Jugend forscht 2024

LEAN - Der Beweis stimmt!

Chiara Cimino, Otto-Hahn-Gymnasium Tuttlingen
Christian Krause, Gymnasium Ochsenhausen

Schülerforschungszentrum Südwürttemberg e.V.
Standorte Tuttlingen und Ochsenhausen

19. Februar 2024

Inhaltsverzeichnis

1 Motivation **1**
1.1 Was ist ein Beweisassistent? . . . 1
1.2 Der Beweisassistent Lean . . . 1
1.3 Das Paradoxon von Banach-Tarski . . . 2

2 Material und Methoden **2**
2.1 Entwicklungsumgebung . . . 2
2.2 Mathlib Bibliothek . . . 3
2.3 Versionsverwaltung . . . 3

3 Erste Ergebnisse und wie man mit Lean arbeitet **3**
3.1 Die Definition der Komplexen Zahlen in Lean . . . 3
3.2 Der „kleine Gauß“; Verifikation des Beweises mit Lean . . . 4
3.3 Ein erstes Lemma . . . 6

4 Das Banach-Tarski-Paradoxon in Lean **9**
4.1 Mathematischer Hintergrund mit Beweisskizze . . . 9
4.2 Umsetzung in Lean . . . 10

5 Ausblick **15**

6 Danksagung **16**

Quellenverzeichnis **16**

Abbildungsverzeichnis

1 Lean Logo [9] . . . 1
2 Anschauliche Darstellung der Hauptaussage von Banach-Tarski [8] . . . 2
3 Die sum_up_to Funktion in VS Code . . . 2
4 Enter Caption . . . 2
5 Test von `sum_up_to` . . . 5

1 Motivation

Die Forschung in der Mathematik ist eine aktive und sehr lebendige Welt. Weltweit wurden vom Fachbereich Mathematik seit 1996 insgesamt 4,5 Millionen Dokumente publiziert. Davon sind rund 300 000 alleine im Jahr 2022 entstanden. Das entspricht einem Anstieg von rund 10% im Vergleich zum Vorjahr [1]. All diese Aufsätze und ihre enthaltenden Beweise müssen selbstverständlich auch auf Korrektheit geprüft werden und mit den stetig steigenden jährlichen Veröffentlichungszahlen gibt es immer mehr zu kontrollieren. Dies ist schwierig, zumal ein auf einem fehlerhaften Resultat aufbauender Beweis ebenfalls ungültig ist. Da mathematische Beweise aber nichts anderes als formale Objekte sind, ist die Idee naheliegend, einen Beweisassistenten zu entwickeln, mit dessen Hilfe Beweise diverser mathematischer Theoreme, Lemmata o.ä. schnell formalisiert und zweifelsfrei verifiziert werden können.
Diese Beweisassistenten gibt es schon sehr lange, aber erst in jüngster Zeit, auch mit dem Aufkommen der KI, sind die Möglichkeiten geradezu explodiert und renommierteste Lehrstühle und Professoren sind involviert. „Formale Beweisassistenten ermöglichen eine Zusammenarbeit in wirklich großem Ausmaß“. Dieses Zitat stammt von dem Mathematiker und Fields-Medallienträger Terence Tao aus seinem Vortrag „Machine assisted proofs“ vom 3. Januar 2024. Es hat sich eine Lean-Community gebildet, die hier intensivst forscht und große Schritte nach vorne macht. Auch wenn wir noch sehr jung sind, wollen wir ein Teil dieser Lean-Community werden. Uns haben die Gedanken fasziniert und wir haben uns zum Ziel gesetzt einen Teil in der Mathlib Bibliothek der maschinell überprüften Beweise beizutragen und so an einer ganz kleinen Stelle als Autoren verewigt zu werden.

1.1 Was ist ein Beweisassistent?

Ein mathematischer Beweisassistent ist eine Software, die über eine Programmiersprache verfügt, welche es dem Benutzer erlaubt, mathematische Objekte zu definieren. Ausgehend von diesen Objekten können dann unter Verwendung der zugrunde liegenden Programmiersprache die zugehörigen Eigenschaften, inklusive Beweis, formalisiert werden. Der Beweisassistent kontrolliert ausgehend von seiner logischen Basis die Standhaftigkeit des Beweises und markiert gegebenenfalls Fehler. Diese Basis kann sich allerdings von Assistent zu Assistent unterscheiden.

1.2 Der Beweisassistent Lean

Seit 2013 hat sich zu den wenigen bereits existierenden mathematischen Beweisassistenten der Beweisassistent namens Lean gesellt. Das Prinzip von Lean wurde von Leonardo de Moura, damals Microsoft-Mitarbeiter, entwickelt. Mittlerweile ist Lean ein Open Source Project, dass von der Community weiterentwickelt wird.

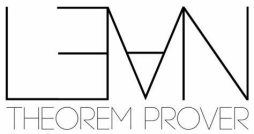

Abbildung 1: Lean Logo [9]

Die aktuellste Version von Lean (Lean4) ermöglicht die Kontrolle von Beweisen in fast allen Teilgebieten der Mathematik auf Basis der sogenannten Typentheorie. Die erste offizielle Version von Lean4 wurde erst im September 2023 veröffentlicht [2], was Lean zu einem der aktuellsten Beweisassistenten macht.
Bereits in Lean formalisierte mathematische Theoreme werden in einer allgemein zugänglichen Bibliothek, genannt Mathlib Bibliothek, gespeichert. Die Aussagen dieser Sätze können dann ohne einen weiteren Beweis vorausgesetzt werden und bei dem Formalisieren von neuen Sätzen inkl. Beweis verwendet werden. Aufgrund des jungen Alters von Lean umfasst diese Bibliothek aber noch lange nicht den aktuellen Stand der mathematischen Forschung und es sind viele Lücken vorzufinden. [5]
Das wohl bekannteste Beispiel ist die Formalisierung des von dem deutschen Fields-Medallienträger Peter Scholze aufgestellten Liquid Tensor Experiments in Lean. Durch Lean konnten mehrere kleinere Ungenauigkeiten in seinem Beweis gefunden und korrigiert werden. Außerdem konnten an einzelnen Stellen Vereinfachungen gefunden werden und im Juli 2022 wurde die Verifizierung des Liquid Tensor Experiments abgeschlossen [10]. Es ist erstaunlich, dass Lean diesen Beweis verifizieren konnte, da selbst Peter Scholze Zweifel hatte („I think nobody else has dared to look at the details of this, and so I still have some small lingering doubts“; Peter Scholze, Dezember 2020) und bis dahin fast niemand das Liquid Tensor Experiment vollständig durchdringen konnte.

Das ist auch der Grund, weshalb sich Mathematikerinnen und Mathematiker weltweit vernetzen, um weitere Sätze in Lean zu formalisieren. Unser Team ist, wie gesagt, nun eines der jüngsten Mitglieder dieser Community, denn uns hat diese Fortschrittlichkeit und das vielfältige Netzwerk der Lean-Community fasziniert.

1.3 Das Paradoxon von Banach-Tarski

Auf der Suche nach einem spannenden und alltagsnahen mathematischen Satz, der noch nicht in der sogenannten Mathlib Bibliothek ist, sind wir auf den Satz von Banach-Tarski gestoßen, auch bekannt als Banach-Tarski-Paradoxon.

Abbildung 2: Anschauliche Darstellung der Hauptaussage von Banach-Tarski [8]

Dieser besagt, dass die Einheitskugel in eine endliche Anzahl an Teilen zerlegt werden kann, welche neu angeordnet zwei Kopien der ursprünglichen Einheitskugel ergeben. Dies ist entgegen aller anfänglichen Intuition, da Verschiebungen und Drehungen im euklidischen Raum bekanntlich volumenerhaltend sind. Es ist aber zu erwähnen, dass dieser Volumenbegriff nur bei sog. messbaren Mengen sinnvoll ist. Im Satz von Banach-Tarski wird die Einheitskugel aber in nicht Lebesgue-messbare Mengen zerlegt. Die Existenz solcher nicht Lebesgue-messbaren Mengen folgt mit dem Auswahlaxiom.
Auf den Punkt gebracht folgt also mit dem Satz von Banach-Tarski unter bestimmten Rahmenbedingungen: „Aus eins mach zwei durch zerschneiden, drehen und wieder zusammen setzen“.

2 Material und Methoden

2.1 Entwicklungsumgebung

Als Entwicklungsumgebung verwenden wir VS Code in Kombination mit dem Lean4 Plugin [3]. Dieses implementiert einen Language Server, der die Lean Programme in Echtzeit kompiliert und auf Fehler überprüft. Außerdem werden in einem Fenster neben dem Programm Informationen, z.B. zu dem aktuellen Theorem angezeigt.

Abbildung 3: Die sum_up_to Funktion in VS Code

In Abbildung 3 sieht man exemplarisch, wie das Lean4 VS Code Plugin bestimmte Syntax Elemente farblich hervorhebt. Außerdem wird das Ergebnis des Befehls `#eval sum_up_to 4` direkt berechnet und in dem Fenster rechts angezeigt.

