫_ℂ}
```

To prove this, we again use the theorem `isClosed_eq` (Theorem 3.1.12) to allow us to prove this goal by proving that the function of `x` on the left of the equality and the function of `x` on the right of the inequality are both continuous. We can see how Lean states these goals by moving our cursor to the end of `isClosed_eq`, where we can then see two goals:

```
Continuous fun y ↦
  ⟪(f.leftMulMapPreGNS (star a)).completion
    y.1, y.2⟫_ℂ
```

and

```
Continuous fun y ↦
  ⟪y.1, (f.leftMulMapPreGNS a).completion y.2⟫_ℂ
```

Informally, these goals are proving that for a $y \in H_f \times H_f$,

$$y \mapsto \langle \overline{\pi}_f(a^*)(y_1), y_2 \rangle$$

and

$$y \mapsto \langle y_1, \overline{\pi}_f(a)(y_2) \rangle$$

are continuous, respectively.

The tactic `fun_prop` is then able to prove these goals, using continuity of π_f as we proved earlier and continuity of the inner product, likely using a theorem like `continuous_inner`.

Now that the first assumption for `induction_on₂` has been proven, all that remains is to prove the second required assumption, which in this case means we must prove that

```
⟪(f.leftMulMapPreGNS (star a)).completion ↑x, ↑y⟫_ℂ =
  ⟪↑x, (f.leftMulMapPreGNS a).completion ↑y⟫_ℂ
```

or informally, that

$$\langle \overline{\pi}_f(a^*)(x), y \rangle = \langle x, \overline{\pi}_f(a)(y) \rangle,$$

where, due to `induction_on₂`, we have that $x, y \in A_f$ instead of in H_f.

At this point, we complete the proof by writing

```
simp [mul_assoc, preGNS_inner_def]
```

which when inspected, becomes

```
simp only [completion_apply_coe,
leftMulMapPreGNS_apply,
  inner_coe, preGNS_inner_def, ofPreGNS_toPreGNS,
  star_mul, star_star, mul_assoc]
```

If you wish to inspect each step of this proof, you can replace it with the following sequence of re-writes:

```
rw [completion_apply_coe, completion_apply_coe,
  leftMulMapPreGNS_apply, leftMulMapPreGNS_apply,
  inner_coe, inner_coe,
  preGNS_inner_def, preGNS_inner_def,
  ofPreGNS_toPreGNS, ofPreGNS_toPreGNS,
  star_mul, star_star, mul_assoc]
```

In this case, this sequence of re-writes can give us a pretty good idea of what `simp` is doing. If we follow this sequence of re-writes informally (ignoring the theorems about type coercions), we can see that it simplifies the goal

like so

$$
\begin{aligned}
\langle \overline{\pi}_f(a^*)(x), y\rangle &= \langle x, \overline{\pi}_f(a)(y)\rangle && \\
\langle \pi_f(a^*)(x), y\rangle &= \langle x, \overline{\pi}_f(a)(y)\rangle && \text{by } \texttt{completion\_apply\_coe} \\
\langle \pi_f(a^*)(x), y\rangle &= \langle x, \pi_f(a)(y)\rangle && \text{by } \texttt{completion\_apply\_coe} \\
\langle a^*x, y\rangle &= \langle x, \pi_f(a)(y)\rangle && \text{by } \texttt{leftMulMapPreGNS\_apply} \\
\langle a^*x, y\rangle &= \langle x, ay\rangle && \text{by } \texttt{leftMulMapPreGNS\_apply} \\
f((a^*x)^*y) &= \langle x, ay\rangle && \text{by } \texttt{preGNS\_inner\_def} \\
f((a^*x)^*y) &= f(x^*(ay)) && \text{by } \texttt{preGNS\_inner\_def} \\
f(x^*(a^*)^*y) &= f(x^*(ay)) && \text{by } \texttt{star\_mul} \\
f(x^*ay) &= f(x^*(ay)) && \text{by } \texttt{star\_star} \\
f(x^*(ay)) &= f(x^*(ay)) && \text{by } \texttt{mul\_assoc},
\end{aligned}
$$

thus completing the proof.

5.2.3 A Few Additional Lemmas

Now that we have defined a map,

