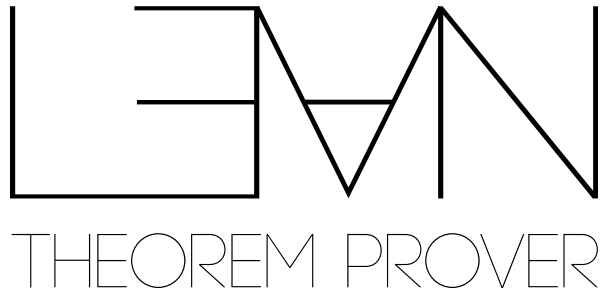


SCHULMATHEMATIK MIT DEM



Peter Pfaffelhuber
Sommersemester 2023

universität freiburg



2. April 2023

Schulmathematik mit dem Lean Theorem Prover

Inhaltsverzeichnis

1	Einleitung	3
2	Mathematik	4
2.1	Logik	4
2.2	Mengen	5
3	Hinweise zu Lean	6
4	Hinweise zu vscode	7
5	Taktiken	7
5.1	Cheatsheet	7
6	Taktiken	10
6.1	<code>apply</code>	10
6.2	<code>assumption</code>	10
6.3	<code>bycases</code>	11
6.4	<code>bycontra</code>	11
6.5	<code>cases</code>	11
6.6	<code>clear</code>	12
6.7	<code>exact</code>	13
6.8	<code>exfalse</code>	13
6.9	<code>have</code>	13
6.10	<code>intro</code>	14
6.11	<code>intros</code>	14
6.12	<code>left</code>	14
6.13	<code>library_search</code>	15
6.14	<code>linarith</code>	15
6.15	<code>norm_num</code>	15
6.16	<code>obtain</code>	16
6.17	<code>rcases</code>	16
6.18	<code>refine</code>	16
6.19	<code>refl</code>	17
6.20	<code>right</code>	17
6.21	<code>ring</code>	17
6.22	<code>rintro</code>	18

6.23	<code>rw</code>	19
6.24	<code>simp</code>	19
6.25	<code>specialize</code>	19
6.26	<code>split</code>	20
6.27	<code>tauto</code>	20
6.28	<code>triv</code>	20
6.29	<code>use</code>	21
7	Beweisbeispiele aus der Schulmathematik	21

1 Einleitung

Dies sind die Notizen zu einem Kurs zum formalen Beweisen mit einem interaktiven Theorem-Prover im Sommersemester 2023 an der Universität Freiburg. 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 Beweises, die von der Zukunftsfähigkeit von Theorem Provers überzeugt ist.

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 Lean 3 zu Lean 4 an. Da insbesondere noch nicht die gesamte `mathlib` in Lean 4 zur Verfügung steht, werden wir Lean 3 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, von der Installation, über die Syntax, bis hin zu verwendeten Gleichheitsbegriffen. Im Kapitel 5 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 \Rightarrow verwendet wird, sondern ein einfacher \rightarrow . Wir gehen hier folgende logische Schlüsse durch:

- Blatt 01-a:

Die Aussage $\vdash P \rightarrow Q$ (d.h. aus P folgt Q) bedeutet ja, dass Q gilt, falls man annehmen darf, dass die Hypothese P richtig ist. Dieser Übergang von $\vdash P \rightarrow Q$ zur Hypothese $hP : P$ mit Ziel $\vdash Q$ erfolgt mittels `intro hP`. Mehrere `intro`-Befehle kann man mittels `intros h1 h2...` abkürzen.

Gilt die Hypothese $hP : P$, und wir wollen $\vdash P$ 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 $\vdash Q$ beweisen, und weiß, dass $hPQ : P \rightarrow Q$ gilt, so genügt es, $\vdash P$ zu beweisen (da mit hPQ daraus dann $\vdash Q$ folgt). Mit `apply hPQ` wird in diesem Fall das Ziel nach $\vdash P$ geändert.

Hinter einer Äquivalenz-Aussage $\vdash P \leftrightarrow Q$ stehen eigentlich die beiden Aussagen $\vdash P \rightarrow Q$ und $\vdash Q \rightarrow P$. Mittels `split` wandelt man das Ziel $\vdash P \leftrightarrow Q$ in zwei Ziele für die beiden Richtungen um.

Die logische Verneinung wird in Lean3 mit \neg notiert. Die Aussage $\neg P$ ist dabei definitorisch gleich $P \rightarrow \text{false}$, wobei `false` für eine falsche Aussage steht.