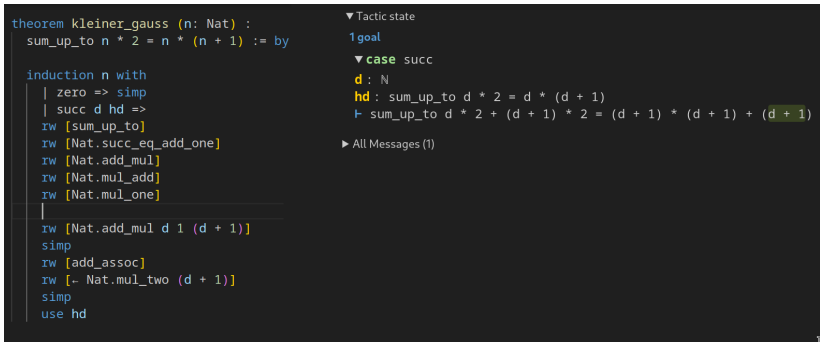

Abbildung 4: Enter Caption

Die Anzeige des Lean Plugins während eines Beweises ist beispielhaft in Abbildung 4 zu erkennen. Es wird der Zwischenstand des Beweises an der Position des Cursors angezeigt. Das ist vor allem bei Beweisen, die mit Taktiken arbeiten, hilfreich.

2.2 Mathlib Bibliothek

Eines der großen Ziele von Lean ist es, alle Beweise der Mathematik zu formalisieren und so zu überprüfen. Davon ist die Lean-Community zwar noch weit entfernt, aber es sind bereits viele grundlegende mathematische Theoreme und Lemmata aus vielen verschiedenen Bereichen in Lean implementiert. Diese Theoreme und Lemmata sind in der Mathlib [7] katalogisiert und können so auch in darauf aufbauenden Beweisen verwendet werden. Die Mathlib wurde ursprünglich in einer älteren Lean3-Version verfasst, ist aber mittlerweile vollständing in Lean4 übersetzt. Außerdem ist sie frei zugänglich, d. h. jeder kann sie in eigenen Lean Projekten verwenden und auch Erweiterungsvorschläge machen.

2.3 Versionsverwaltung

Wir speichern die Versionen von unseren Lean Programmen in einem Git Repository, das auch auf Github zu finden ist. Git ist der Standart für Versionsverwaltung von größeren Informatikprojekten, die Mathlib verwendet ebenfalls git und ist auf Github hochgeladen.

3 Erste Ergebnisse und wie man mit Lean arbeitet

Bevor wir uns an das Leanen des Banach-Tarski-Paradoxons machten, fangen wir mit kleineren Beispielen an; zunächst mal, um die Arbeitsweisen von Lean zu verstehen. Diese von uns selbst „geleanten“ Beispiele wollen wir dem Leser in den folgenden Unterkapiteln vorstellen, damit man sich selber ein Bild machen kann.

3.1 Die Definition der Komplexen Zahlen in Lean

Zum besseren Verständnis dieses Kapitels und aller weiteren Aktivitäten mit Lean, haben wir mit live-lean für den Leser mit Internetzugang eine Möglichkeit geschaffen, die einzelnen Schritte über folgenden Link nachzuvollziehen: live-lean.complex.

Wir beginnen mit dem Definieren eines grundlegenden Konstrukts: Dem Konzept der komplexen Zahlen. Definitionen aufstellen zu können ist nicht nur wichtig, um die mathematischen Objekte, die man verwenden möchte, herzustellen, sondern erleichtert später auch das Formalisieren von mathematischen Sätzen.
Um nun eine komplexe Zahl zu konstruieren, geben wir zunächst folgendes in Lean ein:

```
import Mathlib.Data.Real.Basic
```

Mit dieser ersten Zeile unseres Dokuments importieren wir die `Real.Basic` Datei aus der Mathlib, in der die Grundlagen des Systems der reellen Zahlen definiert sind. Selbstverständlich kann man auch mehrere Dateien importieren, uns reicht für die Definition der komplexen Zahlen aber dieses.
Nachdem wir Lean die Datei vorgegeben haben, das wir referenzieren möchten, können wir mit der Definition anfangen. Hierfür müssen wir zuerst definieren, wie eine komplexe Zahl überhaupt auszusehen hat, was wir mit den folgenden drei Zeilen erreichen:

```
structure CC_complex where
  re : Real
  im : Real
```

In diesem Programmabschnitt wird mit dem `structure` Befehl eine neue Struktur mit dem Namen `CC_complex` definiert. Diese Struktur kann bestimmte Attribute enthalten und kann im nachfolgenden Programm weiter verwendet werden. Neben Strukturen kann man in Lean auch mathematische Sätze mit dem Keyword `theorem` oder Variablen mit dem Keyword `variable` erstellt werden. Jegliche andere Definitionen können mit `def` vorgenommen werden.
In den folgenden Zeilen wird der Realteil, gekennzeichnet mit `re` , und der Imaginärteil mit `im` definiert. Dabei bedeutet es nicht etwa, dass der Realteil und der Imaginärteil Element der reellen Zahlen sind.

```
  re : Real
  im : Real
```

Diese Notation wird uns in jeder Aktion begleiten und heißt, dass `re` und `im` *vom Typ* der reellen Zahlen sind. Hier begegnet uns zum ersten Mal in der Anwendung, dass Lean auf der Typentheorie aufgebaut ist.
Nach diesen wenigen Zeilen Code weiß Lean nun schon, wie eine komplexe Zahl auszusehen hat. Aktuell können wir aber nicht viel mit ihnen machen, weshalb wir im Folgenden exemplarisch die Addition und Multiplikation von komplexen Zahlen mit jeweiligem neutralem und inversem Element definieren.
Wir starten mit der Null der komplexen Zahlen, wofür wir folgende Definition verwenden:

```
instance : Zero CC_complex :=
    ⟨⟨0,0⟩⟩
```

Hier begegnen wir der Aktion `instance`, welche unsere Struktur mit einer danach definierten Eigenschaft ausstattet.
`⟨⟨0,0⟩⟩` veranschaulicht, dass wir es bei einer komplexen Zahl mit einer zweidimensionalen Zahl zu tun haben, wobei 0 die Null der reellen Zahlen kennzeichnet, da wir die Datei der reellen Zahlen aus der Mathlib importiert haben.
Anschließend dazu definieren wir uns nun die Addition von komplexen Zahlen:

```
instance : Add CC_complex :=
  ⟨ fun x y  =>  ⟨ x.re + y.re, x.im + y.im ⟩ ⟩
```

Diese definieren wir über eine Funktion, die zwei komplexe Zahlen, hier repräsentiert durch `x` und `y`, auf das gewünschte Ergebnis abbildet. Nach unserer obigen Struktur bezeichnet `x.re` und `y.re` den Realteil von `x` bzw. `y`. Analog haben wird hier den Imaginärteil von `x` und `y` dargestellt. Aufgrund der vorausgesetzten Eigenschaften der reellen Zahlen können wir die Addition auf dem Ring der reellen Zahlen übernehmen, weshalb wir `x.re` und `y.re` bzw. `x.im` und `y.im` problemlos miteinander addieren können.
Bzgl. der Addition fehlt uns nur noch ein inverses Element für alle komplexen Zahlen. Die Definition eines solchen Elements erhalten wir durch die folgenden zwei Zeilen:

```
instance : Neg CC_complex :=
  ⟨ fun x => ⟨ -x.re, -x.im⟩ ⟩
```

Analog wie bei der Definition der Addition erreichen wir das Erstellen eines additiven Inversen durch eine Funktion, wobei wir wieder die Eigenschaften des Rings der reellen Zahlen ausnutzen, indem wir die Existenz des inversen Elements bzgl. der Addition von `x.re` und `x.im` voraussetzen.
Die Eins der komplexen Zahlen erhalten wir wie folgt:

```
instance : One CC_complex :=
    ⟨⟨1,1⟩⟩
```

Hierbei gehen wir analog vor, wie bei der Definition der Null der komplexen Zahlen, mit der Ausnahme, dass wir statt 0 1 verwenden, was wieder die Eins der reellen Zahlen kennzeichnet.
Ebenso wie bei der Addition komplexer Zahlen definieren wir uns die Multiplikation eben dieser über eine Funktion:

```
instance : Mul CC_complex :=
  ⟨ fun x y  =>   ⟨ x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re⟩⟩
```

Nun fehlt uns noch das multiplikative Inverse, was wir durch die zwei folgenden Zeilen definieren:

```
noncomputable instance : Inv CC_complex :=
  ⟨ fun x => ⟨ x.re/(x.re^2 + x.im^2), -x.im/(x.re^2 + x.im^2)⟩⟩
```

Hiermit haben wir die Addition und Multiplikation auf unseren eigens definierten komplexen Zahlen definiert.

