Schulmathematik mit dem LEAN THEOREM PROVER

Peter Pfaffelhuber
Sommersemester 2023

universität freiburg

8. Mai 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: Aus falschem folgt Beliebiges ist eigentlich die Aussage `⊢ false → P`. Ist das aktuelle Ziel `⊢ P`, und wendet man die Aussage `⊢ false → P` mittels `apply` an, so ist das äquivalent zur Anwendung von `exfalso`.
  Die beiden Ausdrücke `false` und `true` stehen für zwei Aussagen, die falsch bzw. wahr sind. Also sollte `true` leicht beweisbar sein. Dies liefert die Taktik `triv`.
  Bei einem Beweis durch Widerspruch beweist man statt `⊢ P` die Aussage `⊢ ¬P → false` (was nach `intro h` zur Annahme `h : ¬P` und dem neuen Ziel `⊢ false` führt). 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` bzw. `by_contra h`.

- Blatt 01-d:
  Für *und*- bzw. *oder*-Verknüpfungen von Aussagen stellt Lean die üblichen Bezeichnungen $\wedge$ bzw. $\vee$ zur Verfügung. Mit diesen Verbindungen verknüpfte Aussagen können sowohl in einer Hypothese als auch im Ziel vorkommen. Nun gibt es folgende vier Fälle:
  $\vdash P \wedge Q$ Hier müssen also die beiden Aussagen P und Q bewiesen werden. Mit `split` werden genau diese beiden Ziele (mit denselben Voraussetzungen) erzeugt, also $\vdash P$ und $\vdash Q$. Sind diese beiden nämlich gezeigt, ist offebar auch $\vdash P \wedge Q$ gezeigt.
  $\vdash P \vee Q$ Um dies zu zeigen, genügt es ja, entweder P zu zeigen, oder Q zu zeigen. Im ersten Fall wird mit `left` das Ziel durch $\vdash P$ ersetzt, mit `right` wird das Ziel mit $\vdash Q$. $h : P \wedge Q$ Offenbar zerfällt die Hypothese h in zwei Hypothesen, die beide gelten müssen. Mittels `cases h with hP hQ` wird aus $h : P \wedge Q$ zwei Hypothesen generiert, nämlich $hP : P$ und leanstatehQ : Q.
  $h : P \vee Q$ Ähnlich wie im letzten Fall erzeugt `cases h with hP hQ` nun zwei neue Goals, nämlich eines bei dem $h : P \vee Q$ durch $hP : P$ ersetzt wurde, und eines bei dem $h : P \vee Q$ 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-e:
  Hier geht es um die Einführung neuer Hypothesen. Bei der `by_cases`-Taktik – angewandt auf eine Hypothese $h : P$ – werden alle Möglichkeiten durchgegangen, die P annehmen kann. Diese sind, dass P entweder `true` oder `false` ist. Mit `by_cases h : P` werden also zwei neue Ziele eingeführt, eines mit der Hypothese $h : P$ und eines mit der Hypothese $h : \neg P$.
  Eine sehr allgemeine Taktik ist **`have`**. Hier können beliebige Hypothesen formuliert werden, die zunächst gezeigt werden müssen.

- Blatt 01-f:
  Nun kommen wir zu abkürzenden Schreibweisen. Zunächst führen wir die abkürzenden Schreibweise $\langle hP, hQ, hR \rangle$ für die $\wedge$-Verknüpfung der Aussagen hP, hQ und hR. (Dies funktioniert ebenfalls mit nur zwei oder mehr als drei Hypothesen). Analog ist $(hP \mid hQ)$ eine Schreibweise für $hP \vee hQ$. Diese beiden Schreibweisen können ebenso verschachtelt werden. Die drei Taktiken, die wir hier besprechen, sind `rintros` für `intros` + `cases`, `rcases` für eine flexiblere Version von `cases`, bei der man die gerade eingeführten Schreibweisen verwenden kann, und `obtain` für `intros` + **`have`**.