```
gnsNonUnitalStarAlgHom : A →⋆na[ℂ] f.GNS →L[ℂ] f.GNS
```

which informally, is $\overline{\pi}_f : A \to B(H_f)$, and we have proven that it is a *-homomorphism, we state and prove two additional minor lemmas to make working with `gnsNonUnitalStarAlgHom` easier.

The first lemma is

```
lemma gnsNonUnitalStarAlgHom_apply {a : A} :
    f.gnsNonUnitalStarAlgHom a =
      (f.leftMulMapPreGNS a).completion := rfl
```

which states that when we pass some $a \in A$ our *-homomorphism, it produces the same result as if we were passing a to the completion of the left-multiplication map that we defined previously. This is exactly the definition of our *-homomorphism, `gnsNonUnitalStarAlgHom`, so this lemma can be proven by `rfl`.

The second lemma is

```
@[simp]
lemma gnsNonUnitalStarAlgHom_apply_coe
```

```
    {a : A} {b : f.PreGNS} :
    f.gnsNonUnitalStarAlgHom a b =
      f.leftMulMapPreGNS a b := by
  simp [gnsNonUnitalStarAlgHom_apply]
```

which states that if we pass the same input $b \in A_f$ to the functions on either side of the equality in the previous lemma, both functions give the same result. This proof is completed by writing

```
simp [gnsNonUnitalStarAlgHom_apply]
```

which uses the theorems `gnsNonUnitalStarAlgHom_apply`, `completion_apply_coe`, and `leftMulMapPreGNS_apply`, which are, respectively, the prior lemma, the fact that $\pi_f(a) = \overline{\pi}_f(a)$, and the definition of `leftMulMapPreGNS`.

5.2.4 The Unital Case

Now that we have proven the more general non-unital case of the GNS construction, we can quickly prove the unital case.

We begin by stating our new assumptions

```
variable {A : Type*} [CStarAlgebra A] [PartialOrder A]
    [StarOrderedRing A] (f : A →ₚ[ℂ] ℂ)
```

which are identical to our original assumptions except for `CStarAlgebra` which replaces our `NonUnitalCStarAlgebra`.

Proving `map_one'`

Next, to prove that the *-homomorphism we defined maps 1 to 1 (that is, that $\overline{\pi}_f(1_A) = 1_{B(H_f)}$), when A is unital, we prove the following lemma:

```
@[simp]
private lemma gnsNonUnitalStarAlgHom_map_one :
    f.gnsNonUnitalStarAlgHom 1 = 1 := by
  ext b
  induction b using Completion.induction_on with
  | hp => apply isClosed_eq <;> fun_prop
  | ih b => simp [gnsNonUnitalStarAlgHom]
```

As you can see, the proof is nearly identical to most of the prior proofs of properties of our *-homomorphism, $\overline{\pi}_f$.

The first step is to use `ext b` to pass an arbitrary $b \in H_f$ to the functions on both sides of our expression. Because in this proof, our goal is to prove that

$$\overline{\pi}_f(1_A) = 1_{B(H_f)},$$

using `ext b` changes to goal to

$$\overline{\pi}_f(1_A)(b) = 1_{B(H_f)}(b),$$

which Lean expresses as

```
(f.gnsNonUnitalStarAlgHom 1) b = 1 b
```

Note that $1_{B(H_f)} : H_f \to H_f$ is the identity function.

The next step of the proof is to use the same induction principle as used in the prior proofs, `induction_on` (Theorem 3.1.11).

In this case, the first step of using that induction principle is to prove

```
IsClosed {a | (f.gnsNonUnitalStarAlgHom 1) a = 1 a}
```

which, when we apply `isClosed_eq` means we must prove that

```
Continuous ⇑(f.gnsNonUnitalStarAlgHom 1)
```

and

```
Continuous ⇑1
```

Informally, this means that we have to prove $\overline{\pi}_f(1_A)$ is continuous and that $1_{B(H_f)}$ (the identity function) is continuous.

The tactic `fun_prop` is able to easily prove both of these goals because we already proved the continuity of $\overline{\pi}_f(a)$, and the identity function is always continuous. Thus, this step of the proof is completed.

The final step is then to prove that

```
(f.gnsNonUnitalStarAlgHom 1) ↑b = 1 ↑b
```

or informally, that for $b \in A_f$,

$$\overline{\pi}_f(1_A)(b) = 1_{B(H_f)}(b).$$

To prove this, we simply write

