Schulmathematik mit dem

LEAN THEOREM PROVER

Peter Pfaffelhuber
Sommersemester 2023

universität freiburg

11. April 2023

Schulmathematik mit dem Lean Theorem Prover

Inhaltsverzeichnis

0 Vorbereitung zur Nutzung des Skriptes

Dies sind die Notizen zu einem Kurs zum formalen Beweisen mit dem interaktiven Theorem-Prover Lean3 (im folgenden schreiben wir nur Lean) im Sommersemester 2023 an der Universität Freiburg. Um den Kurs sinnvoll durcharbeiten zu können, sind folgende technische Vorbereitungen zu treffen:

1. Lokale Installation von Lean und der dazugehörigen Tools: Folgen Sie bitte den Hinweisen auf `https://leanprover-community.github.io/get_started.html`.

2. Installation von `vscode`. Bitte befolgen Sie die Download-Hinweise auf `https://code.visualstudio.com/download`.

3. Installation des Repositories des Kurses: Navigieren Sie zu einem Ort, wo Sie die Kursunterlagen ablegen möchten und verwenden Sie
   `leanproject get https://github.com/pfaffelh/schulmathematik_mit_lean`
   Dies sollte die Kursunterlagen herunterladen. Anschließend finden Sie das Manuskript unter `Manuskript/skript.pdf`, und Sie können die Übungen mit `code schulmathematik_mit_lean` öffnen. Die Übungen befinden sich dabei in `src`. Wir empfehlen, dieses Verzeichnis zunächst zu kopieren, etwa nach `mysrc`. Andernfalls kann es sein, dass durch ein Update des Repositories die lokalen Dateien überschrieben werden. Um die Kursunterlagen auf den neuesten Stand zu bringen, geben Sie `git pull` im Verzeichnis `schulmathematik_mit_lean` ein.

1 Einleitung

Der Kurs hat einen Fokus auf das Lehramts-Studium Mathematik an Gymnasien und hat mindestens zwei Ziele:

- Erlernen der Techniken zum interaktiven, formalen Beweisen mit Hilfe der funktionalen Programmiersprache Lean: In den letzten Jahren haben in der Mathematik Bemühungen drastisch zugenommen, computergestützte Beweise zu führen. Während vor ein paar Jahrzehnten eher das konsequente Abarbeiten vieler Fälle dem Computer überlassen wurde, sind interaktive Theorem-Prover anders. Hier kann ein sehr kleiner Kern dazu verwendet werden, alle logischen Schlüsse eines mathematischen Beweises nachzuvollziehen oder interaktiv zu generieren. Der Computer berichtet dann interaktiv über den Fortschritt im Beweis und wann alle Schritte vollzogen wurden.

- Herstellung von Verbindungen zur Schulmathematik: Manchmal geht im Mathematik-Studium der Bezug zur Schulmathematik verloren. Dieser Kurs ist der Versuch, diesen einerseits wieder herzustellen, und auf dem Weg ein tieferes Verständnis für die bereits verinnerlichte Mathematik zu bekommen. Um einem Computer zu *erklären*, wie ein Beweis (oder eine Rechnung oder ein anderweitiges Argument) funktioniert, muss man ihn erstmal selbst sehr gut verstanden haben. Außerdem muss man den Beweis – zumindest wenn er ein paar Zeilen übersteigt – gut planen, damit die eingegebenen Befehle (die wir Taktiken nennen werden) zusammenpassen.

Formalisierung in der Mathematik

Obwohl Mathematik den Anspruch hat, sauber zu argumentieren, finden sich in mathematischen Veröffentlichungen Fehler. Oftmals sind diese nicht entscheidend für die Richtigkeit der Aussage, die zu beweisen war. Manchmal wird eine Voraussetzung vergessen, und manchmal gibt es auch echte Fehler.

Stellt ein Theorem Prover die Richtigkeit eines Beweises fest, so ist die Glaubwürdigkeit deutlich größer. Zwar muss man sich immer noch auf die Fehlerfreiheit des Kerns (also etwa 10000 Zeilen Code) der Programmiersprache verlassen, sonst jedoch nur noch darauf, dass man die zu beweisende Aussage auch versteht und richtig interpretiert.

Heute wächst die Anzahl an Aussagen, die formal bewiesen werden, immer noch deutlich langsamer als die Anzahl an Veröffentlichungen in der Mathematik. Andererseits gibt es mittlerweile eine Community des formalen Beweisens, die von der Zukunftsfähigkeit von Theorem Provers überzeugt ist.