- Blatt 01-g: Quantoren

Natürliche Zahlen
Definition
rw
calc
simp
Funktionsdefinition
2n = n+n
Induktion
congr?

3 Hinweise zu Lean und vscode

xxx naming convention
Klammern
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

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

Proof state	**Kommando**	**Neuer proof state**
		hQ : Q ⊢ R
h : false ⊢ P	`cases h`	**goals accomplished** 🎉
⊢ : P → false	`change ¬P`	⊢ ¬P
⊢ P ∧ Q	`split`	⊢ P ⊢ Q
⊢ 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	`linarith`	**goals accomplished** 🎉

[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.

<table>
<tr><th>Proof state</th><th>Kommando</th><th>Neuer proof state</th></tr>
<tr><td>h2 : b ≤ c
⊢ a < c [4]</td><td></td><td></td></tr>
<tr><td>h : P
⊢ Q</td><td>clear h</td><td>⊢ Q</td></tr>
<tr><td>f : ℕ → Prop
h : ∀ (n : ℕ), f n
⊢ P</td><td>specialize h 13</td><td>f: ℕ → Prop
h : f 13
⊢ 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>⊢ R</td><td>have h : P ↔ Q,</td><td>⊢ P ↔ Q
h : P ↔ Q
⊢ R</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>
</table>

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

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

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

xxx change xxx let

```
def f : ℕ → ℝ := λ n, n^2 + 3
```

`apply`

Zusammenfassung

Angenommen, das Goal ist $\vdash Q$, wobei bereits ein Beweis für $h : P \to Q$ zur Verfügung steht. Deshalb braucht man nur noch einen Beweis für $\vdash P$ zu finden. Diese Umwandlung passiert mit `apply h`.

Beispiele

Proof state	Kommando	Neuer proof state
$h : P \to Q$ $\vdash Q$	`apply h`	$h : P \to Q$ $\vdash P$
$h : \neg P$ $\vdash$ false	`apply h`	$h : \neg P$ $\vdash P$

Die Taktik `apply` wendet sich iterativ an. Das bedeutet: liefert `apply h` (identisch mit `refine h _`) keinen Fortschritt, so wird es mit `refine h _ _` versucht:

Proof state	Kommando	Neuer proof state
$h : P \to Q \to R$ $\vdash R$	`apply h`	$h : P \to Q \to R$ $\vdash P$ $h : P \to Q \to R$ $\vdash Q$

Anmerkungen

1. `apply` arbeitet bis hin zur definitorischen Gleichheit. Dies kann man im zweiten Beispiel sehen, da $\neg P$ per Definition gleich $P \to$ false ist:

2. `apply h` ist identisch zu `refine h _`, bzw. (wie im zweiten Beispiel oben) zu `refine h _ _`.

`assumption`

Zusammenfassung

Ist eine der Annahmen identisch mit dem Goal, dann schließt `assumption,` das Goal.

Beispiele

Proof state	Kommando	Neuer proof state
h : P ⊢ P	`assumption`	**goals accomplished** 🎉
h : ¬P ⊢ P → false	`assumption,`	**goals accomplished** 🎉

Anmerkungen

1. Wie andere Taktiken auch arbeitet `assumption` bis zur definitorischen Geichheit.
2. Hier ein Trick: Verwendet man als Abschluss einer Taktik ein Semicolon `;`, so wird die nachfolgende Taktik auf alle Goals angewendet:

Proof state	Kommando	Neuer proof state
hP : P hQ : Q ⊢ : P ∧ Q	`split; assumption,`	**goals accomplished** 🎉

`by_cases`

Zusammenfassung

Hat man einen Term `P : Prop` als Hypothese, so liefert `by_cases hP : P` zwei Goals. Beim ersten gibt es zusätzlich `hP : P`, beim zweiten `hP : ¬P`. Diese Taktik ist also identisch mit `have hP : P ∨ ¬ P, exact em P, cases hP,`. (Der zweite Ausdruck ist `em : ∀ (p : Prop), p ∨ ¬p`.