```
simp [gnsNonUnitalStarAlgHom]
```

When we inspect the `simp` we see that we can re-write it as

```
simp only [gnsNonUnitalStarAlgHom, MonoidHom.coe_id,
  NonUnitalStarAlgHom.coe_mk', NonUnitalAlgHom.coe_mk,
  completion_apply_coe, leftMulMapPreGNS_apply,
  one_mul, toPreGNS_ofPreGNS, one_apply]
```

so it evidently uses many theorems to complete this proof. However, like in the prior proofs, these theorems just apply definitions and type coercions to complete the proof in much the same manner as we completed this proof in section (5.1.3).

If you wish to inspect each step of the proof, you can replace it with

```
rw [gnsNonUnitalStarAlgHom,
  NonUnitalStarAlgHom.coe_mk',
  NonUnitalAlgHom.coe_mk, completion_apply_coe,
  leftMulMapPreGNS_apply, one_mul, toPreGNS_ofPreGNS,
  one_apply]
```

Note that this `rw` proof does not use all the same theorems as `simp`. Specifically, it does not use the theorem `MonoidHom.coe_id`. In fact, `simp` doesn't need that theorem either. That is, if you remove it from the `simp only` proof, it will still work. The normal `simp` proof likely uses that theorem for some intermediate step that can be avoided by the correct sequence of re-writes.

Defining the Unital *-Homomorphism

Now that we have proven that

```
f.gnsNonUnitalStarAlgHom 1 = 1
```

we can define our unital *-homomorphism by writing

```
@[simps]
noncomputable
def gnsStarAlgHom : A →⋆ₐ[ℂ] (f.GNS →L[ℂ] f.GNS) where
  __ := f.gnsNonUnitalStarAlgHom
  map_one' := by simp
  commutes' _ := by simp
  [Algebra.algebraMap_eq_smul_one]
```

The `__ := f.gnsNonUnitalStarAlgHom` states that `gnsStarAlgHom` should inherit any shared properties, including its definition, from `gnsNonUnitalStarAlgHom`.

Then, because we proved `map_one'` in the prior lemma, which we marked as `@[simp]`, we can prove `map_one'` here just by writing `simp`.

The final property, `commutes'` asks us to prove that for $c \in \mathbb{C}$,

$$\overline{\pi}_f(c1_A) = c1_{B(H_f)}.$$

When we have that A is unital and we have proven `map_one'` (that $\overline{\pi}_f(1_A) = 1_{B(H_f)}$), this is an immediate consequence of `map_smul'` (that $\overline{\pi}_f$ maps scalars). Thus, the proof can be completed by writing

```
simp [Algebra.algebraMap_eq_smul_one]
```

where the theorem `Algebra.algebraMap_eq_smul_one` states that Mathlib's abstract concept of `AlgebraMap` that maps scalars to elements of the algebra, is equivalent to just scaling the vector 1_A by that scalar.

5.3 Design Considerations & Development Process

5.3.1 The Unital Continuity Proof

The proof of the continuity of π_f / `leftMulMapPreGNS` in the prior sections was provided entirely by Loreaux (44). However, before he provided that version of the proof which worked for both the unital and non-unital cases, I had formalized a proof, following the one provided by Aguilar (14) and many other sources including (46), (1), and (45), for the unital case.

The code for that version of the formalization can be accessed via the link/citations for Loreaux's proof (44) or directly via this link. The unital version of the proof involves defining an additional positive linear functional (which is done implicitly in some sources) and using the fact that for a positive linear functional $g : A \to \mathbb{C}$, we have that $\|g\|_{\text{op}} = g(1_A)$, which naturally relies on there being a unit, $1_A \in A$. This version of the formalization that followed this proof was very long and less general, so it made sense to replace it with Loreaux's proof.

5.3.2 fun_prop and two variable induction

Two of the tricks used in the formalization explained in this chapter were the use of the tactic `fun_prop` and the use of an induction principle on two variable at once. Alongside the suggestions mentioned in (3.2), Loreaux made

two important suggestions that helped me simplify other parts of my code as well. His first suggestion was to use `Completion.induction_on₂` to use an induction principle as used in (5.2.2) on two variables at once. His second suggestion was to add the attribute `fun_prop` to the theorem `UniformSpace.Completion.continuous_map`, which was in a separate Mathlib file, to allow the tactic `fun_prop` to automatically infer the continuity of the functions being used. Both of these suggestions helped to made the code more concise.

5.4 Summary

In this chapter we completed the GNS construction by defining $\overline{\pi}_f : A \to B(H_f)$ and proving that it is a *-homomorphism. In Mathlib, this function is denoted