Interactive Theorem Prover

Mittlerweile gibt es einige Theorem Prover. Wir werden hier Lean (von Microsoft Research) verwenden. Grund für diese Wahl ist vor allem, dass es hier die größte Anzahl an Mathematikern gibt, die die `mathlib`, also die mathematische Bibliothek, die auf formal bewiesenen Aussagen mit dem Theorem Prover besteht, weiterentwickeln.

Momentan steht der Wechsel von Lean3 zu Lean4 an. Da insbesondere noch nicht die gesamte `mathlib` in Lean4 zur Verfügung steht, werden wir Lean3 verwenden.

Zum Inhalt

Dieses Manuskript gliedert sich in drei Abschnitte. In Kapitel 2 werden wir die Mathematik besprechen, die wir in den Übungen formal beweisen werden. Dies wird mit einfachen logischen Schlüssen anfangen, und am Ende werden einige Aussagen der Schulmathematik formal bewiesen. Dieser Teil ist der einzige Teil, den man von vorne nach hinten entlang der Übungsaufgaben abarbeiten sollte. Kapitel 3 gibt nützliche Hinweise zu Lean und `vscode`, von der Installation, über die Syntax, bis hin zu verwendetetn GLeichheitsbegriffen. Im Kapitel 4 werden wir alle verwendeten Befehle (also die *Taktiken*) besprechen. Diese werden hier als Nachschlagewerk zur Verfügung gestellt, wobei in den Übungen jeweils darauf verwiesen wird, welche neuen Taktiken gerade zu erlernen sind.

2 Mathematik

2.1 Logik

Wir beginnen mit einfachen logischen Aussagen. Wir unterscheiden immer (wie auch in jedem mathematischen Theorem) zwischen den Hypothesen und der Aussage. Um unsere Hypothesen einzuführen, führen wir sie in allen `lean`-Dateien auf einmal mit `variables (P Q R S T: Prop)` ein. Für die Lean-Syntax bemerken wir, dass hier kein üblicher Doppelpfeil $\Longrightarrow$ verwendet wird, sondern ein einfacher `→`. Wir gehen hier folgende logische Schlüsse durch:

- Blatt 01-a:
  Die Aussage `⊢ P → Q` (d.h. aus `P` folgt `Q`) bedeutet ja, dass `Q` gilt, falls man annehmen darf, dass die Hypothese `P` richtig ist. Dieser Übergang von `⊢ P → Q` zur Hypothese `hP : P` mit Ziel `⊢ Q` erfolgt mittels `intro hP`. Mehrere `intro`-Befehle kann man mittels `intros h1 h2...` abkürzen.
  Gilt die Hypothese `hP : P`, und wir wollen `⊢ P` beweisen, so müssen wir ja nur `hP` auf das Ziel anwenden. Ist Ziel und Hypothese identisch, so geschieht dies mit `exact hP`. Etwas

allgemeiner sucht `assumption` alle Hypothesen danach durch, ob sie mit dem Ziel definitorisch gleich sind.

- Blatt 01-b:
  Will man `⊢ Q` beweisen, und weiß, dass `hPQ : P → Q` gilt, so genügt es, `⊢ P` zu beweisen (da mit `hPQ` daraus dann `⊢ Q` folgt). Mit `apply hPQ` wird in diesem Fall das Ziel nach `⊢ P` geändert.
  Hinter einer Äquivalenz-Aussage `⊢ P ↔ Q` stehen eigentlich die beiden Aussagen `⊢ P → Q` und `⊢ Q → P`. Mittels `split` wandelt man das Ziel `⊢ P ↔ Q` in zwei Ziele für die beiden Richtungen um.
  Die logische Verneinung wird in Lean mit `¬` notiert. Die Aussage `¬P` ist dabei definitorisch gleich `P → false`, wobei `false` für eine falsche Aussage steht.

- Blatt 01-c:

  exfalso

  Bei einem Beweis durch Widerspruch beweist man statt `⊢ P` die Aussage `⊢ ¬P → false`. (Dies ist logisch korrekt, da `P` genau dann wahr ist, wenn `¬P` auf einen Widerspruch, also eine falsche Aussage, führt.) Die Umwandlung des Goals auf diese Art und Weise erreicht man mit der Taktik `by_contra`.