3.2 Der „kleine Gauß“; Verifikation des Beweises mit Lean

Auch hier besteht die Möglichkeit, zum besseren Verständnis des Beweises durch folgenden Link auf das original Lean-Dokument zuzugreifen: live-lean.kleiner.gauß

Da der wesentliche Part dieses Programms aber das Kontrollieren von Beweisen ist, dreht sich nun dieses Kapitel um das Codieren des Beweises einer der bekanntesten Formeln: Der Gaußschen Summenformel oder besser bekannt, als der „kleine Gauß“. Diese ermöglicht die Berechnung der Summe der ersten n natürlichen Zahlen.
Bevor es nun an unseren ersten Satz geht, benötigen wir aber vorab noch eine Definition, welche uns folgende Summe für $n \in \mathbb{N}_0$ definiert: $\sum_{k=0}^{n} k$. Diese Summe haben wir wie folgt definiert:

```
def CC_sum_up_to (n : Nat): ℕ :=
  match n with
    | 0 => 0
    | (Nat.succ n) => CC_sum_up_to n + Nat.succ n
```

Um die Summe allgemein definieren zu können, führen wir am Anfang unserer Definition eine Variable `n` ein, welche vom Typ $\mathbb{N}$ ist. Hierbei ist zu erwähnen, dass Lean 0 in diesem Fall als eine natürliche Zahl auffasst. Da wir das Ganze über eine Zuordnung definieren möchten, müssen wir selbstverständlich auch unseren Wertebereich angeben, was wir durch `: ℕ` erreichen. Dies heißt, dass unser Funktionswert ebenfalls vom Typ der natürlichen Zahlen ist.
Mit `match n with` machen wir Lean klar, dass es sich hier um eine Zuordnung handelt, wobei `n` unser Ausgangswert ist.
In den folgenden beiden Zeilen Code definieren wir die Abbildungsvorschrift induktiv. Da Lean 0 als natürliche Zahl auffasst, ist unser Induktionsanfang hier 0, statt 1.
Die letzte Zeile beinhaltet unseren Induktionsschritt. Bzgl. der Begriffsklärung ist zu sagen, dass `Nat.succ n` die Notation für die Zahl $n+1$ ist, wenn $n \in \mathbb{N}_0$ gilt.
Um zu kontrollieren, dass unsere Definition das Gewünschte liefert, können wir Lean für `n` eine bestimmte Zahl einsetzen lassen bspw. 5. Dann erhalten wir in unserer Kontrolleiste rechts folgende Statusmeldung:

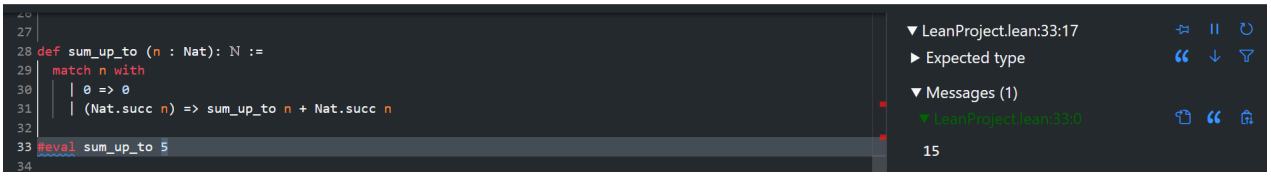

Abbildung 5: Test von `sum_up_to`

Hierfür verwenden wir die Funktion `eval`, welche hier 5 nach den definierten Zuordnungsvorschriften dem entsprechendem Wert zuordnet. Dass 15 das korrekte Ergebnis der Summe ist, kann man schnell nachrechnen. Nachdem wir nun $\sum_{k=0}^{n} k$ für ein $n \in \mathbb{N}_0$ definiert haben, können wir nun unser Theorem formalisieren:

```
theorem CC_kleiner_gauss (n: Nat) : CC_sum_up_to n * 2 = n * (n + 1) := by
```

Auf die Benennung des Theorems folgt sogleich die Voraussetzung `(n: Nat)`, was bedeutet, dass `n` den Typ der natürlichen Zahlen hat. Nach dem zweiten Doppelpunkt folgt unsere Behauptung:

$$\sum_{k=0}^{n} k = \frac{n \cdot (n+1)}{2} \textbf{ oder } 2 \cdot \sum_{k=0}^{n} k = n \cdot (n+1)$$

Diese Umformung erleichtert uns später den Beweis. Die obige Gleichheit formalisieren wir in Lean mithilfe von `CC_sum_up_to`. `:= by` kennzeichnet, dass die nachfolgende Zeile den Beweis der vorausgehenden Behauptung einläutet.
Sprechen wir schon von dem Beweis, so ist zu erwähnen, dass beim Beweisen in Lean verschiedene Taktiken verwendet werden. Die wohl von uns am meisten verwendete Taktik ist die „rewrite“ (kurz: `rw`) Taktik. Diese Taktik wird verwendet falls wir ein Beweis h eines Ausdrucks $X = Y$ gegeben haben. Dann erreichen wir durch `rw[h]`, dass alle X im noch zu zeigenden Ziel durch Y ausgetauscht werden. Die „rewrite“ Taktik kann daher insbesondere bei bereits in Lean bewiesenen Axiomen bzgl. Äquivalenzumformung verwendet werden. Dann erreichen wir mit `rw` und dem Namen des Axioms, dass das zu zeigende Ziel entsprechend des Axioms umgeschrieben wird. Zum Beispiel können wir einen Ausdruck der Form $(a+b) \cdot c$ in unserem Ziel mit `rw[add_mul]` zu $a \cdot c + b \cdot c$ umschreiben.
Eine weitere sehr essentielle Taktik ist die „simplify“ (kurz: `simp`) Taktik. Diese vereinfacht das noch zu zeigende Ziel unter Verwendung von `rw` so weit wie nur möglich. Wir können grundsätzlich nicht alle Umformungen, die wir mit `rw` durchführen, einfach durch `simp` ersetzen. `simp` kann nur sehr grundlegende Umformungen durchführen und ob man mit `simp` eine sinnvolle Umformung erhält ist situationsabhängig. Ein Versuch ist es aber meistens Wert.
Sollte man mit `simp` keinen Erfolg haben, so muss mit einer anderen Taktik weiter gearbeitet werden.

Abgesehen von `rw` und `simp` gibt es noch `exact`. Diese Taktik verwenden wir genau dann, wenn wir allgemein eine Aussage, nennen wir sie P, nach dem Vereinfachen unserer Behauptung noch zu zeigen haben, wobei ein Beweis h dieser Aussage P in Lean schon gegeben ist. `exact h` sagt Lean dann, dass es den Beweis h verwenden soll, womit dann der Beweis abgeschlossen wird.

Kommen wir nun also wieder auf unser ursprüngliches Theorem, der Gültigkeit der Gaußschen Summenformel für alle natürlichen Zahlen, inklusive der Null, zurück, so erhalten wir nur durch die Verwendung der oben vorgestellten Taktiken folgenden Beweis:

```
theorem CC_kleiner_gauss (n: Nat) : CC_sum_up_to n * 2 = n * (n + 1) := by
  induction n with
    | zero =>
    simp
    rw [CC_sum_up_to]
    | succ d hd =>
    rw [CC_sum_up_to]
    rw [Nat.succ_eq_add_one]
    rw [Nat.add_mul]
    rw [Nat.mul_add]
    rw [Nat.mul_one]
    rw [Nat.add_mul d 1 (d + 1)]
    simp
    rw [add_assoc]
    rw [← Nat.mul_two (d + 1)]
    simp
    exact hd
```

Wie man gut erkennen kann, haben wir den Beweis mithilfe einer Induktion über `n` aufgestellt. Wir haben daher zunächst Folgendes zu zeigen: `CC_sum_up_to Nat.zero * 2 = Nat.zero * (Nat.zero + 1)`. Die Taktik `simp` formt das Ziel zu `CC_sum_up_to 0 = 0` um. Nun können wir die Definition von `CC_sum_up_to` verwenden und damit den Beweis für `n=0` abschließen.
Anschließend gehen wir zum Induktionsschritt über, in dem wir voraussetzen, dass die Gleichheit für eine natürliche Zahl d gilt. Diese Voraussetzung betiteln wir hier mit `hd`. Dann zeigen wir, dass die Behauptung auch für $d+1$ wahr ist. Hierfür schreiben wir unser Ziel zunächst mit unserer eigenen Definition `CC_sum_up_to` um. Anschließend verwenden wir unter der Taktik `rw` eine Reihe an in der Mathlib Bibliothek verfügbaren bereits bewiesen Axiome. Die Namen dieser Axiome verraten schon, wie diese das Ziel umformen werden.
Damit dennoch keine Unklarheiten entstehen, sind in der folgenden Tabelle die Aussagen der einzelnen Axiome aufgeführt. a, b und c sind hier beliebige natürliche Zahlen.