- Blatt 01-c:

`exfalso`

Bei einem Beweis durch Widerspruch beweist man statt $\vdash P$ die Aussage $\vdash \neg P \rightarrow \text{false}$. (Dies ist logisch korrekt, da P genau dann wahr ist, wenn $\neg P$ 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 Lean3 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 offenbar 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 $hQ : Q$ ersetzt 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:

$P : \text{Prop}$ 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

$$f(S) := f '' S := \{y : Y \mid \exists x : X, x \in S \wedge f x = y\},$$

$$f^{-1}(T) := f^{-1} ' T := \{x : X \mid f x \in T\}.$$

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 $\alpha \rightarrow \alpha$ hat. Anders gesagt muss x^{-1} denselben Typ haben wie x . Die Notation $f^{-1} ' T$ ist für `set.preimage`.

3 Hinweise zu Lean

Installation

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

Eine Hypothese in Lean3 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 $P \rightarrow Q$, also die Aussage, dass aus P die Aussage Q folgt.

Terms of Types

sorry-Taktik

Wir beginnen mit der Beschreibung zweier grundlegender Taktiken, nämlich `intro` und `exact`; bitte in Kapitel 5 nachlesen. Das denkbar einfachste Beispiel ist die Aussage $P \rightarrow P$, d.h. aus P folgt P . In Lean3 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 direkt nach `begin`, so ist der *proof-state*

```
P : Prop
⊢ P → P
```

Hier ist wichtig zu wissen, dass hinter \vdash 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 $P \rightarrow P$ 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 sucht 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 Lean3 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 $P \rightarrow Q \rightarrow R$ zu lesen als $P \rightarrow (Q \rightarrow R)$.

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.)

5 Taktiken

5.1 Cheatsheet



Binäre Operatoren wie *und* (\wedge), *oder* (\vee), *Schnittmengen* (\cap) und *Vereinigungsmengen* (\cup) sind rechts-assoziativ, also z.B. $P \wedge Q \wedge R := P \wedge (Q \wedge R)$.

Proof state	Kommando	Neuer proof state
$\vdash P \rightarrow Q$	<code>intro hP</code>	$hP : P$ $\vdash Q$
$f : \alpha \rightarrow \text{Prop}$ $\vdash \forall \{x : \alpha\}, f\ x$	<code>intro x</code>	$f : \alpha \rightarrow \text{Prop}$ $x : \alpha$ $\vdash f\ x$
$\vdash \text{true}$	<code>triv</code>	goals accomplished 🏁

Proof state	Kommando	Neuer proof state
$\vdash P \rightarrow Q$	<code>intro hP</code>	$hP : P$ $\vdash Q$
$f : \alpha \rightarrow \text{Prop}$ $\vdash \forall \{x : \alpha\}, f\ x$	<code>intro x</code>	$f : \alpha \rightarrow \text{Prop}$ $x : \alpha$ $\vdash f\ x$
$h : P$ $\vdash P$	<code>exact h</code>	goals accomplished 🏁
$h : P$ $\vdash P$	<code>assumption</code>	goals accomplished 🏁
$h : P \rightarrow Q$ $\vdash Q$	<code>apply h</code>	$h : P \rightarrow Q$ $\vdash P$
$\vdash P \rightarrow Q \rightarrow R$	<code>intros hP hQ</code>	$hP : P$ $hQ : Q$ $\vdash R$
$\vdash P \wedge Q \rightarrow P^1$	<code>tauto</code> oder <code>tauto!</code>	goals accomplished 🏁