- Blatt 01-d:
  Für *und*- bzw. *oder*-Verknüpfungen von Aussagen stellt Lean die üblichen Bezeichnungen `∧` bzw. `∨` zur Verfügung. Mit diesen Verbindungen verknüpfte Aussagen können sowohl in einer Hypothese als auch im Ziel vorkommen. Nun gibt es folgende vier Fälle:
  `⊢ P ∧ Q` Hier müssen also die beiden Aussagen `P` und `Q` bewiesen werden. Mit `split` werden genau diese beiden Ziele (mit denselben Voraussetzungen) erzeugt, also `⊢ P` und `⊢ Q`. Sind diese beiden nämlich gezeigt, ist offebar auch `⊢ P ∧ Q` gezeigt.
  `⊢ P ∨ Q` Um dies zu zeigen, genügt es ja, entweder `P` zu zeigen, oder `Q` zu zeigen. Im ersten Fall wird mit `left` das Ziel durch `⊢ P` ersetzt, mit `right` wird das Ziel mit `⊢ Q`.
  `h : P ∧ Q` Offenbar zerfällt die Hypothese `h` in zwei Hypothesen, die beide gelten müssen. Mittels `cases h with hP hQ` wird aus `h : P ∧ Q` zwei Hypothesen generiert, nämlich `hP : P` und leanstatehQ : Q.
  `h : P ∨ Q` Ähnlich wie im letzten Fall erzeugt `cases h with hP hQ` nun zwei neue Goals, nämlich eines bei dem `h : P ∨ Q` durch `hP : P` ersetzt wurde, und eines bei dem `h : P ∨ Q` durch `hQ : Q` ersetze wurde. Dies ist logisch in Ordnung, weil man ja so gerade die Fälle, bei denen `P` oder `Q` gelten, voneinander treffen kann.

Blatt 01

.

Wir müssen uns zunächst einmal die Syntax ansehen, in der Lean mathematische Aussage, sowie deren Beweise darstellt. Wir beginnen mit einer Übersetzungstabelle:

`P : Prop`	`P` ist eine Aussage, die wahr oder falsch sein kann.
`hP : P`	`P` ist wahr, und diese Aussage heißt `hP` bzw. `hP` ist ein Beweis für `P`.

In Lean unterscheiden wir zwischen Termen (die links von `:` stehen), und Typen (die rechts vom `:` stehen). Dies ist sehr ähnlich zu Mengen, die man ja gewohnt ist. Ein Term ist dann so etwas wie ein Element, und ein Typ ist eine Menge. Ein besonderer Typ ist `Prop`. Dieser umfasst alle Aussagen, die wahr oder falsch sein können.

2.2 Mengen

Seien nun

```
variables (X Y : Type) (f : X → Y) (S : set X) (T : set Y)
```

Wir wollen nun die Bildmenge $f(S)$ und das Urbild $f^{-1}(T)$ definieren, in Lean ist das

$$f(S) := \texttt{f '' S := \{y : Y | ∃ x : X, x ∈ S ∧ f x = y\}},$$
$$f^{-1}(T) := \texttt{f⁻¹' T := \{x : X | f x ∈ T\}}.$$

In der ersten Zeile bemerken wir, dass der Term `f(S)` für Lean keinen Sinn ergibt, weil `f` nur Terme vom Typ `X` verarbeiten kann, und `S` vom Typ `set X` ist. Deshalb verwenden wir die Notation $f(S) :=$ `f '' S`. Dies ist die Notation für die Bildmenge (`set.image`). Analog würde die Notation `f⁻¹(T)` in Lean keinen Sinn ergeben, da `x⁻¹` die Notation für `has_inv.inv x` ist, und Typ `α → α` hat. Anders gesagt muss `x⁻¹` denselben Typ haben wir `x`. Die Notation `f ⁻¹' T` ist für `set.preimage`.

3 Hinweise zu Lean und vscode

In Abschnitt 0 haben wir uns bereits mit der Installation von Lean und `vscode` befasst. Hier folgt eine kurze, unzusammenhängende Einführung.

3.1 Dependent type theory

Lean ist eine funktionale Programmiersprache (d.h. es besteht eigentlich nur aus Funktionen) und basiert auf der *dependent type theory*. Typen in Programmiersprachen wir etwa Python sind `bool`, `int`, `double` etc. Lean lebt davon, eigene Typen zu definieren und zu verwenden. Wir werden sehen imd Verlauf des Kurses sehen, dass man über die entstehenden Typen wie Mengen denken kann. Der Typ `ℕ` wird etwa die Menge der natürlichen Zahlen, und `ℝ` die Menge der reellen Zahlen sein. Allerdings steht `ℕ` in der Tat für eine unendliche Menge, die dadurch charakterisiert ist, dass sie `0` enthält, und wenn sie `n` enthält, so enthält sie auch den Nachfolger von `n` (der mitt `succ n` dargestellt wird). Entsprechend sind die reellen Zahlen durch eine Äquivalenzrelation auf Cauchy-Folgen definiert, was schon recht aufwändig ist. Typen können dabei von anderen Typen abhängen, und deshalb sprechen wir von *depoendent types*. Etwa ist der Raum $\mathbb{R}^n$ abhängig von der Dimension n. Wie wir sehen werden, sind mathematische Aussagen ebenfalls solche Typen.

```
example : P → P :=
begin
  sorry,
end
```