`Nat.succ_eq_add_one`	`Nat.succ a = a+1`
`Nat.add_mul`	$(a+b)\cdot c = a\cdot c + b\cdot c$
`Nat.mul_add`	$a\cdot c + b\cdot c = (a+b)\cdot c$
`Nat.mul_one`	$a\cdot 1 = a$
`Nat.add_mul a 1 (a + 1)`	$(a+1)\cdot(a+1) = a\cdot(a+1) + 1\cdot(a+1)$
`add_assoc`	$(a+b)+c = a+b+c$
`← Nat.mul_two (a + 1)`	$a+1+(a+1) = (a+1)\cdot 2$

Nach der chronologischen Verwendung dieser Axiome und nach mehrfachem Vereinfachen mithilfe von `simp`, haben wir noch `CC_sum_up_to d * 2 = d * (d + 1)` zu zeigen übrig. Dies entspricht aber genau der Aussage von `hd`. Anwenden der Taktik `apply` in Kombination mit `hd` liefert dann das gewünschte Ergebnis.
Falls das Bedürfnis besteht, den mathematischen Beweis einzusehen, kann man dies über folgenden Link machen: kleiner-gauß

3.3 Ein erstes Lemma

Auch hier besteht die Möglichkeit, zum besseren Verständnis des Beweises durch folgenden Link auf das Original Lean-Dokument zuzugreifen: live-lean.äquivalenz

Nachdem wir den Beweis der Gaußschen Summenformel nun erfolgreich in Lean formalisiert haben, nehmen wir uns jetzt ein Lemma aus der Analysis 1 vor, um das Formalisieren in Lean auf das nächst höhere Level

zu heben. Wir haben uns hierbei für folgenden Satz entschieden, da wir metrische Räume äußerst spannend finden und dieser Satz sehr vielfältig verwendet werden kann.

Lemma 1. *Sei (X, d) ein metrischer Raum. Dann ist $M \subset X$ offen, genau dann, wenn M^c abgeschlossen ist.*

Um diesen Satz zu leanen, benötigen wir zunächst die hierfür notwendigen Software-Pakete. Nach dem Durchforsten der Mathlib Bibliothek stellen sich die folgenden Usepackages als zielführend heraus:

```
import Mathlib.Topology.MetricSpace.Basic
import Mathlib.Topology.MetricSpace.PseudoMetric
```

Bevor wir nun das Lemma formalisieren, sind vorab noch Definitionen aufzustellen und damit wir nicht jedes Mal einen metrischen Raum `X` und eine Teilmenge `M` von `X` wählen müssen, definieren wir uns diese zunächst als Variablen:

```
variable {X : Type*} [MetricSpace X] (M : Set X)
```

Diese Definition der Variablen wird in die kommenden Definitionen und in das eigentliche Theorem übertragen.
Um die Äquivalenz aus Lemma 1 zu zeigen, müssen wir als erstes aber in Lean definieren, was es bedeutet, wenn `M` offen bzw. abgeschlossen ist. Hierbei kann man die Definition von abgeschlossenen und offenen Mengen in metrischen Räumen gut wiedererkennen.

```
-- Definition der Offenheit
def CC_is_open_set : Prop :=
  ∀ x ∈ M, ∃ ε > 0, Metric.ball x ε ⊆ M

-- Definition der Abgeschlossenheit
def CC_is_closed_set : Prop :=
  is_open_set M\compl
```

`M\compl` bezeichnet dabei das Komplement von M.
Dass wir für die beiden Definitionen den Typ `Prop` vorgeben, hat den einfachen Grund, dass wir beide Mengeneigenschaften in der jeweiligen nächsten Zeile über Propositionen definieren. Die Verwendung dieses Typs ermöglicht es uns, die mit der Definition verbundenen Proposition in den kommenden als solche zu verwenden.
Vor dem endgültigen Start, Lemma 1 zu formalisieren, stellen wir uns zuvor noch ein Hilfslemma auf, welches besagt, dass `M=M\compl\compl` für ein `M : Set X` gilt.

```
theorem CC_compl_compl_eq_set (M : Set X) : M = M\compl\compl := by
```

Um dies zu zeigen, verwenden wir das in Lean schon formalisierte Lemma, welches auf unsere Metrik übertragen besagt, dass für `M N : Set X` die Äquivalenz `M\compl=N \Leftrightarrow M=N^\compl` gilt. Dieses Lemma trägt in Lean den Namen `eq_compl_comm.mpr` . In Kombination mit der Taktik `exact` und verbunden mit `rfl` , womit Lean das noch zu zeigende Ziel so weit wie möglich selbständig vereinfacht, Ist das Hilfslemma dann bewiesen.
Zusammengefasst sieht der Beweis in Lean dann wie folgt aus:

```
theorem CC_compl_compl_eq_set (M : Set x) : M = M\compl\compl := by
  exact eq_compl_comm.mpr rfl
```

Nach diesen Vorbereitungen können wir nun anfangen, Satz 1 zu formalisieren.

```
theorem CC_open_iff_complement_closed :
 CC_is_open_set M ↔ CC_is_closed_set M\compl := by
```

Da wir eine Äquivalenz zu zeigen haben, zerlegen wir diese zunächst in zwei Implikationen, die wir dann separat zeigen:

```
  rw [iff_def]
```

Nun einigen wir uns darauf, zuerst die Implikation `CC_is_open_set M -> CC_is_closed_set M\compl` zu beweisen:

```
  have h: CC_is_open_set M -> CC_is_closed_set M\compl
```

Hierbei darf man sich nicht von dem Taktiknamen `have` in die Irre führen lassen. Mit `have` formalisieren wir uns ein Zwischenziel `CC_is_open_set M -> CC_is_closed_set Mᶜ`, dass wir `h` nennen und welches wir in den folgenden Zeilen Code beweisen müssen. Die Taktik bedeutet nicht, dass wir die Implikation einfach voraussetzen.
Jetzt haben wir eine Implikation zu zeigen, wofür wir `CC_is_open_set M` voraussetzen, d.h. wir setzen voraus, dass `M` eine offene Menge ist. Dies wollen wir mit `h1` betiteln. Nun müssen wir noch zeigen, dass `Mᶜ` abgeschlossen ist und nach der Definition der Abgeschlossenheit, ist dies äquivalent dazu, dass `Mᶜᶜ` offen ist. Damit wir also zeigen können, dass `Mᶜᶜ` offen ist, müssen wir eine für-alle-Aussage zeigen. Daher wählen wir uns in Verbindung mit `h1` direkt auch ein `x` vom Typ `X`. Dies erreichen wir durch folgende Zeile:

```
intro h1 x
```

Wir wollen anschließend den Ausdruck `Mᶜᶜ` im Ziel zu `M` vereinfachen, weshalb wir nun das in Lean bereits formalisierte entsprechende Lemma `compl_compl` verwenden. Da wir dann immer noch eine Implikation `x ∈ M → ∃ ε > 0, Metric.ball x ε ⊆ M` zu zeigen haben, setzen wir `x ∈ M` unter dem Namen `hx` voraus:

```
rw [compl_compl]
intro hx
```

Jetzt können wir verwenden, dass `M` offen ist. Dies haben wir `h1` genannt:

```
apply h1
```

Da wir nun noch `x ∈ M` zu zeigen haben, was wir ebenfalls schon in unserer Voraussetzungsleiste als `hx` stehen haben, verwenden wir dieselbe Taktik mit `hx`:

```
apply hx
```

Nun haben wir h erfolgreich gezeigt. Da wir aber selbstverständlich noch die Implikation in die andere Richtung zeigen müssen, verwenden wir ein weiteres Mal die Taktik `have`:

```
have k: CC_is_closed_set Mᶜ -> CC_is_open_set M
```

Um diese Implikation zu zerlegen, gehen wir zunächst analog vor, wie bei dem Beweis von `h`. Wir setzen daher voraus, dieses Mal unter dem Namen `k1`, dass `Mᶜ` abgeschlossen ist. Da wir nun zeigen müssen, dass `M` offen ist, greifen wir erneut auf unsere Definition der Offenheit einer Menge zurück und wählen uns daher wieder ein `x` vom Typ `X`:

```
intro k1 x
```

Nun bleibt uns wieder die Implikation `x ∈ M → ∃ ε > 0, Metric.ball x ε ⊆ M` zu zeigen. Anders als bei dem Beweis von `h`, schreiben wir nun aber `M` zu `Mᶜᶜ` um. Hierfür verwenden wir unser selbst definiertes Lemma `CC_complᶜ_complᶜ_eqᶜ_set`:

```
rw [CC_compl_compl_eq_set M]
```

Damit wir nun die Implikation `x ∈ Mᶜᶜ → ∃ ε > 0, Metric.ball x ε ⊆ Mᶜᶜ` zeigen können, setzen wir `x ∈ Mᶜᶜ` voraus, was wir `kx` nennen wollen:

```
intro kx
```

