Natürliches Schließen in Lean

Burkhardt Renz

11.05.2025

Inhaltsverzeichnis

Natürliches Schließen

David Hilbert hat bei der Formalisierung der Aussagen- und der Prädikatenlogik eine Reihe von Axiomen bzw. Axiomenschemata verwendet und im Grunde nur eine einzige Schlussregel: den *Modus Ponens*.

Folge: Beweise im Hilbert-Kalkül zu führen ist nicht ganz einfach, jedenfalls nicht so, wie der praktizierende Mathematiker oder Logiker normalerweise argumentieren würde.

Gerhard Gentzen hat demgegenüber ein Beweissystem entwickelt, das nahezu ohne Axiome auskommt, dafür viele Schlussregeln kennt, genau gesagt: für jeden logischen Operator und Quantor jeweils eine Regel zum Einführen der Operators und eine zum Auflösen des Operators. Sein Beweissystem wird als *Natürliches Schließen* bezeichnet.

Lassen wir Gerhard Gentzen selbst sprechen:

> „Mein erster Gesichtspunkt war folgender: Die Formalisierung des logischen Schließens, wie sie insbesondere durch Frege, Russell und Hilbert entwickelt worden ist, entfernt sich ziemlich weit von der Art des Schließens, wie sie in Wirklichkeit bei mathematischen Beweisen geübt wird. […] Ich wollte nun zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahekommt. So ergab sich ein ‚Kalkül des natürlichen Schließens'."

Wir wollen uns ansehen, wie man diese Kalkül des natürlichen Schließens in *Lean* verwenden kann.

Lean

Lean ist eine funktionale Programmiersprache basierend auf *Dependent Type Theory* und zugleich ein interaktives Beweissystem basierend auf dem *Calculus of Inductive Constructions*.

Lean als funktionale Sprache

Lean ist eine effiziente funktionale Sprache, die zu C compiliert wird. Als Sprache hat Lean viel Ähnlichkeit zu Haskell. Lean als funktionale Sprache wird beschrieben in Functional Programming in Lean von David Thrane Christiansen.

Ein erstes Beispiel, damit man einen Eindruck von Lean als funktionaler Sprache bekommt:

```
import MathLib.Tactic
open Nat

def fact: ℕ → ℕ
| 0     => 1
| (n+1) => (n+1) * fact n

#eval fact 10
#eval fact 100
```

Lean als Beweissystem

Gleichzeitig ist Lean ein interaktives Beweissystem, es wird im Detail vorgestellt in Theorem Proving in Lean 4 von Jeremy Avigad, Leonardo de Moura, Soonho Kong und Sebastian Ullrich.

Mit Lean wurde zum Beispiel die Unabhängigkeit der Kontinuums-Hypothese von den Axiomen der Zermelo-Fraenkel-Mengenlehre formal bewiesen. Kurt Gödel hatte 1938 gezeigt, dass mit ZFC sich die Kontinuums-Hypothexe nicht widerlegen lässt und Paul Cohen hat 1963 beweisen, dass sie aus ZFC auch nicht herleiten werden kann.

Dies ist nur ein Beispiel einer ganzen Reihe nicht gerade trivialer mathematischer Ergebnisse, die mit Lean formalisiert wurden.

Auch hier mag ein simples Beispiel einen ersten Eindruck geben:

```
example (a b c: ℕ): a + b + c = a + c + b := by
  rw [add_assoc]
  rw [add_comm b]
  rw [<-add_assoc ]
  done
```

oder kürzer:

```
lemma add_three (a b c: ℕ): a + b + c = a + c + b := by
  rw [add_assoc, add_comm b, <-add_assoc]
```

Propositions-as-Types

Wie kann der Typchecker einer funktionalen Sprache zugleich ein Beweissystem sein?

Stellen wir uns vor, die Sprache kennt einen Typ `Prop`, der Aussagen (der Aussagen- oder Prädikatenlogik) repräsentiert und außerdem für jede Aussage `p: Prop` einen Typ `Proof p` für die Beweise von `p`.