```
gnsNonUnitalStarAlgHom : A →⋆ₙₐ[ℂ] f.GNS →L[ℂ] f.GNS
```

In general this homomorphism is non-unital, but we also showed that when A is unital, $\overline{\pi}_f$ is unital, and the unital version of this *-homomorphism is denoted in Mathlib as

```
gnsStarAlgHom : A →⋆ₐ[ℂ] (f.GNS →L[ℂ] f.GNS)
```

This result is the core of the GNS construction, as it provides a way to construct a family of representations of a C^*-algebra, parametrized by a positive linear functional.

Chapter 6

Conclusion & Future Directions

This thesis provides an overview of the traditional method of doing the GNS construction, but focuses on a slightly modified method suggested by Loreaux (44). Because this alternative method is what I formalized and is what is currently in Mathlib, this thesis has focused on explaining the Mathlib method of the GNS construction, which, to the best of my knowledge, has not been described anywhere else.

This project was a small step in the process of formalizing the theory of C^*-algebras in Lean. The most obvious next step would be to formalize the Gelfand-Naimark theorem, but other possible directions may include generalizing from the GNS construction to Stinespring's dilation theorem (also called Stinespring's factorization theorem, or just Stinespring's theorem), which generalizes from positive linear functionals to completely positive operator-valued functions (completely positive maps) (47). This generalization would then provide a few other notable theorems as immediate corollaries (48) and potentially allow the GNS construction itself to be refactored as a corollary.

The GNS construction has significance beyond pure mathematics, as evidenced by the fact that the GNS construction is listed as number 51 on the list compiled by Marco David and Jens Palsberg, a PhD student in mathematical physics and UC Berkeley and Computer Science professor at UCLA, respectively, in their attempt to identify the "Top 100 Quantum Theorems" to formalize (49). Their list also includes the Gelfand-Naimark Theorem at 52 and Stinespring's Theorem at 54.

This shows that the formalization of the GNS construction is of interest to the quantum physics/computing communities in addition to the portion of the mathematics community interested in formalization. As formalization becomes more widely adopted within physics it will become ever more important to have a robust library of mathematics that is relevant to physics. Already, formalization has been used to reveal a significant error in a "widely cited" physics paper (50), which demonstrates the applicability of formalization to physics.

Working on this project showed me that adopting open source practices in mathematics can help not only to advance mathematical formalization, but also to advance mathematics itself. Without Mathlib, none of the high-profile, multi-collaborator projects mentioned in the introduction would be possible, and without the hundreds of people who have contributed to it, there would be no Mathlib. Mathlib provides the foundation for all major mathematics projects in Lean, both those that formalize known mathematics and those that develop or confirm new mathematics.

Mathlib and many of the projects supported by Mathlib involve very large numbers of contributors, and contributions to Mathlib and other formalization projects are significant and meaningful intellectual work. However, such contributions cannot often be made to conform neatly to the paradigm of publishing papers with small lists of authors, so the formalization community has had to re-think how credit should be allocated and contributions can be recognized.

In particular, many people want to ensure that formalization efforts which are well-designed for others to build upon are credited accordingly and that claiming credit for work which may technically prove a result, but which is worthless to future formalization projects is discouraged. As put by David Loeffler,

> We want to signal that getting results into Mathlib has the same kind of "community adoption and approval" status as publication in a peer-reviewed journal in the informal world – with the implication that just "having a proof that compiles on your computer" should correspond to "having a proof on your blackboard that you and your coauthors believe", i.e. a preliminary step in the right direction but not the important one. (51)

To this end, after public discussion by some members of the Lean community, a definition of a peer-reviewed formalization was drafted (52), which

emphasizes the importance of producing formalizations that others can understand and build upon. Although this definition has no "official" status imparted to it by any institution, it is nevertheless a useful way of thinking about the work involved in formalization. That is why I emphasize the importance of this formalization of the GNS construction being included in Mathlib and I take care to mention some of the not-strictly-mathematical aspects of the design and development process. Producing a technically correct proof of an already well-established mathematical result is not the purpose of formalization. Producing useful, clean, reusable formal mathematics is the goal, and any work to achieve that goal is inherently collaborative.

Appendix A

Glossary of Names/Identifiers

Object	Mathematical Name	Lean/Mathlib Name
The main C^*-algebra	A	`A`
The positive linear functional on A	$f : A \to \mathbb{C}$	`f : A →ₚ[ℂ] ℂ`
The Pre-GNS space generated by f	A_f	`f.PreGNS`
The GNS Hilbert space generated by f	H_f	`f.GNS`
The *-algebra of bounded operators on the Pre-GNS space	$B(A_f)$	`f.PreGNS →L[ℂ] f.PreGNS`
The map from A to the bounded operators on the Pre-GNS space	$\pi_f : A \to B(A_f)$	`leftMulMapPreGNS (a : A) : f.PreGNS →L[ℂ] f.PreGNS`
The $*$-homomorphism from A to the bounded operators on the GNS Hilbert space	$\overline{\pi}_f : A \to B(H_f)$	`gnsNonUnitalStarAlgHom : A →⋆ₙₐ[ℂ] f.GNS →L[ℂ] f.GNS`

Appendix B

Alternate Proofs of Mathlib Content

B.1 Alternate Proof of completion_leftMulMapPreGNS_map_smul