Da wir jetzt zeigen müssen, dass `Mᶜᶜ` offen ist, verwenden wir `k1`, was besagt, dass `Mᶜ` abgeschlossen ist. Dann folgt automatisch mit unserer Definition der Abgeschlossenheit, dass `Mᶜᶜ` offen ist:

```
apply k1
```

Nun bleibt noch `x ∈ Mᶜᶜ` zu zeigen übrig, wofür wir `kx` verwenden:

```
apply kx
```

Damit haben wir auch `k` gezeigt. Jetzt sind wir nur noch einen kleinen Schritt vom Abschluss des Beweises entfernt.
Da wir unseren Satz in Lean beweisen, bleibt uns noch
`(CC_is_open_set M → CC_is_closed_set M\compl) ∧ (CC_is_closed_set M\compl → CC_is_open_set M)` zu zeigen. Dies erreichen wir durch die Verwendung von `h` auf der linken Seite sowie durch die Verwendung von `k` auf der rechten Seite:

```
  exact { left := h, right := k }
```

Im Gesamten sieht Lemma 1 inkl. Beweis dann wie folgt aus:

```
theorem CC_open_iff_complement_closed :
  CC_is_open_set M ↔ CC_is_closed_set M\compl := by

  rw [iff_def]
  have h: CC_is_open_set M -> CC_is_closed_set M\compl
  intro h1 x
  rw [compl_compl]
  intro hx
  apply h1
  apply hx
  have k: CC_is_closed_set M\compl -> CC_is_open_set M
  intro k1 x
  rw [CC_compl_compl_eq_set M]
  intro kx
  apply k1
  apply kx
  exact { left := h, right := k }
```

4 Das Banach-Tarski-Paradoxon in Lean

Nachdem wir uns und den Leser durch kleinere Einheiten mit den Prinzipien von Lean vertraut gemacht haben, wollen wir nun einen komplexeren Satz als Erste überhaupt leanen und dies in einem späteren Schritt in die Mathlib Bibliothek aufnehmen lassen.

4.1 Mathematischer Hintergrund mit Beweisskizze

Die bereits in Kapitel 1.3 informell vorgestellte Herangehensweise, wie man aus einer Kugel zwei mit dem selben Radius der Ausgangskugel erhält, wollen wir nun im mathematischen Licht betrachten.
Eine wesentliche Rolle spielt dabei eine freie Gruppe G, welche von zwei Erzeugern erzeugt wird. Diese sind in diesem Fall Drehwinkel bzw. die zugehörigen Drehmatrizen, welche wir im Folgenden A und B nennen. G soll dabei mit der Gruppenoperation der Konkatenation ausgestattet sein.
Des Weiteren sprechen wir im Folgenden von A^{-1} und B^{-1}. A^{-1} kennzeichnet dabei das inverse Element von A, d. h. wenn man einen Vektor $(x_1, x_2, x_3) \in \mathbb{R}^3$ zuerst um A dreht und anschließend um A^{-1}, so erhält man wieder den ursprünglichen Vektor (x_1, x_2, x_3). Analog definieren wir B^{-1}.
Da wir über G mit der Konkatenation operieren, können wir jedes Element von G als Wort mit den sog. Buchstaben A, B, A^{-1} und B^{-1} darstellen. Dementsprechend ist jedes Element in G eine Hintereinanderausführung der Drehungen A, B, A^{-1} und B^{-1}. Damit man sich dies besser vorstellen kann, machen wir zunächst ein Beispiel. Wir wählen hierfür ein Wort $W = [A^{-1}BAB^{-1}]$. Wollen wir also einen Vektor $(x_1, x_2, x_3) \in \mathbb{R}^3$ um W drehen, so ist dies äquivalent dazu, dass wir ihn zuerst um A^{-1}, dann um B, dann um A und anschließend um B^{-1} rotieren.
Da wir hier von einer freien Gruppe sprechen, ist jedes Wort in G gekürzt, d. h., dass in jedem Wort $[AA^{-1}]$, $[BB^{-1}]$, $[A^{-1}A]$ und $[B^{-1}B]$ raus gekürzt sind und diese Kombinationen somit in G nicht vorkommen.
Natürlich soll es uns auch möglich sein, einen Vektor aus dem $\mathbb{R}^3$ nicht zu drehen. Dieses „leere“ Wort notieren wir hier mit e.
Nun unterteilen wir alle Wörter in G in fünf Mengen. Die Menge $S(A)$ beinhaltet alle Wörter, die mit A enden, d. h. man dreht als letztes um A. Analog definieren wir uns auch die Mengen $S(A^{-1})$, $S(B)$ und $S(B^{-1})$. Die einelementige Menge $\{e\}$ bildet die letzte Menge. Damit gilt nun $G = S(A) \cup S(A^{-1}) \cup S(B) \cup S(B^{-1}) \cup \{e\}$
Die Idee ist nun, die Menge $S(A^{-1})$ um A und die Menge $S(B^{-1})$ um B zu „drehen“. Das soll heißen, dass wir an jedes Wort aus $S(A^{-1})$ A und dementsprechend an jedes Wort aus $S(B^{-1})$ B anhängen. Diese

„gedrehten“ Mengen notieren wir mit $S(A^{-1})A$ und $S(B^{-1})B$. Da sich aber $[A^{-1}A]$ und $[B^{-1}B]$ raus kürzen, entspricht nun die Menge $S(A^{-1})A$ der Vereinigung der Mengen $S(A^{-1})$, $S(B)$ und $S(B^{-1})$, da durch das Anhängen von A das letzte Element A^{-1} aller Elemente in $S(A^{-1})$ eliminiert wurde. $S(A)$ ist in diese Vereinigung nicht mit eingeschlossen, da die Wörter in G bekanntlich alle reduziert sind. Hätte ein Wort W aus $S(A^{-1})$ A als zweitletzten Eintrag, so würden sich $[AA^{-1}]$ raus kürzen und damit wäre W nicht mehr in $S(A^{-1})$ enthalten.
Entsprechend gleicht nun $S(B^{-1})B$ der Vereinigung der Mengen $S(B^{-1})$, $S(A)$ und $S(A^{-1})$.
Damit erhalten wir nun:

$$G = S(A) \cup S(A^{-1})A$$
$$G = S(B) \cup S(B^{-1})B$$

Hierbei ist zu erwähnen, dass in beiden Gleichungen $\{e\}$ nun überflüssig geworden ist, da e bereits in $S(A^{-1}A$ und $S(B^{-1})B$ enthalten ist. Das liegt daran, dass in $S(A^{-1})$ das Wort $[A^{-1}]$ enthalten ist, was wenn A angehängt wird sich zu e kürzt. Analog gilt dies auch bei $S(B^{-1})$
Insbesondere haben wir nun vier der ursprünglichen fünf Mengen zu zwei identischen Kopien der Gruppe zusammengefügt.
Mit zwei Kopien der freien Gruppe G ist es aber noch nicht getan. Wir wollen nun die Einheitskugel verdoppeln, weshalb wir die Drehungen aus G auf die Punkte der Einheitskugel anwenden. Hierfür beginnen wir mit der Einheitsspähre L.
Drehen wir nun einen Punkt $p \in L$ um alle Rotationen, die in G enthalten sind, so bezeichnen wir die Menge aller Punkte, die wir dadurch erhalten, als Bahn von p. Anschaulich kann man sich diese als eine kugelförmige Punktwolke vorstellen. Alle somit erhaltenen Bahnen sind paarweise disjunkt und die Vereinigung eben dieser ergibt wieder L.
Das Auswahlaxiom erlaubt uns nun, aus jeder erzeugten Bahn einen Repräsentanten auszuwählen. Wir bezeichnen die Menge aller Repräsentanten mit R. Anschließend konstruieren wir uns die Mengen $S(A)R$, $S(A^{-1})R$, $S(B)R$ und $S(B^{-1})R$. $S(A)R$ kennzeichnet hier die Menge aller Punkte, die wir erhalten, wenn wir alle in R enthaltenen Punkte jeweils um alle in $S(A)$ zusammengefassten Drehungen rotieren. Analog sind $S(A^{-1})R$, $S(B)R$ und $S(B^{-1})R$ definiert.
Grob gesagt können wir die Sphäre L nun unter Verwendung von $S(A)R$, $S(A^{-1})R$, $S(B)R$ und $S(B^{-1})R$ ähnlich wie bei unserer Gruppe G zerlegen und wieder zusammen setzen. Aus dieser Zerlegung der Sphäre folgt dann auch die Zerlegung der Kugel, in dem wir einfach jeden Punkt der Sphäre und jeden Punkt der Mengen $S(A)R$, $S(A^{-1})R$, $S(B)R$ und $S(B^{-1})R$ mit ihrem Ortsvektor ausstatten.
Somit haben wir also unsere Kugel verdoppelt!
Dass der formale Beweis etwas mehr ins Detail geht ist klar, jedoch führen wir diesen aus Platzgründen nicht vollständig auf. Wer sich für den vollständigen Beweis des Banach-Tarski-Paradoxons im dreidimensionalen Raum interessiert, der kann hier ([4]) eine Version einsehen. Damit der Leser aber das folgende Kapitel versteht, wollen wir auf die durch zwei unabhängige Winkel erzeugte Gruppe, mit der wir arbeiten näher eingehen. Das ein solches Winkelpaar existiert, zeigen wir konstruktiv. Hierfür wählen wir uns den Winkel $arccos(\frac{1}{3})$. Wir definieren uns nun A als die Drehmatrix zum Drehwinkel $arccos(\frac{1}{3})$ um die x-Achse und B als die Drehmatrix zum selben Winkel um die z-Achse.
Damit gilt: $A = \frac{1}{3}\begin{pmatrix} 3 & 0 & 0 \\ 0 & 1 & -2\sqrt{2} \\ 0 & 2\sqrt{2} & 1 \end{pmatrix}$, $B = \frac{1}{3}\begin{pmatrix} 1 & -2\sqrt{2} & 0 \\ 2\sqrt{2} & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$
Wir können zudem nicht einfach voraussetzen, dass die von A und B erzeugte Gruppe G frei ist. Dies müssen wir beweisen. Um diesen Beweis zu ermöglichen zeigen wir aber zunächst folgendes Lemma:

Lemma 2. *Sei $\rho : \mathbb{R}^3 \to \mathbb{R}^3$ ein Ausdruck in G von der Länge n in reduzierter Form. Dann hat $\rho(0,1,0)$ die Form $\frac{1}{3^n}(a\sqrt{2}, b, c\sqrt{2})$, wobei a, b und c ganze Zahlen sind.*

Das ρ in diesem Lemma ist somit eine Abbildung, die einem Punkt aus dem $\mathbb{R}^3$, den Punkt zuordnet, den man erhält, wenn man diesen Punkt durch ein zugehöriges Element W in G dreht. W ist dabei ein Wort von der Länge n und bereits reduziert.

4.2 Umsetzung in Lean

Wie bei dem „kleinen Gauß“ und der Definition der komplexen Zahlen in Lean, gibt es auch für diesen Satz einen Live-Lean-Link, um die einzelnen Schritte besser nachvollziehen zu können: live-lean.satz.3.1

In diesem Kapitel wollen wir Lemma 2 in Lean beweisen. Hierfür müssen wir vorab die Drehmatrizen A und B sowie deren Inverse in Lean definieren:

```
noncomputable section
def CC_matrix_a   : Matrix (Fin 3) (Fin 3) Real :=
                    ![![1, 0, 0; 0, 1/3, -2/3*Real.sqrt 2; 0, 2/3*Real.sqrt 2, 1/3]
def CC_matrix_a'  : Matrix (Fin 3) (Fin 3) Real :=
                    ![![1, 0, 0; 0, 1/3, 2/3*Real.sqrt 2; 0, -2/3*Real.sqrt 2, 1/3]
def CC_matrix_b   : Matrix (Fin 3) (Fin 3) Real :=
                    ![![1/3, -2/3*Real.sqrt 2, 0; (2/3*Real.sqrt 2), 1/3, 0; 0, 0, 1]
def CC_matrix_b'  : Matrix (Fin 3) (Fin 3) Real :=
                    ![![1/3, 2/3*Real.sqrt 2, 0; (-2/3*Real.sqrt 2), 1/3, 0; 0, 0, 1]
end noncomputable section
```

Diese Matrizen sind vom Typ `Matrix (Fin 3) (Fin 3) Real`, d. h., dass die definierten Matrizen 3×3-Matrizen mit reellen Einträgen sind. Der Programmteil, in dem die Matrizen definiert werden ist als `noncomputable section` deklariert. Das liegt daran, dass in der Definition irrationale Zahlen verwendet werden (`Real.sqrt 2`), mit denen Lean nicht konkret rechnen kann. D.h. es ist zwar trotzdem möglich diese Matrizen in Beweisen zu verwenden, aber es ist zum Beispiel nicht möglich, das Ergebnis einer konkreten Drehung um `CC_matrix_a` zu berechnen.
Daher können wir auch nicht durch Matrixmultiplikation zeigen, dass A' die inverse Matrix von A und B' die inverse Matrix von B ist. Damit wir mit der Invertierbarkeit von A und B trotzdem arbeiten können, definieren wir uns die entsprechenden Elemente in der linearen Gruppe der 3×3-Matrizen (`GL (Fin 3) Real`):

```
def CC_gl_a   : GL (Fin 3) Real :=
                Matrix.GeneralLinearGroup.mkOfDetNeZero CC_matrix_a CC_matrix_a_det_neq_zero
def CC_gl_a'  : GL (Fin 3) Real :=
                Matrix.GeneralLinearGroup.mkOfDetNeZero CC_matrix_a' CC_matrix_a'_det_neq_zero
def CC_gl_b   : GL (Fin 3) Real :=
                Matrix.GeneralLinearGroup.mkOfDetNeZero CC_matrix_b CC_matrix_b_det_neq_zero
def CC_gl_b'  : GL (Fin 3) Real :=
                Matrix.GeneralLinearGroup.mkOfDetNeZero CC_matrix_b' CC_matrix_b'_det_neq_zero
```

Damit wir die entsprechenden Elemente in dieser linearen Gruppe überhaupt definieren können, benötigen wir aber vorab noch einen Beweis, dass A und B bzw. A' und B' invertierbar sind. Die Invertierbarkeit dieser Matrizen ist äquivalent dazu, dass ihre Determinante ungleich 0 ist. Dies zeigen wir für A in folgendem Theorem:

```
theorem CC_matrix_a_det_neq_zero : Matrix.det CC_matrix_a ≠ 0 := by
  rw [CC_matrix_a]
  rw [Matrix.det_fin_three]
  simp
  norm_num
  ring_nf
  simp
  rw [Real.sq_sqrt]
  norm_num
  norm_num
```

Der Beweis für die Invertierbarkeit von B, A' und B' sieht genau gleich aus, bis auf dass wir anstatt `CC_matrix_a` `CC_matrix_b` bzw. `CC_matrix_a'` bzw. `CC_matrix_b'` verwenden. Aus Platzgründen sparen wir diese Beweise daher aus. Anschließend verwenden wir die Funktion `GeneralLinearGroup.mkOfDetNeZero` in Kombination mit der jeweiligen ursprünglichen Matrix A,B,A' und B' und dem Beweis, dass diese invertierbar sind, um das entsprechende Element der linearen Gruppe zu erhalten. Das von A induzierte Element wird dabei als `CC_gl_a` bezeichnet und entsprechend kennzeichnet `CC_gl_b` das von B induzierte, `CC_gl_a'` das von A' induzierte und `CC_gl_b'` das von B' induzierte Element.
Wir wollen uns später aber immer noch auf die explizite Darstellung von A und B beziehen, weshalb wir die folgenden beiden Hilfslemmas definieren:

```
theorem CC_coe_gl_a_eq_matrix_a : ↑CC_gl_a = CC_matrix_a := by
  rfl

theorem CC_coe_gl_b_eq_matrix_b : ↑CC_gl_b = CC_matrix_b := by
  rfl
```

Der Pfeil nach oben symbolisiert hier eine Transformation vom Typ der Matrizen `CC_gl_a` und `CC_gl_b` zum Typ der Matrizen `CC_matrix_a` und `CC_matrix_b`. Aufgrund der Definition von `CC_gl_a` und `CC_gl_b` können wir die Gleichheit durch die `rfl` Taktik beweisen.
Damit sollen nun `CC_gl_a` und `CC_gl_b` die Erzeuger unserer Gruppe G sein. Um diese Gruppe aber zu definieren, benötigen wir in Lean vorab die Menge der Erzeuger:

```
def CC_erzeuger : Set (GL (Fin 3) Real) := {CC_gl_a, CC_gl_b}
```

Jetzt können wir nun unsere Gruppe G definieren.

```
def G := Subgroup.closure erzeuger
```

Um diese zu definieren, verwenden wir die `Subgroup.closure` Funktion. Mit dieser Funktion erhalten wir hier eine Untergruppe der linearen Gruppe der 3×3-Matrizen mit reellen Einträgen, die von unserer Menge `CC_erzeuger` erzeugt wird.
Anschließend definieren wir uns nun eine Rotation eines Vektors im $\mathbb{R}^3$ um ein Element von G, welche wir `CC_rotate` nennen wollen.

```
abbrev r_3 := Fin 3 -> ℝ
def CC_rotate (p : GL (Fin 3) Real) (vec : r_3) : r_3 :=
  (p : Matrix (Fin 3) (Fin 3) Real).vecMul vec
```

Damit wir nicht ganz so viel schreiben müssen, definieren wir uns mithilfe von `abbrev r_3` als Abkürzung für den Typ eines dreidimensionalen Vektors mit reellen Einträgen.
Unserer `CC_rotate` Funktion wird also eine Matrix vom Typ `GL (Fin 3) Real` gegeben, welche dann zunächst zum Typ `Matrix (Fin 3) (Fin 3) Real` (also der Typ aller 3×3-Matrizen mit reellen Einträgen) konvertiert wird und anschließend dann mithilfe der Matrix-Vektor-Multiplikationsfunktion in Lean von rechts an den Vektor dran multipliziert wird. Wir definieren dies separat, da eine Matrix-Vektor-Multiplikation mit Matrizen vom Typ `GL (Fin 3) Real` in der uns verfügbaren Bibliothek nicht verfügbar ist.