Zur Notation: Bei Mengen sind wir gewohnt, etwa $n \in \mathbb{N}$ zu schreiben, falls n eine natürliche Zahl ist. In der Typentheorie schreiben wir `n : ℕ` und sagen, dass `n` ein Term (Ausdruck) vom Typ `ℕ` ist. Etwas allgemeiner hat jeder Ausdruck einen Typ und bei der Einführung jedes Ausdrucks überprüft Lean dessen Typ.

Sehen wir uns den rechten Teil des vscode-Fensters an. Dieser heißt aktueller *proof state*.

3.2 Von Universen, Typen und Termen

In Lean gibt es drei Ebenen von Objelten: Universen, Typen und Terme. Wir befassen uns hier mit den letzten beiden. Von besonderem Interesse ist der Typ `Prop`, der aus Aussagen besteht, die wahr oder falsch sein können. Er umfasst mathematische Aussagen, also entweder die Hypothesen, oder das Goal dessen, was zu beweisen ist. Eine Hypothese in Lean hat die Form `hP : P`, was soviel sagt wie `P` gilt, und diese Aussage heißt `hP`. Es kann auch bedeuten, dass `P` gilt und `hP` ein Beweis von `P` ist. Dabei haben die Hypothesen hier Namen `P Q R S`, und die Namen der Hypothesen `hP hQ hR hS`. Alle Namen können beliebig sein. Weiter gibt es Hypothesen der Form `P → Q`, also die Aussage, dass aus `P` die Aussage `Q` folgt.

3.3 Gleichheit

In Lean gibt es drei Arten von Gleichheit:

- Syntaktische Gleichheit: Wenn zwei Terme Buchstabe für Buchstabe gleich sind, so sind sie syntaktisch gleich. Allerdings gibt es noch ein paar weitere Situationen, in denen zwei Terme syntaktisch gleich sind. Ist nämlich ein Term nur eine Abkürzung für den anderen (etwa ist `x=y` eine Abkürzung für `eq x y`), so sind diese beiden Terme syntaktisch gleich. Ebenfalls gleich sind Terme, bei denen global quantifizierte Variablen andere Buchstababen habe. Etwa sind `∀ x, ∃ y, f x y` und `∀ y, ∃ x, f y x` syntaktisch gleich.

- Definitorische Gleichheit: Manche Terme sind in Lean per Definition gleich. Für `x : ℕ` ist `x + 0` per Definition identisch zu `x`. Allerdings ist `0 + x` nicht definitorisch identisch zu `x`. Dies hat offenbar nur mit der internen Definition der Addition natürlicher Zahlen in Lean zu tun.

- Propositionelle Gleichheit: Falls es einen Beweis von `x = y` gibt, so heißen `x` und `y` und propositionell gleich. Analog heißen Terme `P` und `Q` propositionell gleich, wenn man `P ↔ Q` beweisen kann.

rw ist für syntaktische Gleichheit

Viele Taktiken arbeiten bis hin zur definitorischen Gleichheit.

xxx Extensionalität

Terms of Types

sorry-Taktik

Wir beginnen mit der Beschreibung zweier grundlegender Taktiken, nämlich `intro` und `exact`; bitte in Kapitel 4 nachlesen. Das denkbar einfachste Beispiel ist die Aussage `P → P`, d.h. aus `P` folgt `P`. In Lean sieht das dann so aus:

```
example : P → P :=
begin
```

```
  sorry,
end
```

Der *proof state* verändert sich je nachdem, wo der cursor innerhalb des `begin end`-Blockes steht. Ist der Cursor diirekt nach `begin`, so ist der *proof-state*

```
P : Prop
⊢ P → P
```

Hier ist wichtig zu wissen, dass hinter `⊢` die Behauptung steht, und alles darüber Hypothesen sind. (Im gezeigten Fall ist dies nur die Tatsache, dass `P` eine Behauptung/Proposition ist. Diese Darstellung entspricht also genau der Behauptung. Ist der Cursor nach dem `sorry`, so steht nun zwar **goals accomplished** 🎉, allerdings ist die `sorry`-Taktik nur da, um erst einmal unbewiesene Behauptungen ohne weitere Handlung beweisen zu können und es erfolgt eine Warnung in `vscode`. Löscht man das `sorry` und ersetzt es durch ein `intro hP`, so erhält man

```
P : Prop
hP : P
⊢ P
```