Beispiele

Proof state	Kommando	Neuer proof state
⊢ P	`by_cases h : Q`	h : Q ⊢ P h : ¬Q ⊢ P
x: bool ⊢ x = tt ∨ x = ff	`by_cases x=tt,`	x: bool h: x = tt ⊢ x = tt ∨ x = ff x: bool h: ¬x = tt ⊢ x = tt ∨ x = ff

Im zweiten Beispiel verwenden wir eine Variable vom Typ `bool` Diese ist folgendermaen definiert:

```
inductive bool : Type
| ff : bool
| tt : bool
```

Eine Bool'sche Variable hat also nur die Möglichkeiten `tt` (für `true`) und `ff` (für `false`).

Anmerkungen

1. Offenbar benutzt die `by_cases`-Taktik (genau wie `by_contradiction`, dass eine Aussage entweder wahr oder falsch ist. Dies ist auch bekannt unter dem Gesetz des ausgeschlossenen Dritten. In der Mathematik nennt man Beweise, die diese Regel nicht verwenden, konstruktiv.
2. Für Terme vom Typ `Prop` kann die Taktik `tauto` (bzw. `tauto!`) durch eine Wahrheitstabelle viele Schlüsse ziehen.

`by_contra`

Zusammenfassung

Die `by_contra`-Taktik stellt einen Beweis durch Widerspruch dar. Es wird deshalb angenommen (d.h. in eine Hypothese überführt), dass die Aussage (hinter ⊢) nicht wahr ist, und dies muss zu einem Widerspruch geführt werden, d.h. es muss ein Beweis von `false` gefunden werden.

Beispiele

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

Anmerkungen

Diese Taktik ist stärker als `exfalso`. Dort wird ja schließlich das Goal nur in `false` überführt, ohne eine neue Hypothese hinzuzufügen. Bei `by_contra` ist das neue Goal zwar auch `false`, aber es gibt noch eine neue Hypothese.

calc

Zusammenfassung

Wie das Wort schon andeutet, geht es bei `calc` um konkrete Berechnungen. Dies ist kein Taktik, sondern ein `lean`-Modus. Das bedeutet, dass man diesen Modus betreten kann (mit dem Wort `calc`) Rechenschritte eingibt und Beweise dafür, dass jeder einzelne Rechenschritt stimmt.

Beispiele

Hier ein Beweis der ersten binomischen Formel, der nur durch Umschreiben von Recheneigenschaften aus der `mathlib` zustande kommt.

```
example (n : ℕ): (n+1)^2 = n^2 + 2*n + 1 :=
begin
  have h : n + n = 2*n,
  {
    nth_rewrite 0 ← one_mul n,
    nth_rewrite 1 ← one_mul n,
    rw ← add_mul,
  },
   calc (n+1)^2 = (n+1) * (n+1) : by { rw pow_two, }
  ... = (n+1)*n + (n+1) * 1: by {rw mul_add, }
  ... = n*n + 1*n + (n+1) : by {rw add_mul, rw mul_one (n+1),}
  ... = n^2 + n + (n+1) : by {rw one_mul, rw ← pow_two,}
  ... = n^2 + (n + n+1) : by {rw add_assoc, rw ← add_assoc n n 1,}
  ... = n^2 + 2*n + 1 : by { rw ← add_assoc, rw ← h, },
end
```

Dasselbe kann man auch ohne den `calc`-Modus erreichen, etwa so:

```
example (n : ℕ): (n+1)^2 = n^2 + 2*n + 1 :=
begin
  have h : n + n = 2*n, by { nth_rewrite 0 ← one_mul n,
       nth_rewrite 1 ← one_mul n, rw ← add_mul, },
  rw [pow_two, mul_add, add_mul, mul_one (n+1), one_mul,
    ← pow_two,  add_assoc, ← add_assoc n n 1,
    ← add_assoc, ← h],
end
```

Dies ist jedoch deutlich schlechter lesbar.