Kommen wir auf Lemma 2 zurück, so definieren wir uns vorab die Behauptung, dass $\rho(0,1,0)$ als $\frac{1}{3^n}(a\sqrt{2}, b, c\sqrt{2})$ dargestellt werden kann. Dies ermöglicht es uns später, den Beweis dieses Lemmas in Lean übersichtlicher zu gestalten.

```
def CC_a_b_c_vec (a b c : ℤ) (n : Nat) : r_3 :=
   ![1/3^n * a * Real.sqrt 2,1/3^n * b,1/3^n * c * Real.sqrt 2]
```

Damit wir ebenfalls den Vektor (0,1,0) in Lean nicht immer ausschreiben müssen, definieren wir uns diesen als `CC_zero_one_zero`.

```
def CC_zero_one_zero : r_3 := ![0,1,0]
```

Nun haben wir alles zusammen, was wir für das Aufstellen des Lemmas 2 benötigen:

```
theorem CC_lemma_3_1 (p: GL (Fin 3) Real) (h: p ∈ G):
       ∃ a b c : ℤ, ∃ n : ℕ, CC_rotate p CC_zero_one_zero = CC_a_b_c_vec a b c n
```

Um dieses Lemma zu beweisen verwenden wir das Theorem `Subgroup.closure_induction`, dass bereits in der Mathlib vorhanden ist. Unter Verwendung dieser Funktion müssen wir unser Theorem dann nur noch in vier Spezialfällen beweisen. Im ersten Fall zeigen wir, dass die Behauptung für die Drehung eines Vektors im $\mathbb{R}^3$ um `CC_gl_a` und `CC_gl_b` gilt. Anschließend zeigen wir das Ganze noch für die leere Drehung. Schließlich nehmen wir uns zwei Elemente x und y aus G, für die die Behauptung gilt und zeigen, dass die Behauptung auch für $x \cdot y$ gilt. Zum Schluss haben wir nur noch zu zeigen, dass wenn die Behauptung für ein Element x in G gilt, dann gilt die Behauptung auch für das Inverse von x. Haben wir diese vier Fälle gezeigt, so können wir mithilfe von `Subgroup.closure_induction` auf die allgemeine Gültigkeit der in Lemma 2 aufgeführten Darstellung von $\rho(0,1,0)$ schließen.
Bevor wir die Behauptung für den ersten Fall beweisen, müssen wir die Behauptung im Voraus einzeln für unsere beiden erzeugenden Matrizen `CC_gl_a` und `CC_gl_b` zeigen. Für `CC_gl_a` erreichen wir das durch folgenden Code:

```
theorem CC_case_a (x) (h: x = CC_gl_a): ∃ a b c : ℤ,
∃ n : ℕ, CC_rotate x CC_zero_one_zero
        = CC_a_b_c_vec a b c n := by
    rw [h]
    rw [CC_rotate]
    rw [CC_zero_one_zero]
    rw [CC_coe_gl_a_eq_matrix_a]
    rw [CC_matrix_a]
    use 0
    use 1
    use -2
    use 1
    rw [CC_a_b_c_vec]
    simp
    norm_num
```

Nachdem wir das Ziel mithilfe von `rw` mehrfach mit unseren eigens aufgestellten Definitionen umgeschrieben haben, setzen wir durch `use` konkrete Werte für a, b, c und n ein. Vereinfachen unseres noch zu zeigenden Ziels und die Verwendung von `norm_num` liefern dann das Ende des Beweises. `norm_num` ist dabei eine Taktik, mit der eine numerische Gleichheit gezeigt werden, indem Lean sie einfach nachrechnet.

Dasselbe führen wir analog für `CC_gl_b` durch, wobei wir natürlich die Werte, die wir für a, b, c und n einsetzen. Details können hier eingesehen werden: live-lean.case.b Nachdem wir die Behauptung für `CC_gl_a` und `CC_gl_b` gezeigt haben, beginnen wir nun mit dem Beweis des ersten Falls, dass die Behauptung für die Drehung um die Erzeuger gilt.

```
theorem CC_hs (x : GL (Fin 3) Real) (h: x ∈ CC_erzeuger):
∃ a b c : ℤ, ∃ n : ℕ, CC_rotate x CC_zero_one_zero = CC_a_b_c_vec a b c n := by
    cases h with
    | inl ha =>
        apply CC_case_a
        exact ha
    | inr hb =>
        apply CC_case_b
        exact hb
```

In diesem Beweis machen wir eine Fallunterscheidung bzgl. der Erzeuger und verwenden dann unsere vorher gezeigten Hilfslemmas `CC_case_a` und `CC_case_b` . Dem Leser kommt es vielleicht unnötig vor, dass wir zuerst die Einzelfälle zeigen, jedoch ist zu erwähnen, dass in Lean nicht immer alles direkt bewiesen werden kann, sondern oftmals Lemmas im Voraus aufgestellt und gezeigt werden müssen, die man, wenn man den Beweis auf Papier durchführt, nicht benötigt.
Anschließend zeigen wir, dass die Behauptung für den zweiten Fall, also für die leere Drehung, gilt. Für diese erhalten wir als Drehmatrix die 3×3-Einheitsmatrix (im folgenden Theorem durch `1` gekennzeichnet). Dieser Beweis verläuft so ähnlich wie der Beweis von `CC_case_a` bzw. `CC_case_b` :

```
theorem CC_h_one :∃ a b c : ℤ, ∃ n : ℕ, CC_rotate 1 CC_zero_one_zero = CC_a_b_c_vec a b c n := by
    rw [CC_rotate]
    use 0
    use 1
    use 0
    use 0
    rw [CC_a_b_c_vec]
    simp
    rw [CC_zero_one_zero]
```

Bevor wir mit dem Beweis von Lemma 2 weiter verfahren, führen wir nun eine allgemeine Darstellung eines Elements in G ein. Wir zeigen nun mithilfe von Lean, dass jedes Wort in G durch die nebenstehende Drehmatrix dargestellt werden kann, wobei a, b, c, d, e, f, g, h und i ganze Zahlen sind und n eine natürliche Zahl ist. Nun implementieren wir diese Matrix unter dem Namen `CC_general_word_form` in Lean, um uns später unnötige Schreibarbeit zu sparen. Weil in dieser Matrix irrationale Zahlen vorkommen, setzen wir ein `noncomputable` vor die Definition.

$$\frac{1}{3^n}\begin{pmatrix} a & b\sqrt{2} & c \\ d\sqrt{2} & e & f\sqrt{2} \\ g & h\sqrt{2} & i \end{pmatrix}$$

```
noncomputable def CC_general_word_form
(a b c d e f g h i: ℤ) (n: Nat): Matrix (Fin 3) (Fin 3) Real :=
  !![(a : Real) * (1/3 ^ n),b * Real.sqrt 2 * (1/3 ^ n), (c : Real) * (1/3 ^ n);
    d * Real.sqrt 2 * (1/3 ^ n), (e : Real) * (1/3 ^ n), f * Real.sqrt 2 * (1/3 ^ n);
    (g: Real) * (1/3 ^ n), h * Real.sqrt 2 * (1/3 ^ n), (i : Real) * (1/3 ^ n)]
```

Damit Lean die ganzen Zahlen a, b, c, d, e, f, g, h und i mit $\frac{1}{3^n}$ multiplizieren kann, müssen wir ihm zunächst vorgeben, dass a, b, c, d, e, f, g, h und i ebenfalls vom Typ der reellen Zahlen sind. Nun können wir das Theorem in Lean formalisieren, welches besagt, dass wir jedes Element in G in der `CC_general_word_form` schreiben können:

```
theorem CC_general_word_form_exists (x: GL (Fin 3) Real) (h: x ∈ G) :
  ∃ a b c d e f g h i n, x = CC_general_word_form a b c d e f g h i n := by
    sorry
```