Wir haben also die Aussage `P → P` überführt in einen Zustand, bei dem wir `hP : P` annehmen, und `P` folgern müssen. Dies lässt sich nun leicht mittels `assumption,` lösen (bitte das Komma nicht vergessen), und es erscheint das gewünschte **goals accomplished** 🎉. Die `assumption`-Taktik such nach einer Hypothese, die identisch mit der Aussage ist und schließt den Beweis. Etwas anders ist es mit der `exact`-Taktik. Hier muss man wissen, welche Hypothese genau gemeint und, und kann hier mit `exact hP` den Beweis schließen

Klammerung: In Lean wird die Anwendung von Funktionen, oder die Anwendung von Hypothesen, meist ohne Klammern geschrieben, also etwa `f x` anstatt $f(x)$. Eine interne Klammerung erfolgt dabei immer nach rechts, d.h. `g f x` ist eigentlich `g (f x)`. Analog ist bei Aussagen `P → Q → R` zu lesen als `P → (Q → R)`.

3.4 Hinweise zu `vscode`

Hinweise zu vscode

Warnungen und Fehler

Eingabe von Sonderzeichen

Klammerung anzeigen lassen

(In `vscode` gibt man diese mit `\wedge` bzw. `\vee` ein.)

4 Taktiken

4.1 Cheatsheet

Binäre Operatoren wir *und* (`∧`), *oder* (`∨`), *Schnitt*mengen (`∩`) und *Vereinigung*smengen (`∪`) sind rechts-assoziativ, also z.B. `P ∧ Q ∧ R` := `P ∧ (Q ∧ R)`.

Proof state	**Kommando**	**Neuer proof state**
`⊢ P → Q`	`intro hP`	`hP : P` `⊢ Q`

`f : α → Prop`
`⊢ ∀ {x : α}, f x`
	`intro x`	`f: α → Prop`
		`x : α`
		`⊢ f x`

`h : P` `⊢ P`	`exact h`	**goals accomplished** 🎉
`h : P` `⊢ P`	`assumption`	**goals accomplished** 🎉
`h : P → Q` `⊢ Q`	`apply h`	`h : P → Q` `⊢ P`

`⊢ P → Q → R`	`intros hP hQ`	`hP : P`
		`hQ : Q`
		`⊢ R`

`⊢ P ∧ Q → P` [1]	`tauto` oder `tauto!`	**goals accomplished** 🎉
`⊢ true`	`triv`	**goals accomplished** 🎉
`h : P` `⊢ Q`	`exfalso`	`h : P` `⊢ false`
`⊢ P`	`by_contra h`	`h : ¬P` `⊢ false`

`⊢ P`	`by_cases h : Q`	`h : Q`
		`⊢ P`
		`h : ¬Q`
		`⊢ P`

`h : P ∧ Q`
`⊢ R`
	`cases h with hP hQ`	`hP : P`
		`hQ : Q`
		`⊢ R`

`h : P ∨ Q`
`⊢ R`
	`cases h with hP hQ`	`hP : P`
		`⊢ R`
		`hQ : Q`
		`⊢ R`

`h : false` `⊢ P`	`cases h`	**goals accomplished** 🎉
`⊢ P ∧ Q`	`split`	`⊢ P` `⊢ Q`

[1] ...oder ein anderes Statement, das mit Wahrheitstabellen lösbar ist.

Proof state	**Kommando**	**Neuer proof state**
`⊢ P ↔ Q`	`split`	`⊢ P → Q` `⊢ Q → P`
`⊢ P ↔ P` oder `⊢ P = P`	`refl`	**goals accomplished** 🎉
`h : P ↔ Q` `⊢ P`	`rw h`	`h : P ↔ Q` `⊢ Q`
`h : P ↔ Q` `⊢ Q`	`rw ← h`	`h : P ↔ Q` `⊢ P`
`h : P ↔ Q` `hP : P`	`rw h at hP`	`h : P ↔ Q` `hP : Q`
`h : P ↔ Q` `hQ : Q`	`rw ← h at hQ`	`h : P ↔ Q` `hQ : P`
`⊢ P ∨ Q`	`left`	`⊢ P`
`⊢ P ∨ Q`	`right`	`⊢ Q`
`h : ⊢ 2 + 2 = 4` [2]	`norm_num`	**goals accomplished** 🎉