```
import
Mathlib.Analysis.CStarAlgebra.GelfandNaimarkSegal

open scoped ComplexOrder InnerProductSpace
open Complex ContinuousLinearMap UniformSpace
Completion PositiveLinearMap
variable {A : Type*} [NonUnitalCStarAlgebra A]
[PartialOrder A] [StarOrderedRing A] (f : A →ₚ[ℂ] ℂ)

lemma completion_leftMulMapPreGNS_map_smul (m : ℂ) (x
: A) :
    (f.leftMulMapPreGNS (m • x)).completion = m •
    (f.leftMulMapPreGNS x).completion := by
  ext a
  induction a using induction_on with
  | hp => apply isClosed_eq <;> fun_prop
  | ih a => simp [smul_mul_assoc]
```

B.2 Alternate Proof of `map_smul'` for `completion`

```
import Mathlib.Topology.Algebra.GroupCompletion
import Mathlib.Topology.Algebra.Module.LinearMap

namespace ContinuousLinearMap
open UniformSpace Completion

variable {α β : Type*} {R₁ R₂ : Type*} [UniformSpace
α] [AddCommGroup α] [IsUniformAddGroup α]
  [Semiring R₁] [Module R₁ α]
  [UniformContinuousConstSMul R₁ α] [Semiring R₂]
  [UniformSpace β]
  [AddCommGroup β] [IsUniformAddGroup β] [Module R₂ β]
  [UniformContinuousConstSMul R₂ β]
  {σ : R₁ →+* R₂}

/--
Lift a continuous semilinear map to a continuous
semilinear map between the
`UniformSpace.Completion`s of the spaces. This is
`UniformSpace.Completion.map` bundled as a
continuous linear map when the input is itself a
continuous linear map.
-/
noncomputable def completion (f : α →SL[σ] β) :
Completion α →SL[σ] Completion β where
  __ := f.toAddMonoidHom.completion f.continuous
  map_smul' r x := by
    induction x using induction_on with
    | hp =>
      exact isClosed_eq (continuous_map.comp <|
      continuous_const_smul r)
        (continuous_map.const_smul _)
    | ih x =>
      rw [← Completion.coe_smul, ZeroHom.toFun_eq_coe,
        AddMonoidHom.toZeroHom_coe,
        AddMonoidHom.completion_coe,
        AddMonoidHom.completion_coe,
```