Anmerkungen

1. Wichtig ist die genaue Notation im `calc`-Modus, insbesondere die drei `...` und das `: by {}`.

2. Um einen Beweis im `calc`-Modus zu erzeugen, geht man am besten wiefolgt vor (hier am Beispiel eines Beweises von `h : n + n = 2*n` : Zunächst gibt man die einzelnen Rechenschritte an, und entschuldigt sich bei jedem Beweis:

   ```
   example (n : ℕ) : n + n = 2*n :=
   begin
     calc n + n = 1 * n + 1 * n : by {sorry,}
     ... = (1 + 1) * n : by {sorry, }
     ... = 2 * n : by {sorry, }
   end
   ```

   Anschließend kann man in jeder Klammer das `sorry,` mit dem richtigen Beweis schrittweise austauschen.

3. Der `calc`-Modus funktioniert nicht nur für Gleichungsketten, sondern auch für Ungleichungsketten, Teilmengenrelationen etc.

4. Deutlich einfacher löst man obiges Beispiel mit `linarith,` oder `ring,`.

`cases`

Zusammenfassung

Ist eine Hypothese zusammengesetzt, d.h. lässt sich in zwei oder mehr Fälle erweitern, so liefert `cases` genau das. Dies kann nicht nur bei Hypothesen $h : P \lor Q$ oder $h : P \land Q$ angewendet werden, sondern auch bei Strukturen, die aus mehreren Fällen bestehen, wie $\exists$... (hier gibt es eine Variable und eine Aussage) und x : bool oder $n : \mathbb{N}$.

Beispiele

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** 🎉
P: ℕ → Prop h: ∃ (m : ℕ), P m ⊢ Q	`cases x with m h1,`	P : ℕ → Prop m : ℕ h1 : P m ⊢ Q
x: bool ⊢ x = tt ∨ x = ff	`cases x,`	⊢ ff = tt ∨ ff = ff ⊢ tt = tt ∨ tt = ff
n: ℕ ⊢ n > 0 → (∃ (k : ℕ), n = k + 1)	`cases n,`	⊢ 0 > 0 → (∃ (k : ℕ), 0 = k + 1) ⊢ n.succ > 0 → (∃ (k : ℕ), n.succ = k + 1)

Anmerkungen

1. Die Anwendung `cases n` für $n : \mathbb{N}$ ist strikt schwächer als die vollständige Induktion (siehe `induction`). Durch `cases` wird ja nur $n : \mathbb{N}$ in die beiden Fälle 0 und succ n umgewandelt, aber man darf nicht die Aussage für n-1 verwenden, um die Aussage für n zu beweisen.

2. Ein besonderer Fall ist es, wenn h unmöglich ist und damit gar keine

Konstrukturen hat. Ist etwa `h : false`, so schließt `cases h` jedes Goal.

3. Eine etwas flexiblere Version von `cases` ist `rcases`.

`change`

Zusammenfassung

Ändert das Goal (bzw. eine Hypothese) in ein Goal (bzw. eine Hypothese), das (die) definitorisch gleich ist.

Beispiele

Proof state	Kommando	Neuer proof state
⊢ : P → false	`change ¬P`	⊢ ¬P
h : ¬P ⊢ Q	`change P → false at h,`	h: P → false ⊢ Q
xs: x ∈ s ⊢ x ∈ f ⁻¹' (f '' s)	`change f x ∈ f '' s,`	xs: x ∈ s ⊢ f x ∈ f '' s

Anmerkungen

1. Wie man am vorletzten Beispiel sieht, funktioniert `change` auch bei Hypothesen.
2. Da viele Taktiken sowieso auf definitorische Gleichheit testen, ist `change` oftmals nicht nötig. Es kann aber helfen, den Beweis lesbarer zu machen.

`clear`

Zusammenfassung

Mit `clear h` wird die Hypothese `h` aus dem Goal state entfernt (vergessen).

Beispiele

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

`exact`

Zusammenfassung

Ist das Goal durch einen einzigen Befehl zu lösen, dann durch die `exact` Taktik. Wie viele andere Taktiken auch funktioniert `exact` auch bei definitorischen gleichen Termen.