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

Proof state	Kommando	Neuer proof state
$\vdash \text{true}$	<code>triv</code>	goals accomplished 🏆
$h : P$ $\vdash Q$	<code>exfalse</code>	$h : P$ $\vdash \text{false}$
$\vdash P$	<code>by_contra h</code>	$h : \neg P$ $\vdash \text{false}$
$\vdash P$	<code>by_cases h : Q</code>	$h : Q$ $\vdash P$ $h : \neg Q$ $\vdash P$
$h : P \wedge Q$ $\vdash R$	<code>cases h with hP hQ</code>	$hP : P$ $hQ : Q$ $\vdash R$
$h : P \vee Q$ $\vdash R$	<code>cases h with hP hQ</code>	$hP : P$ $\vdash R$ $hQ : Q$ $\vdash R$
$h : \text{false}$ $\vdash P$	<code>cases h</code>	goals accomplished 🏆
$\vdash P \wedge Q$	<code>split</code>	$\vdash P$ $\vdash Q$
$\vdash P \leftrightarrow Q$	<code>split</code>	$\vdash P \rightarrow Q$ $\vdash Q \rightarrow P$
$\vdash P \leftrightarrow P$ oder $\vdash P = P$	<code>refl</code>	goals accomplished 🏆
$h : P \leftrightarrow Q$ $\vdash P$	<code>rw h</code>	$h : P \leftrightarrow Q$ $\vdash Q$
$h : P \leftrightarrow Q$ $\vdash Q$	<code>rw < h</code>	$h : P \leftrightarrow Q$ $\vdash P$
$h : P \leftrightarrow Q$ $hP : P$	<code>rw h at hP</code>	$h : P \leftrightarrow Q$ $hP : Q$
$h : P \leftrightarrow Q$ $hQ : Q$	<code>rw < h at hQ</code>	$h : P \leftrightarrow Q$ $hQ : P$
$\vdash P \vee Q$	<code>left</code>	$\vdash P$
$\vdash P \vee Q$	<code>right</code>	$\vdash Q$

Proof state	Kommando	Neuer proof state
$h : \vdash 2 + 2 = 4^2$	<code>norm_num</code>	goals accomplished 
$f : \alpha \rightarrow \text{Prop}$ $y : \alpha$ $\vdash \exists (x : \alpha), f\ x$	<code>use y</code>	$f : \alpha \rightarrow \text{Prop}$ $y : \alpha$ $\vdash f\ y$
$x\ y : \mathbb{R}$ $\vdash x + y = y + x^3$	<code>ring</code>	goals accomplished 
$\vdash P \rightarrow Q$	<code>intro hP</code>	$hP : P$ $\vdash Q$
$f : \alpha \rightarrow \text{Prop}$ $\vdash \forall \{x : \alpha\}, f\ x$	<code>intro x</code>	$f : \alpha \rightarrow \text{Prop}$ $x : \alpha$ $\vdash f\ x$
$h1 : a < b$ $h2 : b \leq c$ $\vdash a < c^4$	<code>linarith</code>	goals accomplished 
$h : P$ $\vdash Q$	<code>clear h</code>	$\vdash Q$
$f : \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}), f\ n$ $\vdash P$	<code>specialize h 13</code>	$f : \mathbb{N} \rightarrow \text{Prop}$ $h : f\ 13$ $\vdash P$
$f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}),$ $\quad \exists (m : \mathbb{N}), f\ n\ m$	<code>obtain < m, hm ></code> <code>:= h 27</code> <code>=</code>	$f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}),$ $\quad \exists (m : \mathbb{N}), f\ n\ m$
<code>have h1 : $\exists m,$</code>	<code>m : \mathbb{N}</code> <code>f 27 m, ...</code> <code>cases h1 with m hm</code>	$hm : f\ 27\ m$ $\vdash P$
$h1 : a < b$ $h2 : b < c$ $\vdash a < c$	<code>library_search</code>	goals accomplished  Try this: <code>exact lt_trans h1 h2</code>
$hQ : Q$ $\vdash P \wedge Q$	<code>refine < _, hQ ></code>	$hQ : Q$ $\vdash P$
$\vdash P \vee Q \rightarrow R$	<code>rintro (hP hQ)</code> <code>=</code> <code>intro h,</code>	$hP : P$ $\vdash P$ $hQ : Q$

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.

Proof state	Kommando	Neuer proof state
	<code>cases h with hP hQ</code>	$\vdash Q$
$\vdash P \wedge Q \rightarrow R$	<code>rintro < hP , hQ ></code> <code>=</code> <code>intro h,</code> <code>cases h with h1 h2</code>	$hP : P$ $hQ : Q$ $\vdash Q$
$h : P \wedge Q \vee P \wedge R$ $\vdash P$	<code>rcases h with</code> <code>(<hP1,hQ> <hP2,hR>)</code>	$hP1 : P$ $hQ : Q$ $\vdash P$ $hP2 : P$ $hR : R$ $\vdash P$
$\vdash n + 0 = n$ ⁵	<code>simp</code>	goals accomplished 🏁
$h : n + 0 = m$ ⁵ $\vdash P$	<code>simp at h</code>	$h : n = m$ $\vdash P$

change let

```
def f : ℕ → ℝ := λ n, n^2 + 3
```

6 Taktiken

6.1 apply

Proof state	Kommando	Neuer proof state
$h : P \rightarrow Q$ $\vdash Q$	<code>apply h</code>	$h : P \rightarrow Q$ $\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.2 assumption