`f : α → Prop`
`y : α`
`⊢ ∃ (x : α), f x`
	`use y`	`f : α → Prop`
		`y : α`
		`⊢ f y`

`x y : ℝ` `⊢ x + y = y + x` [3]	`ring`	**goals accomplished** 🎉
`⊢ P → Q`	`intro hP`	`hP : P` `⊢ Q`

`f : α → Prop`
`⊢ ∀ {x : α}, f x`
	`intro x`	`f: α → Prop`
		`x : α`
		`⊢ f x`

`h1 : a < b`
`h2 : b ≤ c`
`⊢ a < c` [4]
	`linarith`	**goals accomplished** 🎉

`h : P` `⊢ Q`	`clear h`	`⊢ Q`
`f : ℕ → Prop` `h : ∀ (n : ℕ), f n`	`specialize h 13`	`f: ℕ → Prop` `h : f 13`

[2] ...oder ein anderes Statement, das nur Rechnungen mit numerischen Werte beinhaltet.

[3] ...oder ein anderes Statement, das nur Rechenregeln von kommutativen Ringen verwendet. `ring` zieht Hypothesen nicht in Betracht.

[4] ...oder eine Aussage, die nur `<`, `≤`, `≠` oder `=` verwendet. `linarith` zieht Hypothesen in Betracht.

<table>
<tr><th>Proof state</th><th>Kommando</th><th>Neuer proof state</th></tr>
<tr><td>⊢ P</td><td></td><td>⊢ P</td></tr>
<tr><td>f : ℕ → ℕ → Prop
h : ∀ (n : ℕ),
∃ (m : ℕ), f n m</td><td>obtain ⟨ m, hm ⟩
:= h 27
=</td><td>f: ℕ → ℕ → Prop
h : ∀ (n : ℕ),
∃ (m : ℕ), f n m</td></tr>
<tr><td>have h1 : ∃ m,</td><td>m : ℕ
f 27 m, ...
cases h1 with m hm</td><td>hm: f 27 m
⊢ P</td></tr>
<tr><td>h1 : a < b
h2 : b < c
⊢ a < c</td><td>library_search</td><td>goals accomplished 🎉
Try this:
exact lt_trans h1 h2</td></tr>
<tr><td>hQ : Q
⊢ P ∧ Q</td><td>refine ⟨ _, hQ ⟩</td><td>hQ : Q
⊢ P</td></tr>
<tr><td>⊢ P ∨ Q → R</td><td>rintro (hP | hQ)
=
intro h,
cases h with hP hQ</td><td>hP : P
⊢ P
hQ : Q
⊢ Q</td></tr>
<tr><td>⊢ P ∧ Q → R</td><td>rintro ⟨ hP , hQ ⟩
=
intro h,
cases h with h1 h2</td><td>hP : P
hQ : Q
⊢ Q</td></tr>
<tr><td>h : P ∧ Q ∨ P ∧ R
⊢ P</td><td>rcases h with
(⟨hP1,hQ⟩|⟨hP2,hR⟩)</td><td>hP1 : P
hQ : Q
⊢ P
hP2 : P
hR : R
⊢ P</td></tr>
<tr><td>⊢ n + 0 = n [5]</td><td>simp</td><td>goals accomplished 🎉</td></tr>
<tr><td>h : n + 0 = m [5]
⊢ P</td><td>simp at h</td><td>h : n = m
⊢ P</td></tr>
</table>

xxx change xxx let

```
def f : ℕ → ℝ := λ n, n^2 + 3
```

[5] ...oder ein anderes Statement, das sich durch Äquivalenz-Aussagen der Bibliothek vereinfachen lassen.

`apply`

Proof state	**Kommando**	**Neuer proof state**
`h : P → Q` `⊢ Q`	`apply h`	`h : P → Q` `⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`assumption`

Proof state	**Kommando**	**Neuer proof state**
`h : P` `⊢ P`	`assumption`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`bycases`

Proof state	**Kommando**	**Neuer proof state**

`⊢ P`	`by_cases h : Q`	`h : Q`
		`⊢ P`
		`h : ¬Q`
		`⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`bycontra`

Proof state	**Kommando**	**Neuer proof state**
`⊢ P`	`by_contra h`	`h : ¬P` `⊢ false`

Zusammenfassung

Beispiele

Anmerkungen

`cases`

Proof state	Kommando	Neuer proof state

`h : P ∧ Q`
`⊢ R`
	`cases h with hP hQ`	`hP : P`
		`hQ : Q`
		`⊢ R`

`h : P ∨ Q`
`⊢ R`
	`cases h with hP hQ`	`hP : P`
		`⊢ R`
		`hQ : Q`
		`⊢ R`

`h : false` `⊢ P`	`cases h`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`change`

xxx add table

Zusammenfassung

Ändert eine Hypothese bzw. das Goal in eine Hypothese bzw. das Goal, das definitorisch gleich ist.