Beispiele

Proof state	Kommando	Neuer proof state
h : P ⊢ P	`exact h`	**goals accomplished** 🎉
hP: P hQ: Q ⊢ P ∧ Q	`exact ⟨ hP, hQ ⟩,`	**goals accomplished** 🎉
hP: P hnP: P → false ⊢ false	`exact hnP hP,`	**goals accomplished** 🎉

Anmerkungen

Beim dritten Beispiel sollte man sich die Reihenfolge einprägen, in der die beiden Hapothesen `hP` und `hnP` angewendet werden. Die erste Hypothese nach `exact` ist immer die, deren rechte Seite mit dem Goal übereinstimmt. Benötigt diese weiteren Input, wird er danach fortgeschrieben.

`exfalso`

Zusammenfassung

Die Aussage `false → P` is für alle `P` wahr. Ist das momentane Goal also `⊢ P`, und man würde diese wahre Aussage mittels `apply` anwenden, wäre das neue Goal `⊢ false`. Genau dies macht die `exfalso`-Taktik.

Beispiele

Proof state	**Kommando**	**Neuer proof state**
h : P ⊢ Q	`exfalso`	h : P ⊢ false
hP: P hnP: ¬P ⊢ Q	`exfalso,`	hP: P hnP: ¬P ⊢ false

Anmerkungen

Falls man diese Taktik anwendet, verlässt man den Bereich der konstruktiven Mathematik. (Diese verzichtet auf die Regel des ausgeschlossenen Dritten.)

`have`

Zusammenfassung

Mittels `have` führt man eine neue Behauptung ein, die man zunächst beweisen muss. Anschließend steht sie als Hypothese in allen weiteren Goals zur Verfügung. Dies ist identisch damit, zunächst ein Lemma `h` mit der Aussage nach `have h :` zu beweisen, und es dann an gegebener Stelle im Beweis wieder zu verwenden (etwa mit `apply` oder `rw`.

Beispiele

Proof state	Kommando	Neuer proof state
⊢ R	`have h : P ↔ Q,`	⊢ P ↔ Q h : P ↔ Q ⊢ R

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

Anmerkungen

1. Hat man zwei Goals (nennen wir sie `⊢1` und `⊢2`), und benötigt man im Beweis von `⊢2` die Aussage von `⊢1`, so kann man zunächst ein drittes Goal mit `have h := ⊢1` einführen (wobei `⊢1` durch die Aussage zu ersetzen ist). Anschließend kann man `⊢1` mit `exact` beweisen, und hat im Beweis von `⊢2` die Aussage `⊢1` zur Verfügung.

`intro`

Zusammenfassung

Ist das Goal von der Form `⊢ P → Q` oder `∀ (n : ℕ), P n`, so kann man mit `intro P` bzw. `intro n` weiterkommen.

Beispiele

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`

Anmerkungen

1. Mehrere `intro`-Befehle hintereinander fasst man am besten mit `intros` zusammen. Weiter ist `rintro` eine flexiblere Variante.
2. Eine Umkehrung von `intro` ist `revert`.

xxx revert

`intros`

Zusammenfassung

Dies ist genau wie bei `intro`, aber es können gleichzeitig mehrere `intro`-Befehle zu einem einzigen zusammengefasst werden. Etwas genauer ist `intros h1 h2 h3`, identisch mit `intro h1, intro h2, intro h3`.

Beispiele

Proof state	Kommando	Neuer proof state
⊢ P → Q → R	`intros hP hQ`	hP : P hQ : Q ⊢ R
P: ℕ → **Prop** ⊢ ∀ (n : ℕ), P n → Q	`intros n hP`	P: ℕ → **Prop** n: ℕ hP: P n ⊢ Q

Anmerkungen

`rintro` ist eine flexiblere Variante, bei der gleichzeitig `cases`-Anwendungen ausgeführt werden können.