Proof state	Kommando	Neuer proof state
$h : P$ $\vdash P$	<code>assumption</code>	goals accomplished 🏁

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

Zusammenfassung

Beispiele

Anmerkungen

6.3 bycases

Proof state	Kommando	Neuer proof state
$\vdash P$	<code>by_cases h : Q</code>	$h : Q$ $\vdash P$ $h : \neg Q$ $\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.4 bycontra

Proof state	Kommando	Neuer proof state
$\vdash P$	<code>by_contra h</code>	$h : \neg P$ $\vdash \text{false}$

Zusammenfassung

Beispiele

Anmerkungen

6.5 cases

Proof state	Kommando	Neuer proof state
$h : P \wedge Q$ $\vdash R$	<code>cases h with hP hQ</code>	$hP : P$ $hQ : Q$ $\vdash R$
$h : P \vee Q$ $\vdash R$	<code>cases h with hP hQ</code>	$hP : P$ $\vdash R$ $hQ : Q$ $\vdash R$
$h : \text{false}$ $\vdash P$	<code>cases h</code>	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:  $\alpha \rightarrow \beta$ 
s: set  $\alpha$ 
x:  $\alpha$ 
xs:  $x \in s$ 
 $\vdash x \in f^{-1}(f \text{ '' } s)$ 
```

so ändert `change f x \in f '' s`, das Goal zu $\vdash f x \in f \text{ '' } s$.

`change` funktioniert auch bei Hypothesen. Ist eine Hypothese $h : x \in f^{-1}(f \text{ '' } s)$, dann ist nach `change f x \in f '' s at h` die Hypothese $h : f x \in f \text{ '' } s$.

Anmerkungen

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

6.6 clear

Proof state	Kommando	Neuer proof state
$h : P$ $\vdash Q$	<code>clear h</code>	$\vdash Q$

Zusammenfassung

Beispiele

Anmerkungen

6.7 `exact`

Proof state	Kommando	Neuer proof state
<code>h : P</code> <code>└ P</code>	<code>exact h</code>	goals accomplished 🏆

Zusammenfassung

Beispiele

Anmerkungen

6.8 `exfalse`

Proof state	Kommando	Neuer proof state
<code>h : P</code> <code>└ Q</code>	<code>exfalse</code>	<code>h : P</code> <code>└ false</code>

Zusammenfassung

Beispiele

Anmerkungen

6.9 `have`

Proof state	Kommando	Neuer proof state
<code>have h1 : ∃ m,</code>	<code>m : N</code> <code>f 27 m, ...</code> <code>cases h1 with m hm</code>	<code>hm: f 27 m</code> <code>└ P</code>

Zusammenfassung

Beispiele

Anmerkungen

6.10 `intro`

Proof state	Kommando	Neuer proof state
$\vdash P \rightarrow Q$	<code>intro hP</code>	$hP : P$ $\vdash Q$
$f : a \rightarrow \text{Prop}$ $\vdash \forall \{x : a\}, f\ x$	<code>intro x</code>	$f : a \rightarrow \text{Prop}$ $x : a$ $\vdash f\ x$

Zusammenfassung

Beispiele

Anmerkungen

6.11 `intros`

Proof state	Kommando	Neuer proof state
$\vdash P \rightarrow Q \rightarrow R$	<code>intros hP hQ</code>	$hP : P$ $hQ : Q$ $\vdash R$

Zusammenfassung

Beispiele

Anmerkungen

6.12 `left`

Proof state	Kommando	Neuer proof state
$\vdash P \vee Q$	<code>left</code>	$\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.13 library_search

Proof state	Kommando	Neuer proof state
$\begin{array}{l} h1 : a < b \\ h2 : b < c \\ \vdash a < c \end{array}$	library_search	goals accomplished 🏆 Try this: exact lt_trans h1 h2

Zusammenfassung

Beispiele

Anmerkungen

6.14 linarith

Proof state	Kommando	Neuer proof state
$\begin{array}{l} h1 : a < b \\ h2 : b \leq c \\ \vdash a < c^6 \end{array}$	linarith	goals accomplished 🏆

Zusammenfassung

Beispiele

Anmerkungen

6.15 norm_num