```
      LinearMap.toAddMonoidHom_coe,
      LinearMap.map_smul_sl, coe_coe,
      ← Completion.coe_smul]
cont := continuous_map
```

Bibliography

[1] B. Aupetit, *A Primer on Spectral Theory*. Springer Science & Business Media, 2012.

[2] C. Bailey and GitHub Contributors, "Type checking in lean 4," https://ammkrn.github.io/type_checking_in_lean4/, 2025, accessed: 2025-12-07.

[3] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[4] N. Megill and D. A. Wheeler, *Metamath: A Computer Language for Mathematical Proofs*. Lulu. com, 2019.

[5] The Rocq Development Team, *The Rocq Prover Reference Manual*. The Rocq Development Team, 2025.

[6] Lean FRO, "About - Lean Lang," https://lean-lang.org/fro/about/, 2025, accessed: 2025-12-07.

[7] W. D. Heaven, "What's next for AI and math," https://www.technologyreview.com/2025/06/04/1117753/whats-next-for-ai-and-math/, 2025, accessed: 2025-12-07.

[8] Lean FRO, "Mathlib: A Foundation for Formal Mathematics Research and Verification," https://lean-lang.org/use-cases/mathlib/, 2025, accessed: 2025-12-07.

[9] T. Tao, "Lean4," https://terrytao.wordpress.com/tag/lean4/, 2025, accessed: 2025-12-07.

[10] K. Buzzard, "What is the Xena project?" https://xenaproject.wordpress.com/what-is-the-xena-project/, 2025, accessed: 2025-12-07.

[11] The mathlib Community, "The Lean Mathematical Library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New Orleans, LA, USA: ACM, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[12] leanprover-community, "Mathlib statistics," https://leanprover-community.github.io/mathlib_stats.html, 2025, accessed: 2025-12-07.

[13] ——, "Lean projects," https://leanprover-community.github.io/lean_projects.html, 2025, accessed: 2025-12-07.

[14] K. Aguilar, "Real and Functional Analysis I and II: The Limits of Integration," 2025.

[15] Various GitHub Contributors, "Mathlib.Analysis.InnerProductSpace.Defs," https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/InnerProductSpace/Defs.html#InnerProductSpace, 2025, accessed: 2025-12-08.

[16] W. Rudin, *Functional Analysis*. McGraw-Hill, 1991.

[17] M. Takesaki *et al.*, *Theory of Operator Algebras I*. Springer, 1979.

[18] I. Gelfand and M. Neumark, "On the imbedding of normed rings into the ring of operators in hilbert space," , vol. 12, no. 2, pp. 197–217, 1943.

[19] K. Aguilar, personal communication.

[20] I. E. Segal, "Irreducible representations of operator algebras," *Bulletin of the American Mathematical Society*, 1947.

[21] J. v. Neumann, "Die eindeutigkeit der schrödingerschen operatoren," *Mathematische Annalen*, vol. 104, no. 1, pp. 570–578, 1931.

[22] S. S. Horuzhy, *Introduction to algebraic quantum field theory*. Springer Science & Business Media, 2012.

[23] S. Sekhon, "C^*-algebras: The (Quantum) Path Less Traveled," 2021. [Online]. Available: https://arxiv.org/abs/2104.02038

[24] P. Massot, "Why formalize mathematics?" https://www.imo.universite-paris-saclay.fr/~patrick.massot/files/exposition/why_formalize.pdf, 2021.

[25] F. Tran Minh, L. Gonnord, and J. Narboux, "Proof Assistants for Teaching: a Survey," *Electronic Proceedings in Theoretical Computer Science*, vol. 419, p. 1–27, May 2025. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.419.1

[26] K. Buzzard and R. Taylor, "A Lean proof of Fermat's Last Theorem," Imperial College of London, Tech. Rep., 2025.

[27] ImperialCollegeLondon, "ImperialCollegeLondon/FLT: Ongoing Lean formalisation of the proof of Fermat's Last Theorem," https://github.com/ImperialCollegeLondon/FLT, 2025, accessed: 2025-11-14.

[28] The mathlib Community, "Completion of the Liquid Tensor Experiment," https://leanprover-community.github.io/blog/posts/lte-final/, 2022.

[29] leanprover-community, "leanprover-community/lean-liquid: Liquid Tensor Experiment," https://github.com/leanprover-community/lean-liquid, 2022, accessed: 2025-11-14.