`left`

Zusammenfassung

Die Anwendung von `left,` ist identisch mit `apply h,` für `h : P → P ∨ Q`. Hat man also eine Goal der Form `⊢ P ∨ Q`, so bewirkt `left,`, dass man nur noch das Goal `⊢ P` hat. Schließlich genügt es ja, `P` zu zeigen, um das Goal zu schließen.

Beispiele

Proof state	Kommando	Neuer proof state
`⊢ P ∨ Q`	`left`	`⊢ P`
`⊢ ℕ`	`left,`	**goals accomplished** 🎉

Das zweite Beispiel bedarf einer kleinen Erklärung. Zunächst muss man verstehen, dass beim Goal `⊢ ℕ` zu zeigen ist, dass es einen Term vom Typ `ℕ` gibt, also dass es eine natürlich Zahl gibt. Nun muss man wissen, wie `ℕ` in Lean implementiert ist. Dies ist

```
inductive nat
| zero : nat
| succ (n : nat) : nat
```

zusammen mit

```
notation `ℕ` := nat
```

Das bedeutet: Der Typ `ℕ` ist definiert dadurch, dass `zero` ein Term von diesem Typ ist, und dass es eine Funktion `succ : ℕ → ℕ` gibt. Mit der Eingabe von `left,` im zweiten Beispiel wird das Goal also deshalb geschlossen, weil per Definition `zero : ℕ` gilt, insbesondere gibt es also einen Term vom Typ `ℕ`.

Anmerkungen

1. Siehe auch `right,` für die entsprechende Taktik, die äquivalent ist zu `apply h` für `h : Q → P ∨ Q`.

2. Wie im zweiten Beispiel lässt sich `left,` immer dann anwenden, wenn man einen induktiven Typ mit zwei Konstruktoren (so wie `ℕ`) vor sich hat.

library_search

Zusammenfassung

Es gibt ja sehr viele bereits bewiesene Aussage in mathlib . Bei der Verwendung von library_search wird die mathlib auf Aussagen hin durchsucht, deren Typen denen der zu beweisenden Aussage entsprechen. Führt dies nicht zum Erfolg, meldet Lean einen timeout . Im Fall eines Erfolges wird außerdem berichtet, welches Kommando gefunden wurde. Clickt man darauf, so wird dies an Stelle von library_search eingesetzt.

Beispiele

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

Anmerkungen

Die Taktik suggest ist ähnlich und funktioniert auch dann, wenn das Goal nicht geschlossen werden kann.

linarith

Zusammenfassung

Beispiele

Proof state	Kommando	Neuer proof state
h1 : a < b h2 : b ≤ c ⊢ a < c [6]	linarith	**goals accomplished** 🎉

Anmerkungen

norm_num

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

Zusammenfassung

Beispiele

Anmerkungen

nth_rewrite

Proof state	Kommando	Neuer proof state
h : ⊢ 2 + 2 = 4 [8]	norm_num	**goals accomplished** 🎉

Zusammenfassung

Diese Taktik ist verwandt zu `rw`. Der Unterschied ist, dass man angeben kann, auf das wievielte Vorkommen des zu ersetzenden Terms das `rw` angewendet werden soll. Die genaue Syntax ist `nth_rewrite k h`, wobei `k` die Nummer (beginnend mit 0) des zu ersetzenden Terms ist und `h` die zu ersetzende Hypothese. Wie bei `rw` muss diese von der Form `h : x=y` oder `h : A↔B` sein.

Beispiele

Proof state	Kommando	Neuer proof state
n : ℕ ⊢ 0 + n = 0 + 0 + n	nth_rewrite 0 zero_add,	n: ℕ ⊢ n = 0 + 0 + n
n : ℕ ⊢ 0 + n = 0 + 0 + n	nth_rewrite 1 zero_add,	n: ℕ ⊢ 0 + n = n
n : ℕ ⊢ 0 + n = 0 + 0 + n	nth_rewrite 2 zero_add,	n: ℕ ⊢ 0 + n = 0 + n

Lean sieht in obigem Beispiel dreimal ein Term der Form `0 + _`: Nummer 0 ist auf der linken Seite, für Nummer 1 und 2 wird auf der rechten Seite (wegen der Klammerung `0 + 0 + n = (0 + 0) + n`) zunächst das zweite `=` gecheckt. Links davon steht `0 + 0`, was definitorisch identisch ist zu `0`. Wendet man das `rw zero_add` also hier an, wird der Term zu `n` umgewandelt. Für Nummer 2 sieht man das `0 + 0` an, stellt fest, dass es von der gewünschten Form ist und wandelt es in `0` um.