Beispiele

Ist etwa

```
f: α → β
s: set α
x: α
xs: x ∈ s
⊢ x ∈ f ⁻¹' (f '' s)
```

so ändert `change f x ∈ f '' s,` das Goal zu `⊢ f x ∈ f '' s`.

`change` funktioniert auch bei Hypothesen. Ist eine Hypothese `h : x ∈ f ⁻¹' (f '' s)`, dann ist nach `change f x ∈ f '' s at h` die Hypothese `h : f x ∈ f '' s`.

Anmerkungen

Da viele Taktiken sowieso auf definitorische Gleichheit testen, ist `change` oftmals nicht nötig.

`clear`

Proof state	**Kommando**	**Neuer proof state**
`h : P` `⊢ Q`	`clear h`	`⊢ Q`

Zusammenfassung

Beispiele

Anmerkungen

`exact`

Proof state	**Kommando**	**Neuer proof state**
`h : P` `⊢ P`	`exact h`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`exfalso`

Proof state	**Kommando**	**Neuer proof state**
`h : P` `⊢ Q`	`exfalso`	`h : P` `⊢ false`

Zusammenfassung

Beispiele

Anmerkungen

`have`

Proof state	Kommando	Neuer proof state

`have h1 : ∃ m,`	`m : ℕ`
	`f 27 m, ...`
	`cases h1 with m hm`
		`hm: f 27 m`
		`⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`intro`

Proof state	Kommando	Neuer proof state
`⊢ P → Q`	`intro hP`	`hP : P` `⊢ Q`

`f : α → Prop`
`⊢ ∀ {x : α}, f x`
	`intro x`	`f: α → Prop`
		`x : α`
		`⊢ f x`

Zusammenfassung

Beispiele

Anmerkungen

`intros`

Proof state	Kommando	Neuer proof state

`⊢ P → Q → R`	`intros hP hQ`	`hP : P`
		`hQ : Q`
		`⊢ R`

Zusammenfassung

Beispiele

Anmerkungen

`left`

Proof state	Kommando	Neuer proof state
`⊢ P ∨ Q`	`left`	`⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`library_search`

Proof state	Kommando	Neuer proof state

`h1 : a < b`
`h2 : b < c`
`⊢ a < c`
	`library_search`	**goals accomplished** 🎉
		`Try this:`
		`exact lt_trans h1 h2`

Zusammenfassung

Beispiele

Anmerkungen

`linarith`

Proof state	Kommando	Neuer proof state

`h1 : a < b`
`h2 : b ≤ c`
`⊢ a < c`[6]
	`linarith`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`norm_num`

Proof state	Kommando	Neuer proof state
`h : ⊢ 2 + 2 = 4` [7]	`norm_num`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`obtain`

Proof state	Kommando	Neuer proof state

`f : ℕ → ℕ → Prop`
`h : ∀ (n : ℕ),`
`∃ (m : ℕ), f n m`
	`obtain ⟨ m, hm ⟩`
	`:= h 27`
	`=`
		`f: ℕ → ℕ → Prop`
		`h : ∀ (n : ℕ),`
		`∃ (m : ℕ), f n m`

Zusammenfassung

Beispiele

Anmerkungen

`rcases`

Proof state	Kommando	Neuer proof state

`h : P ∧ Q ∨ P ∧ R`
`⊢ P`
	`rcases h with`
	`(⟨hP1,hQ⟩|⟨hP2,hR⟩)`
		`hP1 : P`
		`hQ : Q`
		`⊢ P`
		`hP2 : P`
		`hR : R`
		`⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`refine`

Proof state	Kommando	Neuer proof state
`hQ : Q` `⊢ P ∧ Q`	`refine ⟨ _, hQ ⟩`	`hQ : Q` `⊢ P`

Zusammenfassung

Die `refine`-Taktik ist wie `exact` mit Löchern. Etwas genauer: Wenn das Goal darin besteht, eine Kombination aus Hypothesen anzuwenden, so kann man das mittels `refine` machen und für jeden offene Term `_` schreiben. Dann erhält man jeden `_` als neues Ziel zurück (wobei solche mit definitorischer Gleichheit sofort gelöst werden).