[30] P. Massot, "PatrickMassot/leanblueprint: plasTeX plugin to build formalization blueprints," https://github.com/PatrickMassot/leanblueprint, 2025, accessed: 2025-11-14.

[31] M. Thum, "Formalizing Discrete Exterior Calculus in Lean," *N/A*, 2023.

[32] Project Numina, "LeanDex," https://leandex.projectnumina.ai/, 2025, accessed: 2025-12-08.

[33] LeanExplore, "Documentation - LeanExplore," https://www.leanexplore.com/docs, 2025, accessed: 2025-12-08.

[34] Y. Tao, H. Liu, S. Wang, and H. Xu, "Assisting Mathematical Formalization with A Learning-based Premise Retriever," 2025. [Online]. Available: https://arxiv.org/abs/2501.13959

[35] B. AI4M Team, "LeanSearch," https://leansearch.net/, 2025, accessed: 2025-12-08.

[36] J. Breitner, "Loogle," https://loogle.lean-lang.org/, 2025, accessed: 2025-12-08.

[37] Various GitHub Contributors, "Mathlib Index," https://leanprover-community.github.io/mathlib4_docs/index.html, 2025, accessed: 2025-12-08.

[38] Renaissance Philanthropy, "A Dataset of Modern Formalized Theorem Statements," https://www.renaissancephilanthropy.org/a-dataset-of-modern-formalized-theorem-statements, 2025, accessed: 2025-12-08.

[39] T. Hubert, R. Mehta, L. Sartran, M. Z. Horváth, G. Žužić, E. Wieser, A. Huang, J. Schrittwieser, Y. Schroecker, H. Masoom, O. Bertolli, T. Zahavy, A. Mandhane, J. Yung, I. Beloshapka, B. Ibarz, V. Veeriah, L. Yu, O. Nash, P. Lezeau, S. Mercuri, C. Sönne, B. Mehta, A. Davies, D. Zheng, F. Pedregosa, Y. Li, I. von Glehn, M. Rowland, S. Albanie, A. Velingker, S. Schmitt, E. Lockhart, E. Hughes, H. Michalewski, N. Sonnerat, D. Hassabis, P. Kohli, and D. Silver, "Olympiad-level formal mathematical reasoning with reinforcement learning," *Nature*, 2025.

[40] T. Achim, A. Best, A. Bietti, K. Der, M. Fédérico, S. Gukov, D. Halpern-Leistner, K. Henningsgard, Y. Kudryashov, A. Meiburg, M. Michelsen, R. Patterson, E. Rodriguez, L. Scharff, V. Shanker, V. Sicca, H. Sowrirajan, A. Swope, M. Tamas, V. Tenev, J. Thomm, H. Williams, and L. Wu, "Aristotle: IMO-level Automated Theorem Proving," 2025. [Online]. Available: https://arxiv.org/abs/2510.01346

[41] P. Song, K. Yang, and A. Anandkumar, "Lean Copilot: Large Language Models as Copilots for Theorem Proving in Lean," 2025. [Online]. Available: https://arxiv.org/abs/2404.12534

[42] J. Avigad, L. de Moura, S. Kong, and S. Ullrich, "Theorem Proving in Lean 4," https://leanprover.github.io/theorem_proving_in_lean4/, 2025, accessed: 2025-11-14.

[43] The Lean Developers, "The Lean Language Reference," https://lean-lang.org/doc/reference/latest/, 2025, accessed: 2025-11-14.

[44] G. Wickham, "Mathlib Pull Request #33116," https://github.com/leanprover-community/mathlib4/pull/33116, 2026, accessed: 2026-01-09.

[45] K. R. Davidson, *C*-Algebras by Example*. American Mathematical Soc., 1996, vol. 6.

[46] W. Arveson, *An Invitation to C*-Algebras*. Springer Science & Business Media, 1998, vol. 39.

[47] W. F. Stinespring, "Positive functions on C*-algebras," *Proceedings of the american mathematical society*, vol. 6, no. 2, pp. 211–216, 1955.

[48] V. Paulsen, *Completely Bounded Maps and Operator Algebras*. Cambridge University Press, 2002, vol. 78.

[49] M. David and J. Palsberg, "Top 100 Quantum Theorems," https://marcodavid.net/top100/, 2026.

[50] J. Tooby-Smith, "Formalizing the stability of the two Higgs doublet model potential into Lean: identifying an error in the literature," 2026. [Online]. Available: https://arxiv.org/abs/2603.08139

[51] D. Loeffler, "#general > The role of AI companies in large formalisation projects," https://leanprover.zulipchat.com/#narrow/channel/113488-general/topic/The.20role.20of.20AI.20companies.20in.20large.20formalisation.20projects/near/581040355, 2026.

[52] J. Tooby-Smith, "Definition of a peer-reviewed formalization," https://github.com/jstoobysmith/PeerReviewedFormalization, 2026.