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

`rcases` ist eine flexiblere Version von `cases`. Etwas genauer ist es hier erlaubt, mittels `⟨ hP, hQ ⟩` (bzw. `(hP | hQ)`) die durch ∧ (bzw. ∨) verknüpfte Hypothesen `hP` und `hQ` in ihre Fälle aufzuteilen. Wie man im obigen Beispiel sieht, ist dabei auch eine Schachtelung von `⟨.,.⟩` und `(.|.)` möglich.

Beispiele

Proof state	Kommando	Neuer proof state
h : P ∧ Q ⊢ R	`rcases h with ⟨ hP, hQ ⟩`	hP : P hQ : Q ⊢ R
h : P ∨ Q ⊢ R	`rcases h with ( hP | hQ )`	hP : P ⊢ R hQ : Q ⊢ R
h: ∃ (m : ℕ) (hg : 0 ≤ m), m < n ⊢ P	`rcases h with ⟨ m, h1, h2 ⟩,`	n m: ℕ h1: 0 ≤ m h2: m < n ⊢ 1 < n

Anmerkungen

Im letzten Beispiel sieht man, wie man mit `rcases` einen ∃-Quantor in einer Hypothese, der mehr als eine Einschränkung hat (hier: `0 ≤ m)` und `m < n` direkt auflösen kann.

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`

Zusammenfassung

Siehe `left`, wobei die Anpassungen offensichtlich sind.

Beispiele

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

ring

Proof state	Kommando	Neuer proof state
x y : ℝ ⊢ x + y = y + x 9	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`

Zusammenfassung

`rw` steht für *rewrite*. Für `rw h` muss `h` eine Aussage vom Typ `h : x=y` oder `h : A↔B` sein. In diesem Fall wird durch `rw h` jeder Term, der syntaktisch identisch zu `x` (bzw. `A`) ist durch `y` (bzw. `B`) ersetzt. Dies funktioniert auch, wenn `h` ein bereits bewiesenes Ergebnis (also ein **lemma** oder **theorem**) ist. Mit `rw ← h` wird `rw` von rechts nach links angewendet. (In obigem Beispiel wird also `y` durch `x` bzw. `B` durch `A` ersetzt.)

Beispiele

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
k m: ℕ ⊢ k + m + 0 = m + k + 0	`rw add_comm,`	k m: ℕ ⊢ 0 + (k + m) = m + k + 0
k m: ℕ ⊢ k + m + 0 = m + k + 0	`rw add_comm k m,`	⊢ m + k + 0 = m + k + 0
k m: ℕ ⊢ k + m + 0 = m + k + 0	`rw ← add_comm k m,`	⊢ k + m + 0 = k + m + 0
k m: ℕ ⊢ k + m + 0 = m + k + 0	`rw [add_zero,` `add_zero,]`	k m: ℕ ⊢ k + m = m + k

Für die letzten vier Beispiele muss man erstmal wissen, dass `add_comm` und `add_zero` die Aussagen

```
add_comm : ∀ {G : Type} [_inst_1 : add_comm_semigroup G] (a b : G),
                                        a + b = b + a
add_zero : ∀ {M : Type} [_inst_1 : add_zero_class M] (a : M), a + 0 = a
```

sind. Im ersten der vier Beispiele wendet `rw` auf das erste Vorkommen eines Terms vom Typ `a + b` an. Durch die interne Klammerung steht auf der linken Seite `(k + m) + 0`, so dass das `rw` zu einem `0 + k + m` führt. Will man stattdessen die Kommutativität im Term `k + m` ausnutzen, so benötigt man das zweite (bzw. dritte) Beispiel, bei dem `rw add_comm k m` zum gewünschten Fortschritt führt. Im letzten Beispiel werden zunächst die beiden `+ 0`-Terme durch `rw add_zero` beseitigt.