Beispiele

Angenommen, folgendes ist zu zeigen:

```
f : ℝ → ℝ
x y ε : ℝ
hε : 0 < ε
⊢ ∃ ( δ : ℝ) (H : δ > 0), |f y - f x| < δ
```

Dann wird mit `refine ⟨ε^2, by nlinarith, _ ⟩` das neue Goal zu `⊢ |f y - f x| < ε ^ 2.` Hier haben wir die `nlinarith`-Taktik verwendet, um `ε^2 > 0` aus `0 < ε` zu beweisen.

Anmerkungen

`refl`

Proof state	Kommando	Neuer proof state
`⊢ P ↔ P` oder `⊢ P = P`	`refl`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`right`

Proof state	Kommando	Neuer proof state
`⊢ P ∨ Q`	`right`	`⊢ Q`

Zusammenfassung

Beispiele

Anmerkungen

`ring`

Proof state	Kommando	Neuer proof state
`x y : ℝ` `⊢ x + y = y + x` [8]	`ring`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`rintro`

Proof state	Kommando	Neuer proof state

`⊢ P ∨ Q → R`	`rintro ( hP | hQ )`
	=
	`intro h,`
	`cases h with hP hQ`
		`hP : P`
		`⊢ P`
		`hQ : Q`
		`⊢ Q`

`⊢ P ∧ Q → R`	`rintro ⟨ hP , hQ ⟩`
	=
	`intro h,`
	`cases h with h1 h2`
		`hP : P`
		`hQ : Q`
		`⊢ Q`

Zusammenfassung

Die `rintro`-Taktik wird dazu verwendet, mehrere `intro`- und `cases`-Taktiken in einer Zeile zu verarbeiten.

Beispiele

Für

```
hP : P
hQ : Q
hR : R
⊢ S
```

ist `intro hP, intro h, cases h with hQ hR,` identisch mit `rintro hP ⟨ hQ, hR ⟩,`
Für

```
⊢ P ∨ Q → R
```

ist `intro h, cases h with hP hQ` identisch mit `rintro (hP | hQ)`.

Anmerkungen

Hier können auch mehr als zwei `∨` in einem Schritt in Fälle aufgeteilt werden: Bei `A ∨ B ∨ C` werden mit `rintro (A | B | C)` drei Goals eingeführt.

Anmerkungen

`rw`

Proof state	Kommando	Neuer proof state
`h : P ↔ Q` `⊢ P`	`rw h`	`h : P ↔ Q` `⊢ Q`
`h : P ↔ Q` `⊢ Q`	`rw ← h`	`h : P ↔ Q` `⊢ P`
`h : P ↔ Q` `hP : P`	`rw h at hP`	`h : P ↔ Q` `hP : Q`
`h : P ↔ Q` `hQ : Q`	`rw ← h at hQ`	`h : P ↔ Q` `hQ : P`

Zusammenfassung

Mit `rw ← h` wird `rw` von rechts nach links angewendet.

Beispiele

Anmerkungen

`simp`

Proof state	Kommando	Neuer proof state
`⊢ n + 0 = n` [9]	`simp`	**goals accomplished** 🎉
`h : n + 0 = m` [5] `⊢ P`	`simp at h`	`h : n = m` `⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`specialize`

Proof state	Kommando	Neuer proof state

`f : ℕ → Prop`
`h : ∀ (n : ℕ), f n`
`⊢ P`
	`specialize h 13`	`f: ℕ → Prop`
		`h : f 13`
		`⊢ P`

Zusammenfassung

Beispiele

Anmerkungen

`split`

Proof state	Kommando	Neuer proof state
`⊢ P ∧ Q`	`split`	`⊢ P` `⊢ Q`
`⊢ P ↔ Q`	`split`	`⊢ P → Q` `⊢ Q → P`

Zusammenfassung

Beispiele

Anmerkungen

`tauto`

Proof state	Kommando	Neuer proof state
`⊢ P ∧ Q → P` [10]	`tauto` oder `tauto!`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`triv`

Proof state	Kommando	Neuer proof state
`⊢ true`	`triv`	**goals accomplished** 🎉

Zusammenfassung

Beispiele

Anmerkungen

`use`

Proof state	Kommando	Neuer proof state

`f : α → Prop`
`y : α`
`⊢ ∃ (x : α), f x`
	`use y`	`f : α → Prop`
		`y : α`
		`⊢ f y`

Zusammenfassung

Beispiele

Anmerkungen