Proof state	Kommando	Neuer proof state
$h : \vdash 2 + 2 = 4^7$	norm_num	goals accomplished 🏆

Zusammenfassung

Beispiele

Anmerkungen

6.16 obtain

Proof state	Kommando	Neuer proof state
$f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}),$ $\exists (m : \mathbb{N}), f\ n\ m$	<code>obtain < m, hm ></code> <code>:= h 27</code> <code>=</code>	$f : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}),$ $\exists (m : \mathbb{N}), f\ n\ m$

Zusammenfassung

Beispiele

Anmerkungen

6.17 rcases

Proof state	Kommando	Neuer proof state
$h : P \wedge Q \vee P \wedge R$ $\vdash P$	<code>rcases h with</code> <code>(<hP1, hQ> <hP2, hR>)</code>	$hP1 : P$ $hQ : Q$ $\vdash P$ $hP2 : P$ $hR : R$ $\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.18 refine

Proof state	Kommando	Neuer proof state
$hQ : Q$ $\vdash P \wedge Q$	<code>refine < _, hQ ></code>	$hQ : Q$ $\vdash P$

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 $\varepsilon^2 > 0$ aus $0 < \varepsilon$ zu beweisen.

Anmerkungen

6.19 refl

Proof state	Kommando	Neuer proof state
$\vdash P \leftrightarrow P$ oder $\vdash P = P$	<code>refl</code>	goals accomplished 🏁

Zusammenfassung

Beispiele

Anmerkungen

6.20 right

Proof state	Kommando	Neuer proof state
$\vdash P \vee Q$	<code>right</code>	$\vdash Q$

Zusammenfassung

Beispiele

Anmerkungen

6.21 ring

Proof state	Kommando	Neuer proof state
$x y : \mathbb{R}$ $\vdash x + y = y + x$	<code>ring</code>	goals accomplished 🏁

Zusammenfassung

Beispiele

Anmerkungen

6.22 `rintro`

Proof state	Kommando	Neuer proof state
$\vdash P \vee Q \rightarrow R$	<code>rintro (hP hQ)</code> = <code>intro h,</code> <code>cases h with hP hQ</code>	$hP : P$ $\vdash P$ $hQ : Q$ $\vdash Q$
$\vdash P \wedge Q \rightarrow R$	<code>rintro < hP , hQ ></code> = <code>intro h,</code> <code>cases h with h1 h2</code>	$hP : P$ $hQ : Q$ $\vdash Q$

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 \vee in einem Schritt in Fälle aufgeteilt werden: Bei $A \vee B \vee C$ werden mit `rintro (A | B | C)` drei Goals eingeführt.

Anmerkungen

6.23 `rw`

Proof state	Kommando	Neuer proof state
$h : P \leftrightarrow Q$ $\vdash P$	<code>rw h</code>	$h : P \leftrightarrow Q$ $\vdash Q$
$h : P \leftrightarrow Q$ $\vdash Q$	<code>rw ← h</code>	$h : P \leftrightarrow Q$ $\vdash P$
$h : P \leftrightarrow Q$ $hP : P$	<code>rw h at hP</code>	$h : P \leftrightarrow Q$ $hP : Q$
$h : P \leftrightarrow Q$ $hQ : Q$	<code>rw ← h at hQ</code>	$h : P \leftrightarrow Q$ $hQ : P$

Zusammenfassung

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

Beispiele

Anmerkungen

6.24 `simp`

Proof state	Kommando	Neuer proof state
$\vdash n + 0 = n$	<code>simp</code>	goals accomplished 🎉
$h : n + 0 = m$ $\vdash P$	<code>simp at h</code>	$h : n = m$ $\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.25 `specialize`

Proof state	Kommando	Neuer proof state
$f : \mathbb{N} \rightarrow \text{Prop}$ $h : \forall (n : \mathbb{N}), f n$ $\vdash P$	<code>specialize h 13</code>	$f : \mathbb{N} \rightarrow \text{Prop}$ $h : f 13$ $\vdash P$

Zusammenfassung

Beispiele

Anmerkungen

6.26 `split`

Proof state	Kommando	Neuer proof state
$\vdash P \wedge Q$	<code>split</code>	$\vdash P$ $\vdash Q$