Anmerkungen

1. `rw` wird in der Praxis sehr oft verwendet, um Aussagen der `mathlib` anzuwenden (zumindest wenn Sie vom Typ `=` oder `↔` sind).
2. Will man mehrere `rw`-Kommandos kombinieren, so kann man das in eckigen Klammern machen, etwa `rw [h1, h2]` oder `rw [h1, ←h2, h3]`.
3. `rw` führt nach seiner Anwendung sofort ein `refl` durch. Das führt im zweiten und dritten Beispiel der Anwendungen von `add_comm` und `add_zero` dazu, dass der neue Proof state nicht wie angegeben ist, sondern **goals accomplished** 🎉
4. Will man nicht der Reihe nach ein `rw` durchführen (wie etwa bei der doppelten Beseitigung des `+0` oben), so kann man mittels `nth_rewrite` gezielt das zweite Vorkommen eines Terms umschreiben.
5. Die `rw`-Taktik funktioniert nicht, wenn sie nach einem *Binder* steht, was etwa ein `∀ ∃ ∑` sein kann. In diesem Fall hilft hoffentlich `simp_rw` weiter.

simp

Proof state	Kommando	Neuer proof state
⊢ n + 0 = n [10]	`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

Bei einer Hypothese `h : ∀ n, ...` gilt `...` für alle `n`, aber für den Beweis des Goals benötigt man eventuell ja nur ein bestimmtes `n`. Gibt man `specialize h` gefolgt von dem Wert an, für den `h` benötit wird, ändert sich die Hypothese entsprechend.

Beispiele

Proof state	Kommando	Neuer proof state
h: ∀ (n : ℕ), 0 < n + 1 ⊢ 0 < 1	`specialize h 0,`	m : ℕ h: 0 < 0 + 1 ⊢ 0 < 1

Anmerkungen

1. Genau wie bei `use` muss man aufpassen, dass das Goal beweisbar bleibt.
2. Will man zwei Werte der Hypothese `h` verwendet, so liefert `let h' := h` zunächst eine Verdopplung der Hypothese, so dass man anschließend `specialize` auf `h` und `h'` anwenden kann.

`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 [11]	tauto oder tauto!	**goals accomplished** 🎉

Zusammenfassung

tauto löst alle Goals, die mit einer Wahrheitstabelle lösbar sind. Es gibt auch noch tauto!, was zusätzlich noch die Regel des ausgeschlossenen Dritten verwendet.

Beispiele

Anmerkungen

triv

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

Zusammenfassung

triv löst ein Ziel, das definitorisch identisch zu true ist. Es löst ebenfalls Ziele, die mit refl lösbar sind.

Beispiele

Proof state	Kommando	Neuer proof state
⊢ x=x	triv,	**goals accomplished** 🎉

`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

Die `use`-Taktik kommt bei Goals zum Einsatz, die mit ∃ beginnen. Hier wird durch Paramtere gesagt, welches durch ∃ quantifizierte Objekt denn im Beweis weiter verwendet werden soll.

Beispiele

Proof state	Kommando	Neuer proof state
`⊢ ∃ (k : ℕ), k * k = 16`	`use 4,`	⊢ 4 * 4 = 16
`⊢ ∃ (k l : ℕ), k * l = 16`	`use [8, 2],`	⊢ 8 * 2 = 16

Anmerkungen

1. Man muss aufpassen, dass das Goal durch die Verwendung von `use` beweisbar (insbesondere wahr) bleibt. Im ersten Fall unter Beispiele hätten wir ja auch `use 3` schreiben können, und `3 * 3 = 16` ist nicht beweisbar.
2. Man kann gleichzeitig mehr als eine Variable durch `use` angeben. Dies geschieht in eckigen Klammern; siehe das letzte Beispiel.