Bis zum Zeitpunkt der Abgabe der Langfassung ist es uns leider nicht gelungen, dieses Theorem zu beweisen, wir arbeiten aber fleißig daran. Dementsprechend steht anstatt einem Beweis hier `sorry`, was es uns ermöglicht, die Aussage des Theroems ohne Beweis im weiteren Verlauf zu verwenden.
Vor dem Beweis des dritten Falls formulieren wir uns zudem ein weiteres Theorem. Wir möchten nun zeigen, dass wenn wir ein reduziertes Wort aus G rechts an einen beliebigen Vektor der Form `CC_a_b_c_vec` multiplizieren, das Produkt wiederum mithilfe von `CC_a_b_c_vec` dargestellt werden kann. Dies erreichen wir in Lean durch folgende Codesequenz:

```
theorem CC_vec_mul_abc_eq_abc (x : GL (Fin 3) Real) (h1: x ∈ G)
  (h: ∃ j k l : ℤ, ∃ i : ℕ, CC_rotate x CC_zero_one_zero = CC_a_b_c_vec j k l i)
  (a b c : ℤ) (n : Nat) :
  ∃ e f g : ℤ , ∃ m : Nat, Matrix.vecMul (CC_a_b_c_vec a b c n) x = CC_a_b_c_vec e f g m := by
  rw [CC_a_b_c_vec]
  rcases CC_general_word_form_exists x h1 with ⟨a1, b1, c1, d1, e1, f1, g1, h1, i1,n1, h2⟩
  rw [h2]
  rw [CC_general_word_form]
  simp
  use a * a1 + b * d1 + c * g1
  use a * b1 * 2 + c * h1 * 2 + b * e1
  use  a * c1 + b * f1 + c * i1
  use n + n1
  rw [CC_a_b_c_vec]
  ext hx
  fin_cases hx
  simp
  ring
  simp
  ring_nf
  simp
  norm_num
  ring
  norm_num
  ring
```

In diesem Beweis verwenden wir zum ersten Mal die Taktik `ring`. Mit dieser können Ausdrücke in kommutativen Ringen vereinfacht werden und es ist sinnvoll sie anzuwenden, wenn man eine Potenz vereinfachen möchte.
Ausgehend davon können wir jetzt den dritten Fall beweisen, dass wenn die Behauptung für zwei Elemente x und y in G gilt, sie auch für das Produkt $x \cdot y$ folgt:

```
theorem CC_h_mul (x : GL (Fin 3) Real) (hx: x ∈ G) (y: GL (Fin 3) Real) (hy: y ∈ G)
(h1:∃ a b c : ℤ, ∃ n : ℕ, CC_rotate x CC_zero_one_zero = CC_a_b_c_vec a b c n)
(h2 :∃ d e f : ℤ, ∃ m : ℕ, CC_rotate y CC_zero_one_zero = CC_a_b_c_vec d e f m):
∃ g h i : ℤ, ∃ o : ℕ, CC_rotate (x*y) CC_zero_one_zero = CC_a_b_c_vec g h i o := by
  rw [CC_rotate]
  rw [@Matrix.GeneralLinearGroup.coe_mul]
  rw [CC_zero_one_zero]
  rw [CC_rotate] at h1
  rw [CC_zero_one_zero] at h1
  rw [CC_rotate] at h2
  rw [CC_zero_one_zero] at h2
  rcases h1 with ⟨a, b, c, n, h1'⟩
  rcases h2 with ⟨e, f, g, m, h2'⟩
  rw [← @Matrix.vecMul_vecMul]
  rw [h1']
  apply CC_vec_mul_abc_eq_abc
  apply hy
  rw [CC_rotate]
  rw [CC_zero_one_zero]
  rw [h2']
  use e
  use f
  use g
  use m
```

Im letzten Fall müssen wir nun beweisen, dass wenn die Behauptung für ein x in G gilt, so muss sie auch für das Inverse von x (x^{-1}) gelten. Hierfür benötigen wir vorab ein Lemma, welches besagt, dass wenn wir eine Matrix der Form 4.2 rechts an den Vektor $(0, 1, 0)$ multiplizieren, können wir das Produkt mithilfe von `CC_a_b_c_vec` darstellen:

```
theorem CC_general_word_form_abc (a b c d e f g h i: ℤ) (n : Nat):
  ∃ l m o p, Matrix.vecMul CC_zero_one_zero (CC_general_word_form a b c d e f g h i n) =
    CC_a_b_c_vec l m o p := by
    rw [CC_general_word_form]
    use d
    use e
    use f
    use n
    rw [CC_a_b_c_vec]
    rw [CC_zero_one_zero]
    ext h1
    fin_cases h1
    simp
    ring
    simp
    ring
    simp
    ring
```

Zudem benötigen wir in dem Beweis des letzten Falls, dass wenn x in G ist, dann ist auch x^{-1} in G. Dies formulieren wir uns ebenfalls in einem Hilfslemma:

```
theorem CC_x_inv_in_g (x: GL (Fin 3) Real) (h: x ∈ G): x⁻¹ ∈ G := by
  exact Subgroup.inv_mem G h
```

Jetzt haben wir alles, um den letzten Fall zu beweisen:

```
theorem CC_h_inv (x : GL (Fin 3) Real) (hx: x ∈ G)
  (h1:∃ a b c : ℤ, ∃ n : ℕ, CC_rotate x CC_zero_one_zero = CC_a_b_c_vec a b c n):
  ∃ a b c n, CC_rotate (x⁻¹) CC_zero_one_zero = CC_a_b_c_vec a b c n := by
  rw [CC_rotate]
  rw [zero_one_zero]
  rw [CC_rotate] at h1
  rw [CC_zero_one_zero] at h1
  rcases h1 with ⟨a, b, c, n, h1'⟩
  simp
  rcases CC_general_word_form_exists (x)⁻¹(CC_x_inv_in_g x hx) with
  ⟨a1, b1, c1, d1, e1, f1, g1,hh1,  i1, n1, h2⟩
  rw [← @Matrix.coe_units_inv]
  rw [h2]
  apply CC_general_word_form_abc
```

Damit ist durch folgende Codesequenz Satz 2 bewiesen:

```
theorem CC_satz_3_1 (p: GL (Fin 3) Real) (h: p ∈ G):
       ∃ a b c : ℤ, ∃ n : ℕ, CC_rotate p CC_zero_one_zero = CC_a_b_c_vec a b c n:=
  Subgroup.closure_induction' CC_h_s CC_h_one CC_h_mul CC_h_inv h
```

5 Ausblick

Mit Satz 2 ist der Beweis des Banach-Tarski-Paradoxons selbstverständlich noch nicht abgeschlossen. Aber wir sind auf einem guten Weg und werden es schaffen. Unser Ziel ist es, im Rahmen dieses Projekts, die hierfür noch notwendigen Sätze und Lemmas zu leanen und so schlussendlich das Banach-Tarksi-Paradoxon in Gänze zu verifizieren. Den fertigen Beweis werden wir dann zur Mathlib beisteuern und so ein kleines Mosaiksteinchen in der Lean-Community sein.
Auch rein mathematisch, zuerst unabhängig von Lean, arbeiten wir daran, Banach-Tarski zu erweitern. Wir haben uns zum Ziel gesetzt, diesen Satz für beliebig viele Dimensionen zu verallgemeinern und zu beweisen. Natürlich werden wir auch diese eigenen Beweisteile am Ende mit Lean verifizieren.
Daneben wollen wir noch aktiver in der Lean-Community an sich werden und Mitstreitertreffen besuchen. Über das KIT aber auch die Universität Tübingen sind erste Kontakte hergestellt.

6 Danksagung

Ein herzliches Dankeschön an Dr. Jakob von Raumer vom KIT in Karlsruhe, der unsere Fragen zu der Funktionsweise von Lean und der Darstellung grundlegender mathematischer Objekte in Lean bereitwillig beantwortet hat. Er hat uns geholfen, den Überblick in der sehr unübersichtlichen Dokumentation von Lean 4 zu bewahren.

Ein besonderer Dank an unsere Betreuer Noa Bihlmaier, Student an der Universität Tübingen und Helmut Ruf, Lehrer in Tuttlingen. Durch Sie haben wir überhaupt erst von dieser spannenden Welt der Beweissassistenten erfahren. Während der ganzen Zeit haben sie durch regelmäßige Feedback-Runden unser Projekt strukturiert, unterstützt und uns immer wieder motiviert haben, am Ball zu bleiben.

Auch wenn wir aufgrund der Unterschiede unserer Wohnorte viel online gearbeitet haben, haben wir uns immer wieder in den Räumen des SFZ in Tuttlingen getroffen. Wir sind sehr dankbar, dass es diese Möglichkeit gibt.

Quellenverzeichnis

[1] https://www.scimagojr.com/worldreport.php?area=2600

[2] https://github.com/leanprover/lean4/releases/tag/v4.0.0

[3] https://github.com/leanprover/vscode-lean4

[4] https://math.uchicago.edu/~may/REU2014/REUPapers/Robinson.pdf
The Banach-Tarski Paradox, Avery Robinson, 09.01.2015

[5] https://en.wikipedia.org/wiki/Lean_(proof_assistant)

[6] https://lean-lang.org/theorem_proving_in_lean4/tactics.html

[7] https://github.com/leanprover-community/mathlib4/tree/master/Mathlib

[8] https://visualizingmath.tumblr.com/post/117472740306/the-banach-tarski-paradox-the-bana

[9] https://www.microsoft.com/en-us/research/project/lean/

[10] https://www.degruyter.com/document/doi/10.1515/dmvm-2022-0058/html?lang=de