Man könnte dann Konstanten eines solchen Typs als Axiome betrachten. Zum Beispiel so etwas wie

```
axiom modus_ponens: (p q: Prop) → Proof (Implies p q) → Proof p → Proof q
```

Diese Axiom sagt aus, dass aus einem Beweis von `p` impliziert `q` sowie einem Beweis von `p` ein Beweis von `q` folgt.

Mit einem solchen Ansatz könnte ein Typchecker prüfen, ob ein Ausdruck `t` ein korrekter Beweis der Aussage `p: Prop` ist, indem er untersucht, ob `t` den Typ `Proof p` hat.

Tatsächlich geht das auch etwas eleganter:

1. Man kann `Proof p` einfach mit `p` identifizieren. Wir interpretieren ein `p: Prop` selbst als Typ, nämlich dem Typ der Beweise von `p`. Ein Term `t: p` ist also gerade ein Beweis von `p`.
2. Das bedeutet aber, dass eine Implikation `(Implies p q)` nichts anderes ist als eine Funktion `p → q`, die Beweise für `p` in Beweise für `q` überführt.

Die Regeln für die Einführung der Implikation und die Auflösung der Implikation (Modus Ponens) entsprechen in diesem Ansatz gerade der Abstraktion und der Applikation von Funktionen.

Diese Korrespondenz wird gerne *Curry-Howard-Isomophismus* genannt oder auch *Proposition_as_Types*.

Wenn wir `t: p` haben für eine Aussage `p: Prop`, dann können wir dies lesen als

- `t` ist ein Beweis von `p`, oder
- `p` ist wahr, und dieses Fakt heißt `t`.

Also: `p: Prop` bedeutet, dass `p` wahr oder falsch sein kann und `t: p` bedeutet, dass `p` eine wahre Aussage ist.

Intern behandelt Lean zwei Terme `t1 t2: p` als per Definition äquivalent. Dies nennt man die *Irrelevanz des Beweises*.

Fazit:

1. Der Beweischecker von Lean ist einfach der Typchecker der Sprache, der prüft, ob ein Beweis `t` vom Typ `p` ist.
2. Das interaktive Beweissystem Lean hilft uns also einen Term `t` vom Typ `p` zu konstruieren, wenn wir beweisen wollen, dass `p` wahr ist.
3. Haben wir einen Beweis gefunden, dann braucht er uns nicht weiter zu interessieren, wir wechseln zur Perspektive, dass `t: p` bedeutet, dass `p` wahr ist, wie immer der Term `t` im Detail auch aussehen mag.

Regeln des natürlichen Schließens

Zunächst werden die Regeln des natürlichen Schließens kurz vorgestellt und erläutert. Die Regeln sind so zu lesen, dass ein Term unter dem Strich gültig ist, wenn die Terme über dem Strich bereits hergeleitet sind. Oder: ein Term unter dem Strich gibt einem die Beweisverpflichtung, die Terme über dem Strich herzuleiten.

Dabei ist der Gültigkeitsbereich zu berücksichtigen: Innerhalb einer Box können Terme des Kontextes verwendet werden, im Kontext außerhalb der Box aber niemals ein Term innerhalb einer Box.

Es ist recht leicht zu sehen, dass die Regeln Wahrheit erhalten, d.h. dass Herleitungen mittels des Kalküls des natürlichen Schließens *korrekt* sind. Das Kalkül ist auch *vollständig*, d.h. mit den Regeln kann jede wahre Aussage der Aussagen- und der Prädikatenlogik hergeleitet werden. (Genauer wird dies behandelt in jedem ordentlichen Lehrbuch der formalen Logik und auch in meinem Skript Logik und formale Methoden.)

Aussagenlogik

Implikation

Die Regeln für die Implikation sind:

	Einführung	*Elimination*
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow \mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\ \rightarrow \mathrm{e}, \mathrm{MP}$

Die Implikation leitet man her, indem man die Hypothese als gegeben annimmt und dann daraus die Folgerung herleitet. In der Regel wird in der Box oberhalb des Strichs angegeben, dass ϕ nur *innerhalb* der Box als gegeben angenommen werden darf. Die senkrechten Punkte $\vdots$ markieren die Beweisverpflichtung, nämlich dass sie durch einen Beweis ersetzt werden müssen, der ψ aus ϕ herleitet.

Die Implikation kann man entfernen, wenn man die Hypothese ϕ bewiesen hat und ebenso, dass $\phi \rightarrow \psi$ gilt. Dann hat man ψ bewiesen. Diese Schlussfigur ist schon seit der Antike geläufig und wird als *Modus ponens* bezeichnet, deshalb auch die Abkürzung MP.

Konjunktion

	Einführung	*Elimination*	
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\ \wedge \mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\ \wedge \mathrm{e}_1$	$\dfrac{\phi \wedge \psi}{\psi}\ \wedge \mathrm{e}_2$

Die Konjunktion kann man einführen, wenn man Herleitungen für die beiden Formeln der Konjunktion bereits hat.

Für die Elimination der Konjunktion gibt es zwei Subregeln: Eine Herleitung der Gesamtformel der Konjunktion kann man sowohl als Herleitung der linken Teilformel als auch der rechten Teilformel nehmen.

Disjunktion

	Einführung	*Elimination*
$\lor$	$\dfrac{\phi}{\phi \lor \psi}$ $\lor\mathsf{i}_1$ $\quad$ $\dfrac{\psi}{\phi \lor \psi}$ $\lor\mathsf{i}_2$	$\dfrac{\phi \lor \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}$ $\lor\mathsf{e}$

Wenn man eine Herleitung für ϕ hat, hat man auch eine Herleitung für $\phi \lor \psi$, ebenso darf man die Herleitung von ψ als Beweis für $\phi \lor \psi$ nehmen.

Will man die Disjunktion entfernen und dabei χ herleiten, muss man für jede Teilformel der Disjunktion eine Herleitung von χ finden. Diese Regel entspricht also der Beweistechnik der Fallunterscheidung.

Negation

	Einführung	*Elimination*
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}$ $\neg\mathsf{i}$	$\dfrac{\phi \quad \neg\phi}{\psi}$ $\neg$e, EFQ

Will man beweisen, dass $\neg\phi$ gilt — also die Negation einführen —, nimmt man an, dass ϕ bewiesen ist und führt diesen Beweis dann fort, bis man den Widerspruch $\bot$ hergeleitet hat. Daraus ergibt sich, dass $\neg\phi$ bewiesen ist.

Hat man sowohl eine Herleitung für ϕ als auch eine für $\neg\phi$, dann hat man den Widerspruch bewiesen, kann daraus eine beliebige Formel folgern und hat die Negation entfernt. Dass aus dem Widerspruch jede beliebige Aussage folgt, wird auch als *Ex falso quodlibet* oder genauer *Ex falso sequitur quodlibet* bezeichnet.

Oft wird die Regel zerlegt in zwei Regeln: $\dfrac{\phi \quad \neg\phi}{\bot}$ und $\dfrac{\bot}{\phi}$

Tertium non datur

Die bisher diskutierten Regeln gelten für die intuitionistische Logik. Aus ihnen kann man das Gesetz des ausgeschlossenen Dritten (*Tertium non datur*) nicht herleiten.

Fügt man dieses als Regel hinzu

> $$\frac{}{\phi \lor \neg\phi}\ \text{TND}$$

erhält man die Regeln für das natürliche Schließen in der klassischen Logik.

Prädikatenlogik

Zusätzliche Regeln für die Prädikatenlogik:

Allquantor

	Einführung	*Elimination*
$\forall$	$\dfrac{\boxed{\begin{array}{l} x_0 \\ \quad\vdots \\ \quad \phi[x_0/x] \end{array}}}{\forall x \phi}\ \forall\text{x i}$	$\dfrac{\forall x \phi}{\phi[t/x]}\ \forall\text{x e}$

Um den Allquantor einzuführen, hat man folgende Beweisverpflichtung: Gegeben sei ein beliebiges Objekt x_0 des Universums. Man muss dann zeigen, dass die Formel ϕ mit x_0 an Stelle der Variablen x gilt (dies schreibt man kurz als $\phi[x_o/x]$). Dabei darf in dieser Herleitung keinerlei spezielle Eigenschaft von x_0 vorkommen, denn x_0 steht ja für ein *beliebiges* Objekt des Universums. Man sagt auch, dass x_0 ein *frisches* beliebiges Objekt ist, sein Name darf somit nicht außerhalb der Box vorkommen.

Die Entfernung des Allquantors ist ein naheliegender Schritt: Wenn ϕ für alle x gilt, dann kann man ein beliebiges konkretes t des Universums an Stelle von x in die Formel ϕ einsetzen.

Existenzquantor

	Einführung	Elimination
$\exists$	$\dfrac{\phi[t/x]}{\exists x \phi}$ $\exists$x i	$\dfrac{\exists x \phi \quad \boxed{\begin{matrix} x_0 & \phi[x_0/x] \\ & \vdots \\ & \chi \end{matrix}}}{\chi}$ $\exists$x e

Den Existenzquantor kann man einführen, indem man einen *Zeugen* vorweist: Gilt ϕ mit t an Stelle von x, dann gibt es offenbar ein x für das ϕ gilt, nämlich eben t.

Will man den Existenzquantor entfernen, muss man ein beliebiges Objekt x_0 nehmen, das ϕ an Stelle von x erfüllt und hat nun die Beweisverpflichtung zu zeigen, dass daraus χ herleitbar ist. In dieser Herleitung darf man keine spezielle Aussage über x_0 verwenden, außer $\phi[x_o/x]$.

Gleichheit

	Einführung	Elimination
$=$	$\dfrac{}{t = t}$ = i, ID	$\dfrac{t_1 = t_2 \quad \phi[t_1/x]}{\phi[t_2/x]}$ = e, SUB

Die Regel ID besagt, dass ein Symbol, das für ein Objekt steht dieses eindeutig bestimmt. Dies ist gewissermaßen die Charakteristik der Gleichheit.

Die Entfernung der Gleichheit besteht darin, dass wenn t_1 und t_2 gleich sind, man in einer Formel ϕ t_1 durch t_2 ersetzen kann. Dies klingt wie selbstverständlich, muss aber mit Vorsicht gehandhabt werden. Es sind nur gültige Substitutionen erlaubt: In allen Substitutionen $\phi[t/x]$ muss t frei für x in der Formel ϕ sein, d.h. keine freie Variable y in t gelangt durch das Einsetzen von x in ϕ in den Bereich eines Quantors $\forall y$ oder $\exists y$.

Regeln des natürlichen Schließens in Lean

Aussagenlogik

Für alle Beispiele für Regeln in der Aussagenlogik geben wir vor:

```
variable (p q: Prop)
```

Implikation

Die *Einführung der Implikation* besteht darin, dass wir eine Funktion finden, die die Voraussetzung auf die Schlussfolgerung abbildet.

Als triviales Beispiel wollen wir vorgeben, dass `q` wahr ist und zeigen, dass dann `p → q` gilt.

Es gibt zwei Möglichkeiten für einen Beweis:

Im *term style* schreibt man einen Term hin, der den Typ des Beweisziels hat. In unserem ersten Beispiel ist dies einfach eine Funktion, die für ein beliebiges Argument den Beweis `hq` von `q` ergibt.

Die andere Möglichkeit ist der *tactic style*, der mit dem Schlüsselwort `by` eingeleitet wird. Die Taktik `intro` angewandt auf das Beweisziel `p → q` führt die Annahme ein, dass `p`gilt und ändert das Beweisziel zu `q`: Um `p → q` zu zeigen, muss man unter der Annahme, das `p` gilt nun `q` herleiten. In unserem Beispiel ist das trivial, denn dass `q` wahr ist, haben wir vorgegeben durch `(hq: q)`. Die Taktik `exact` gibt an, dass unser Ziel gerade der Annahme `hq` entspricht.

```
example (hq: q): p → q :=
  fun _ => hq

example (hq: q): p → q := by
  intro _
  exact hq
```

Ein interessanteres Beispiel für die Einführung der Implikation ist:

```
lemma weak_pierce: ((((p → q) → p) → p) → q) → q := by
  intro h0
  apply h0
  intro h1
  apply h1
```

```
  intro hp
  apply h0
  intro _
  exact hp
  done

#print weak_pierce
```

`intro` führt eine Annahme ein und verändert das Beweisziel. `apply` wendet eine Funktion an.

`done` ist nicht nötig, hilft aber zu sehen, ob man den Beweis fertiggestellt hat.

Das Kommando `#print` zeigt den Term, der das Lemma beweist.

Die *Auflösung der Implikation* ist einfach die Anwendung der entsprechenden Funktion:

```
example (hp: p) (hpq: p → q) : q := by
  apply hpq
  exact hp
  done

example (hp: p) (hpq: p → q) : q := by
  apply hpq at hp
  exact hp
```

Im ersten Beispiel für die Auflösung der Implikation wird die Taktik `apply` *rückwärts*, d.h. vom Ziel her rückwärts verwendet, im zweiten Beispiel *vorwärts* unter Verwendung der gegebenen Voraussetzung `hp`.

Es ist in Lean möglich, den *term style* und den *tactic style* zu mischen und ebenso *rückwärts* oder *vorwärts* herzuleiten. Das werden wir in den folgenden Beispielen noch sehen.

Äquivalenz

Lean hat auch Regeln für die *Einführung und Auflösung der Äquivalenz*:

```
example: p ↔ p := by
  apply Iff.intro
  . intro hp; exact hp
  . intro hp; exact hp

example (hp: p): (p ↔ q) → q := by
```

```
  intro hpq
  apply hpq.mp
  exact hp
  done

example (hq: q): (p ↔ q) → p := by
  intro hpq
  apply hpq.mpr
  exact hq
```

`Iff.intro` ergibt zwei Ziele: die Implikation von links nach rechts und umgekehrt. Der Punkt leitet jeweils den ersten bzw. den zweiten Fall des Beweises ein.

Hat man eine Äquivalenz wie `hpq`, dann ist `hpq.mp` die Implikation (der Modus Ponens) von links nach rechts und `hpq.mpr` die umgekehrte Richtung.

Konjunktion

Für die *Einführung der Konjunktion* haben wir `And.intro`.

```
#check And.intro
#print And.intro

-- term style
example (hp: p) (hq: q): p ∧ q :=
  And.intro hp hq

-- tactic style
example (hp: p) (hq: q): p ∧ q := by
  apply And.intro hp hq

-- tactic style mit Konstruktor
example (hp: p) (hq: q): p ∧ q := by
  constructor
  exact hp
  exact hq

example (hp: p) (hq: q): p ∧ q := by
  exact ⟨hp, hq⟩

example (hp: p) (hq: q): p ∧ q := by
  constructor <;> assumption
```

Die *Auflösung der Konjunktion* geht mit `And.left` und `And.right`.

```
example (hpq: p ∧ q): p := by
  exact hpq.left

example (hpq: p ∧ q): q := by
  exact hpq.right
```

Disjunktion

Für die *Einführung der Disjunktion* gibt es den Ausdruck `Or.intro_left q hp`, welcher einen Beweis für `p ∨ q` aus einem Beweis `hp: p` ergibt. Analog gibt es `Or.intro_right`.

In der Regel kann Lean das erste Argument dieser Funktion automatisch ermitteln, deshalb kann man

- `Or.inl` als Abkürzung für `Or.intro_left _` und
- `Or.inr` als Abkürzung für `Or.intro_right _`

verwenden. Auch möglich sind die Taktiken `left` und `right`.

```
example (hp: p): p ∨ q := by
  apply Or.inl hp

example (hp: p): p ∨ q := by
  left
  exact hp

example (hq: q): p ∨ q := by
  apply Or.inr hq

example (hq: q): p ∨ q := by
  right
  exact hq
```

Für die *Auflösung der Disjunktion* gibt es `Or.elim hpq hpr hqr` mit den drei Argumenten `hpq: p ∨ q`, `hpr: p → r` und `hqr: q → r` das die Fallunterscheidung durchführt und für beide Fälle `r` herleitet, also `r` beweist.

```
example (p q r : Prop) (hpq : p ∨ q) (hpr : p → r) (hqr : q → r) : r := by
  apply Or.elim hpq
  . intro hp -- case left
    apply hpr
    exact hp
```

```
  . intro hq -- case right
    apply hqr
    exact hq
```

Man kann stattdessen auch `cases` oder `cases'` einsetzen:

```
example (p q r : Prop) (hpq : p ∨ q) (hpr : p → r) (hqr : q → r) : r := by
  cases hpq with
  | inl hp  =>
    exact hpr hp
  | inr hq =>
    exact hqr hq

example (p q r : Prop) (hpq : p ∨ q) (hpr : p → r) (hqr : q → r) : r := by
  cases' hpq with hp hq
  . exact hpr hp
  . exact hqr hq
```

Negation

Die Negation `¬ p` ist in Lean definiert als `p → False`. Die *Einführung der Negation* ergibt sich somit aus der Einführung der Implikation, hier: man erhält `¬ p`, indem man aus `p` den Widerspruch herleitet.

```
example (hpq: p → q) (hnq: ¬ q): ¬ p := by
  intro hp
  apply hnq
  apply hpq
  exact hp

example (hpq: p → q) (hnq: ¬ q): ¬ p := by
  intro hp
  apply hpq at hp
  apply absurd hp hnq
```

Für die *Auflösung der Negation* gibt es die Regel `False.elim`, die *ex falso quodlibet* ausdrückt. Die Funktion `absurd` oder die Taktik `exfalso` können für die Auflösung der Negation verwendet werden.

```
example (hp: p) (hnp: ¬ p): q := by
  apply False.elim (hnp hp)

example (hp: p) (hnp: ¬ p): q := by
```

```
  apply absurd hp hnp

example (hp: p) (hnp: ¬ p): q := by
  exfalso
  apply hnp
  exact hp
```

Weitere Taktiken für die klassische Logik

Die Regeln, die wir bisher betrachtet haben, gelten für die intuitionistische Logik. In Lean kann man das natürliche Schließen auch für die klassische Logik mit der Regel des ausgeschlossenen Dritten (*Tertium non datur*) einsetzen.

Die Taktik `by_cases` ergibt für eine Formel die Disjunktion mit ihrer Negation. Die Taktik verwendet das Theorem `Classical.em` aus dem Namensraum `Classical` von Lean.

```
variable (r: Prop)

example: p ∨ q → ¬ p ∨ r → q ∨ r := by
  intro hpq hnpr
  by_cases tndp: p
  . apply Or.elim hnpr
    . intro hnp
      apply absurd tndp hnp
    . intro hr
      apply Or.inr hr
  . apply Or.elim hpq
    . intro hp
      apply absurd hp tndp
    . intro hq
      apply Or.inl hq
```

Die Taktik `by_contra` verwendet man um einen Widerspruchsbeweis per *reductio ad absurdum* zu machen:

```
example: ¬ ¬ p → p := by
  intro hnnp
  by_contra hnp
  apply hnnp
  exact hnp

example: ¬ ¬ p → p := by
```

```
  intro hnnp
  by_contra hnp
  apply absurd hnp hnnp
```

Prädikatenlogik

In der Prädikatenlogik brauchen wir Objekte, über die wir etwas aussagen wollen und Prädikate. Für die Objekte definieren wir eine Variable `α`, die einen Typ repäsentiert und ferner `p` und `q` als unäre Prädikate.

```
variable (α: Type) (p q: α → Prop)
```

Allquantor

Die *Einführung des Allquantors* besteht darin, dass wir für ein beliebiges Objekt `x: α` zeigen, dass `p x` gilt. In Lean bedeutet das also einfach, wir haben für ein beliebiges `x: α` eine Funktion `(x: α) → p`.

```
example: (∀ x : α, p x ∧ q x) → ∀ y : α, p y := by
  intro hpq y
  exact (hpq y).left
  done
```

Bei der *Auflösung des Allquantors* erhalten wir aus `∀ x, p x` für ein spezielles `t : α` `p t`:

```
example (t: α) (h: ∀ x, p x): p t := by
  apply (h t)
  done

example (f: ℕ → ℕ) (h : ∀n : ℕ, f n = n) : f 42 = 42 := by
  exact h 42
```

Existenzquantor

Die *Einführung des Existenzquantors* gelingt dadurch, dass man einen Zeugen für die Aussage vorzeigt:

```
example: ∃ x: ℕ , x > 0 := by
  have h: 1 > 0 := by apply Nat.zero_lt_succ 0
  apply Exists.intro 1 h
```

```
example: ∃ x: ℕ , x > 0 := by
  have h: 1 > 0 := by apply Nat.zero_lt_succ 0
  exact ⟨1, h⟩

example: ∃ x: ℕ , x > 0 := by
  have _: 1 > 0 := by apply Nat.zero_lt_succ 0
  exists 1

example: ∃ x: ℕ , x > 0 := by
  exists 1

example: ∃ x: ℕ , x > 0 := by
  use 1
  decide
```

Bei der *Auflösung des Existenzquantor* durch `Exists.elim` erhalten wir einen Zeugen für die Aussage:

```
example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x := by
  apply Exists.elim h
  intro w
  intro hw
  exact ⟨w, ⟨hw.right, hw.left⟩⟩
  done

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x := by
  cases' h with w hpq
  exact ⟨w, ⟨hpq.right, hpq.left⟩⟩
  done

example (h : ∃ x, p x ∧ q x) : ∃ x, q x ∧ p x := by
  let ⟨w, ⟨hp, hq⟩⟩  := h
  exact ⟨w, ⟨hq, hp⟩⟩
  done
```

Gleichheit

Gleichheit ist eine Äquivalenzrelation:

```
universe u

#check @Eq.refl.{u}
```

```
  -- @Eq.refl : ∀ {α : Sort u} (a : α), a = a
#check @Eq.symm.{u}
  -- @Eq.symm : ∀ {α : Sort u} {a b : α}, a = b → b = a
#check @Eq.trans.{u}
  -- @Eq.trans : ∀ {α : Sort u} {a b c : α}, a = b → b = c → a = c
```

Substitution macht man in Lean mit der Taktik rw (*rewrite*):

```
example (a b: α) (hab: a = b) (hpa: p a): p b := by
  rw [← hab]
  exact hpa
  done

example (a b: α) (hab: a = b) (hpa: p a): p b := by
  rw [hab] at hpa
  exact hpa
  done
```

Die Taktik rw kann nicht nur Gleichheit von Objekten zur Substitution verwenden, sondern kann auch verwendet werden, um Äquivalenzen von Aussagen einzusetzen.

Beispiele von Gerhard Gentzen

In seiner Arbeit über das natürliche Schließen (Gentzen, Gerhard: Untersuchungen über das logische Schließen. I. In: Mathematische Zeitschrift 39 (1935), S. 176–210) illustriert Gerhad Gentzen das Kalkül an drei Beispielen, die wir jetzt in Lean herleiten wollen.

```
example (x y z: Prop): (x ∨ (y ∧ z)) → ((x ∨ y) ∧ (x ∨ z)) := by
  intro h
  apply Or.elim h
  . intro hx
    have hxy: x ∨ y := Or.inl hx
    have hxz: x ∨ z := Or.inl hx
    exact ⟨hxy, hxz⟩
  . intro hyz
    have hxy: x ∨ y := Or.inr hyz.left
    have hxz: x ∨ z := Or.inr hyz.right
    exact ⟨hxy, hxz⟩
  done
```

In diesem Beispiel haben wir mit have die Einführung der Disjunktion vorwärts angewandt und dabei den *term style* verwendet.

```
example (α β: Type) (f: α → β → Prop):
  (∃ x: α, ∀ y : β, f x y) → (∀ y : β, ∃ x: α , f x y) := by
  intro h
  apply Exists.elim h
  intro wa
  intro ha
  intro hb
  exact ⟨wa, (ha hb)⟩
  done

example (α: Type) (g: α → Prop): (¬ ∃ x, g x) → (∀ y, ¬ g y) := by
  intro h
  intro x hp
  apply h
  exact ⟨x, hp⟩
  done
```

5 Charakterisierungen der klassischen Logik

Die klassische Logik kann auf folgende Weisen charakterisiert werden:

> Peirces Gesetz

```
def Peirce : Prop :=
 ∀ p q : Prop, ((p → q) → p) → p
```

> Doppelte Negation

```
def DoubleNeg: Prop :=
  ∀ p : Prop, ¬ ¬ p → p
```

> Der Satz vom ausgeschlossenen Dritten (*Tertium non datur*)

```
def TND : Prop :=
  ∀ p : Prop, p ∨ ¬ p
```

> De Morgans Gesetz

```
def DeMorgan : Prop :=
  ∀ p q : Prop, ¬ (¬ p ∧ ¬ q) → p ∨ q
```

> Die Definition der Implikation

```
def ImplDef : Prop :=
  ∀ p q : Prop, (p → q) → (¬ p ∨ q)
```

Im Folgenden wird die Äquivalenz all dieser Charakterisierungen bewiesen. Dabei dürfen wir natürlich nur die Regeln der intutionistischen Logik verwenden, denn in der klassischen Logik sind das ja ohnehin Tautologien.

```
lemma Pierce_DoubleNeg : Peirce → DoubleNeg := by
  rw [DoubleNeg]
  intro hPeirce p hnnp
  apply hPeirce p False
  intro h1
  exfalso
  apply hnnp
  exact h1

lemma DoubleNeg_TND : DoubleNeg → TND := by
  rw [TND]
  intro hDoubleNeg p
  apply hDoubleNeg
  intro h
  have hnp: ¬ p := by
    intro hp
    apply h
    apply Or.inl hp
  have hp: p := by
    apply hDoubleNeg
    intro hnp
    apply h
    apply Or.inr hnp
  apply absurd hp hnp

lemma TND_DeMorgan : TND → DeMorgan := by
  intro hTND
  rw [DeMorgan]
  intro p q h
  have emp: p ∨ ¬ p := hTND p
  have emq: q ∨ ¬ q := hTND q
  apply Or.elim emp
  . intro hp
    apply Or.inl hp
  . intro hnp
```

```
    apply Or.elim emq
    . intro hq
      apply Or.inr hq
    . intro hnq
      have andnpnq: ¬ p ∧ ¬ q := ⟨hnp, hnq⟩
      apply absurd andnpnq h

lemma DeMorgan_ImplDef: DeMorgan → ImplDef := by
  intro hDeMorgan
  rw [ImplDef]
  intro p q h
  apply hDeMorgan
  intro ⟨hnnp, hnq⟩
  apply hnnp
  intro hp
  apply hnq
  apply h
  exact hp

theorem ImplDef_Pierce : ImplDef → Peirce := by
  intro hImplDef
  rw [Peirce]
  intro p q h0
  have emp: ¬ p ∨ p := by
    apply hImplDef
    intro hp; exact hp
  apply Or.elim emp
  . intro hnp
    apply h0
    intro hp
    apply absurd hp hnp
  . intro hp; exact hp
  done
```

Das Professoren-Paradoxon

Zum Abschluss dieser kleinen Einführung noch als Spaß das berüchtigte „Trinker-Paradoxon“:

Es gibt einen Professor, für den gilt, dass alle anderen Professoren betrunken sind, wenn er betrunken ist.

Man kann für die Aussagen über den Alkoholspiegel einer Gruppe von Professoren natürlich über jede andere Gruppe von Menschen genauso machen. Sie erscheint paradox, weil dem gesunden Menschenverstand fremd ist, wie die Implikation zu bewerten ist, wenn die Voraussetzung gar nicht zutrifft.

Nehmen wir also an, dass wir einen Typ `Prof` haben und dass es mindestens einen Professor gibt. Dann können wir zeigen:

```
lemma Drunken_Profs (Prof: Type) (p: Prof) (Drunken: Prof → Prop):
  ∃ x: Prof, (Drunken x → ∀ y: Prof, Drunken y) := by
  by_cases tnd: ∀ y: Prof, Drunken y
  . have hpt: Drunken p → ∀ y: Prof, Drunken y := by
      intro _
      exact tnd
    exact ⟨p, hpt⟩
  . rw [Classical.not_forall] at tnd
    let ⟨w, h⟩ := tnd
    have ht: Drunken w → ∀ y: Prof, Drunken y := by
      intro hd
      apply absurd hd h
    exact ⟨w, ht⟩
  done
```

Und damit wollen wir die Gruppe der lustigen Professoren verlassen, sie vergnügen sich weiter mit logischen Spielchen und der *veritas*, die bekanntlich im *vino* ist.