$\vdash P \leftrightarrow Q$	<code>split</code>	$\vdash P \rightarrow Q$ $\vdash Q \rightarrow P$

Zusammenfassung

Beispiele

Anmerkungen

6.27 `tauto`

Proof state	Kommando	Neuer proof state
$\vdash P \wedge Q \rightarrow P^{10}$	<code>tauto</code> oder <code>tauto!</code>	goals accomplished 🎉

Zusammenfassung

Beispiele

Anmerkungen

6.28 `triv`

Proof state	Kommando	Neuer proof state
$\vdash \text{true}$	<code>triv</code>	goals accomplished 🎉

Zusammenfassung

Beispiele

Anmerkungen

6.29 use

Proof state	Kommando	Neuer proof state
$f : \alpha \rightarrow \text{Prop}$ $y : \alpha$ $\vdash \exists (x : \alpha), f x$	use y	$f : \alpha \rightarrow \text{Prop}$ $y : \alpha$ $\vdash f y$

Zusammenfassung

Beispiele

Anmerkungen

7 Beweisbeispiele aus der Schulmathematik

- $\forall n \in \mathbb{N} \exists k \in \mathbb{N}, a_0, \dots, a_k \in \{0, \dots, 9\} : (a_k \neq 0) \wedge (n = \sum_{i=0}^k a_i 10^i)$
- Eindeutigkeit in 1.
- $\forall n = \sum_{k=0}^{\infty} a_k 10^k \in \mathbb{N} : 2|n \iff a_0 \in \{0, 2, 4, 6, 8\}$.
- $\forall n = \sum_{k=0}^{\infty} a_k 10^k \in \mathbb{N} : 3|n \iff 3|\sum_{i=0}^k a_i$.
- Sei \mathbb{P} die Menge aller Primzahlen. Dann gilt
 $\forall n \in \mathbb{N}, p \in \mathbb{P} : p|n^2 \iff p|n$.
- $\sqrt{2} \notin \mathbb{Q}$
- Für $n \in \mathbb{N}$ sei $T_n := \{k \in \mathbb{N} : k > 1 \wedge k|n\}$. Dann gilt $\min T_n \in \mathbb{P}$
- $|\mathbb{P}| = \infty$.
- Formalisierung von m/n kann gekürzt werden.
- Sei $f : x \mapsto x^n$ Dann ist $f'(x) = nx^{n-1}$.
- Zwischenwertsatz.

Aus Fundamente | der Mathematik | Baden-Württemberg Gymnasium 8, S. 33:

Beispiel 1: Die Summe aus einer geraden natürlichen Zahl und ihrem Nachfolger ist stets ungerade. Begründe, dass die Aussage wahr ist.

5. Formuliere die Voraussetzung und die Behauptung mit Variablen und beweise dann die Aussage für natürliche Zahlen. a) Die Summe von zwei aufeinanderfolgenden geraden Zahlen ist immer gerade. b) Das Produkt von zwei aufeinanderfolgenden geraden Zahlen ist immer durch 4 teilbar. c) Jede durch

15 teilbare Zahl ist auch durch 5 teilbar. d) Das Produkt des Vorgängers und Nachfolgers einer Zahl ist kleiner als das Quadrat dieser Zahl. e) Das Quadrat jeder ungeraden Zahl ist ungerade.

6 . Die folgenden Aussagen sind falsch. Beweise dies durch ein Gegenbeispiel. a) Die Summe von vier aufeinanderfolgenden natürlichen Zahlen ist durch 4 teilbar. b) Die Summe von sechs aufeinanderfolgenden natürlichen Zahlen ist durch 6 teilbar. c) Wenn eine Zahl durch 3 teilbar ist, dann ist sie auch durch 6 teilbar.

8 . Forschungsauftrag: Beweise sind in der mathematischen Forschung von großer Bedeutung. Informiere dich über die Goldbachsche Vermutung. Recherchiere auch zu den Hilbertschen Problemen.

S. 86 ff:

Beispiel 1: Beweise folgende Aussage mit einem indirekten Beweis: Wenn das Quadrat einer natürlichen Zahl ungerade ist, dann ist auch die Zahl selbst ungerade.

Klasse 9, S. 41: Beweise: Für $a > 0$ und $b > 0$ gilt immer: $\sqrt{a+b} < \sqrt{a} + \sqrt{b}$.

S. 46:

20 Beweise nachfolgende Regeln für $a > 0; m, n \neq 0; m, n$ natürliche Zahlen. $\sqrt[m]{a} \sqrt[n]{a} = \sqrt[mn]{a^{m+n}}$

12. Rechenregeln für Binomialkoeffizienten: