GLoIn - Einführung in Rocq

Thorsten Wißmann, Lehrstuhl für Informatik 8, FAU (`https://www8.cs.fau.de`)

Stand: 12. Oktober 2025

In interaktiven Theorembeweisern lassen sich mathematische Beweise aufschreiben und noch während des Schreibens auf Korrektheit prüfen. Ein solcher Theorembeweiser ist die freie Software Rocq (`https://rocq-prover.org/`, ehemals unter dem Namen *Coq*).

Beginnen wir mit einem Beispiel eines Beweises der Tranisitivät der Implikation ($\{A \to B, B \to C\} \vdash A \to C$) mittels natürlichen Schließens und des entsprechenden Beweises in Rocq:

1	$A \to B$	
2	$B \to C$	
3	$\quad A$	
4	$\quad B$	$(\to E)$ 1, 3
5	$\quad C$	$(\to E)$ 2, 4
6	$A \to C$	$(\to I)$ 3–5

```
Parameters A B C : Prop.

Lemma imp_transitive :
  (A → B) → (B → C) → (A → C).
Proof.
  intro a2b. (* a2b : A → B *)
  intro b2c. (* b2c : B → C *)
  intro a.   (* a : A *)
  (* aktuelles Ziel: C *)
  apply b2c.
  (* aktuelles Ziel: B *)
  apply a2b.
  (* aktuelles Ziel: A *)
  exact a.
Qed.
```

Alles zwischen `(*... *)` sind lediglich Kommentare. Ein paar Beobachtungen:

- In der Fitch-Notation nutzen wir Zeilennummern um auf vorherige Annahmen oder Schlussfolgerungen zuzugreifen, in Rocq verwenden wir Label (`a2b`,`b2c`, `a`).
- Die rechts abgesetzten Kommandos `intro`, `apply` etc. heißen *Taktiken*.
- Damit Zeilen wirklich ausgeführt werden, müssen sie mit einem Punkt abgeschlossen werden.
- Die Taktik `intro` entspricht beim aktuell verwendeten Sprachumfang gerade der Einführungsregel für Implikation. Sie macht also einen Unterbeweis auf, der das Antezedenz der Implikation (im Schritt `intro a2b` z.B. die Formel $A \to B$) als Annahme bekommt, und in dem wir das Sukzedens der Implikation (im Schritt `intro a2b` z.B. die Formel C) beweisen müssen. Die zu beweisende Formel wird von Rocq dabei explizit als Beweisziel vorgeblendet; daher tragen Unterbeweise in Rocq die Bezeichnung *subgoal*. Das Argument der Taktik `intro` gibt das Label der Annahme des neuen Unterbeweises vor, unter dem wir später auf die Annahme verweisen (d.h. wie angedeutet ersetzen die Label die Zeilennummern). Das Argument ist optional; wenn wir es weglassen, denkt Rocq sich selbst einen Label aus.
- Die Taktik `apply` entspricht der Eliminationsregel für Implikation; sie transformiert das aktuelle Beweisziel aber *rückwärts*. Generell kann man beim Aufbau von Beweisen von den Zielen ausgehend rückwärts vorgehen (*backward proof*) oder von den Annahmen ausgehend vorwärts (*forward proof*). Schon die oben diskutierte Taktik `intro` arbeitet rückwärts: Ausgehend von einer Implikation als Beweisziel führt sie die Prämisse der Implikationseinführungsregel (einen Unterbeweis)

als Beweisziel ein. Ähnlich transformiert die Taktik `apply` ein Beweisziel B rückwärts in ein Beweisziel A, wenn sie als Argument den Label einer Implikation $A \to B$ bekommt. In der Tat passiert genau dies (d.h. auch in denselben Bezeichnern) im letzten Schritt `apply a2b`.

Wie bereits angedeutet, gibt es innerhalb eines Rocq-Beweises zu jedem Zeitpunkt ein konkretes Beweisziel (also das, was gerade zu zeigen ist), das in Rocq auch als *subgoal* anzeigt wird. Wenn keine Beweisziele mehr abhzuhaken sind ('*no subgoals*'), ist der Beweis fertig, und wir können ihn mit `Qed` abschließen. In unserem Beispielbeweis entwickeln sich die Ziele wie folgt:

1. Zu Beginn des Beweises (also direkt nach `Proof.`) ist das Beweisziel genau die Behauptung des Lemmas:

   `(A → B) → (B → C) → (A → C)`

2. Um eine solche Implikation zu beweisen, würde man auf Papier so etwas schreiben wie „Sei $H : A \to B$; ...". Wie oben diskutiert, entspricht diese Vorgehensweise der Taktik `intro`. Nach Ausführung von `intro a2b` vereinfacht sich unser Beweisziel zu

   `(B → C) → (A → C)`

   und gleichzeitig steht uns im folgenden `a2b` zu Verfügung, das man verstehen kann als „Namen für einen Beweis, dass `A→ B` gilt". Deshalb erscheint in der IDE die Zeile

   `a2b : A → B`

   im Kontext, also in der Liste der Annahmen.

3. Nachdem das Ziel immer noch eine Implikation ist, verwenden wir analog `intro b2c`, wodurch sich das Ziel vereinfacht zu

   `A → C`

   während wir eine neue Annahme

   `b2c : B → C`

   bekommen.

4. Im nächsten Schritt verfahren wir entsprechen und bekommen eine neue Annahme `a : A` und das Beweisziel `C`.

5. Nach Ausführung von `apply b2c` und `apply a2b` verändert sich das Beweisziel zunächst zu `B` und dann zu `A`.

6. Jetzt sind wir in einer Situation, in der das Beweisziel mit einer der Annahmen übereinstimmt; wir weisen Rocq mit `exact a` auf diese Lage hin, woraufhin das Beweisziel verschwindet.

Auch nach Fertigstellung des Beweises können wir in Rocq im Beweis vor und zurückspringen, um uns das jeweils aktuelle Ziel anzeigen zu lassen.

1 Übersetzung zwischen natürlichem Schließen und Rocq

1.1 Konzepte

Natürliches Schließen (via Fitch)	Rocq
Regeln	Taktiken
Zeilennummern (0, 1, 2, ...)	Label (a, b, ...)
Unterbeweise	Subgoals

1.2 Formeln

Natürliches Schließen	Rocq
$\rightarrow$	`->`
$\wedge$	`/\`
$\vee$	`\/`
$\bot$	`False` (wird oft als $\bot$ angezeigt)
$\top$	`True` (wird oft als $\top$ angezeigt)
$\neg A$	`~A` (ist per Definition `A -> False`)
$A \leftrightarrow B$	`A <-> B` (ist per Definition `(A -> B) /\(B -> A)`)

Im System natürlichen Schließens haben wir für $\leftrightarrow$ auch keine expliziten Einführungs- und Eliminationsregeln definiert. Stattdessen definieren wir üblicherweise $\phi \leftrightarrow \psi$ als Notation für $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

1.3 Regeln vs. Taktiken

Natürliches Schließen	Rocq

($\wedge$I)	`split.`
	`– (* Rocq-Beweis des ersten Konjunkts *)`
	`– (* Rocq-Beweis des zweiten Konjunkts *)`

($\wedge E_1$) / ($\wedge E_2$)	`destruct x as [a b]`
	`(* Wenn 'x : A ∧ B', dann sind danach 'a : A' und`
	`'b : B' im Kontext. Man kann statt a oder b auch`
	`_ schreiben, um entsprechende Seite zu verwerfen. *)`

($\vee I_1$)	`left.`
($\vee I_2$)	`right.`

($\vee$E)	`(* Sei x : A ∨ B im Kontext: *)`
	`destruct x as [a|b].`
	`– (* Jetzt ist a : A im Kontext *)`
	`(* Fall 1: (gleiches Ziel, aber mehr Kontext.) *)`
	`– (* Jetzt ist b : B im Kontext *)`
	`(* Fall 2: (gleiches Ziel, aber mehr Kontext.) *)`

($\bot$E)	`exfalso` (falls man `x : False` bereits im Kontext hat, dann ist `destruct x` schneller, weil es das Beweisziel sofort erfüllt)
($\rightarrow$I)	`intro`
($\rightarrow$E)	`apply` (siehe auch `apply as in` später)
($\neg$I)	`intro` (da $\neg$ in Rocq als .. $\rightarrow \bot$ definiert ist)
($\neg$E)	`contradiction` oder wenn man `na : ~A` im Kontext hat, dann auch `apply na` gefolgt vom Nachweis von `A`.
($\neg\neg$E)	`apply NNPP.`
Reiteration	`exact x.`

Vielleicht ist Ihnen aufgefallen, dass Eliminations-Regeln in Rocq mit `destruct` bezeichnet werden. Beachten Sie aber, dass bei ($\wedge E_1$) / ($\wedge E_2$) das `as [a b]` leicht anders aussieht als `as [a|b]` bei ($\vee$E). Das `|` trennt Fälle (also mehrere Beweisziele) voneinander, und das Leerzeichen steht zwischen Namen, die im selben (Unter-)Ziel zur Verfügung stehen. Bei ($\wedge E_1$) / ($\wedge E_2$) bleibt es bei einem Ziel, in welchem wir fortan `a` als auch `b` zur Verfügung haben, also werden `a` und `b` durch ein Leerzeichen getrennt. Bei ($\vee$E) landen `a` und `b` als Annahmen in unterschiedlichen Unterbeweisen: einem für `a` und einem für `b`.

1.4 Beispiel

Hier ist ein Beispiel eines Rocq-Beweises für $A \vee ((A \to A) \to A) \vdash A$:

```
Parameters A B C : Prop.   (* A,B,C nutzen wir als propositionale Variablen *)

Lemma bsp :                (* Wir wollen einen (Hilfs-)Satz namens 'bsp' beweisen *)
  A ∨ ((A → A) → A)        (* Unter dieser Annahme ... *)
  → A.                     (* ... soll A gelten. *)
Proof.                     (* Der Beweis beginnt: *)
  intro H.                 (* Wir geben der Annahme den Namen 'H' *)
  destruct H as [a|I].     (* Wir unterscheiden die zwei Fälle von ∨ *)
  * exact a.               (* Fall 1: wir haben direkt die Annahme a : A *)
                           (* und nutzen sie um das Ziel A zu zeigen *)
  *                        (* Fall 2: I : (A → A) → A im Kontext *)
    apply I.               (* Das Ziel wird zu: (A → A) *)
    intro x.               (* wir erhalten x : A im Kontext und das Ziel wird zu A *)
    exact x.               (* und nutzen genau diese Annahme um das Ziel zu zeigen.*)
Qed.                       (* Wir haben beide Fälle bewiesen und somit auch das *)
                           (* eigentliche Ziel; deshalb können wir den Beweis mit *)
                           (* 'Qed.' schließen. *)
```

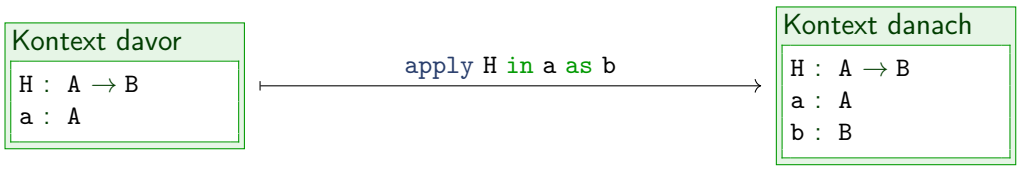
2 Bottom-Up statt Top-Down

Während man beim natürlichen Schließen von den Annahmen zu den Zielen argumentiert (Bottom-Up oder *forward proof*), kann man in Rocq mit den Taktiken das aktuelle Ziel modifizieren (*backward proof*). Manchmal ist es aber trotzdem nötig, in Rocq etwas explizit aus den Annahmen zu konstruieren. Hierfür gibt es folgende Möglichkeiten:

- Die Taktik `assert` erlaubt es, beliebige Aussagen als Unterbeweis („Hilfs-Lemma“) zu zeigen und anschließend für das eigentliche Beweisziel einzusetzen. Ein solches `assert` folgt diesem Code-Muster:

  ```
  assert (Aussage) as H. {
    (* Hier ein Beweis der Aussage. *)
    (* Der Beweis muss vor der schließenden geschweiften Klammer abgeschlossen sein: *)
  }
  (* Ab hier ist zusätzlich 'H : Aussage' im Kontext. *)
  ```

- Die Taktik `apply H in a as b` entspricht direkt der Regel (→E). Man kann sie anwenden, wann immer man `H : A → B` und `a : A` im Kontext hat. Das Ergebnis der Anwendung von `H` auf `a : A` hat den Typ `B` und wird unter dem Namen `b` in den Kontext aufgenommen:



Hier ist ein Beispiel, dass beide Konzepte verwendet:

```
Parameters A B C D : Prop.

Lemma bsp_assert_app : (A ∨ B → C ∧ D) → (A → C).
Proof.
  intro imp.
  intro a.
  (* Wir zeigen A ∨ B als Zwischenergebnis, um dann
```

```
    'imp : A ∨ B → C ∧ D' darauf anzuwenden. *)
  (* Noch ist das Ziel: C *)
  assert (A ∨ B) as aob. {  (* Jetzt ist das Ziel: A ∨ B *)
    left.                   (* Jetzt ist das Ziel: A *)
    exact a.
  }   (* Jetzt ist 'aob : A ∨ B' im Kontext. *)
  apply imp in aob as cd. (* Jetzt ist 'cd : C ∧ D' im Kontext. *)
  destruct cd as [c _]. (* Jetzt ist 'c : C' im Kontext. *)
  (* Wir benötigen nichts vom Typ 'D', deshalb steht in [c _] statt
     d einfach _. Dann wird der rechte Teil der Konjunktion verworfen. *)
  exact c.
Qed.
```

3 Prädikatenlogik

Um mathematische Aussagen in Rocq formalisieren und beweisen zu können, bietet Rocq im Sprachumfang All- und Existenzquantifizierung. Da solche Aussagen jedoch häufig über Prädikatenlogik erster Stufe hinausgehen, müssen wir der Grundmenge, über die wir quantifizieren, einen expliziten Namen geben, z. B. `M`. Hierbei müssen wir erzwingen, dass `M` nicht leer ist, indem wir ein Element `c` annehmen. Deshalb werden die Rocq-Dateien immer wie folgt beginnen:

```
Parameter M : Set.
Parameter c : M. (* erzwingt, dass M nichtleer ist *)
```

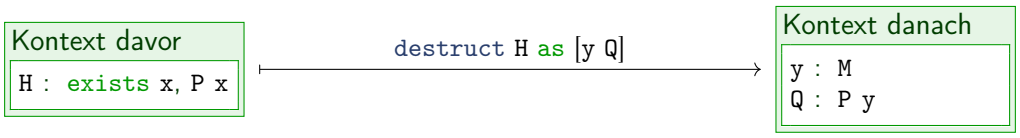
Die Funktions- und Prädikatensymbole der Signatur Σ werden dann zu Funktionen bzw. Prädikaten auf der Grundmenge `M`. Beispielsweise übersetzt sich die Signatur $\Sigma = \{E/1, L/2, \mathsf{zero}/0, \mathsf{suc}/1, \mathsf{add}/2\}$ wie folgt:

```
Parameter E : M → Prop.
Parameter L : M → M → Prop.
Parameter zero : M.
Parameter suc : M → M.
Parameter add : M → M → M.
```

Terme und Prädikaten über solchen Termen, wie z. B. `L zero (suc zero)` für $L(\mathsf{zero}, \mathsf{suc}(\mathsf{zero}))$ lassen sich dann nach belieben zusammenbauen (mit Leerzeichen statt Klammern und Kommata). Formeln mit Quantoren übersetzen sich dann direkt in Rocq:

Natürliches Schließen	Rocq
$\forall x.$	`forall x,` (oder `forall (x:M),`)
$\exists x.$	`exists x,` (oder `exists (x:M),`)
$=$	`=`

Üblicherweise ist es nicht nötig, die Grundmenge `M` in den Quantoren explizit anzugeben, da Rocq dies aus der Verwendung von `x` ableiten kann (beispielsweise in `forall x , E (add x zero)`). Ähnlich wie bei Aussagenlogik agieren die Rocq-Taktiken für Quantoren Bottom-Up, diese sind in Tabelle 1 aufgelistet. Die Elimination von $\exists$ kann man so zusammenfassen:



3.1 Beispiele aus dem Skript

Gleichheit ist symmetrisch:

Natürliches Schließen	Rocq
(∀I)	`intro x.` `(* Wenn das Beweis-Ziel eine Allquantifizierung ist, dann "wechselt" das 'intro' in den entsprechenden Unterbeweis für frisches 'x'. *)`
(∀E)	`apply H with (x:=T1) (y:=T2).` `(* Versucht ein allquantifiziertes H anzuwenden, indem T1 für x und T2 für y in H eingesetzt wird. Diese Parameter für x, y, etc können weggelassen werden, wenn sie sich aus dem Ziel ergeben. *)`
(∃I)	`exists x. (* anschließend folgt ein Beweis, dass 'x' tatsächlich die gewünschte Eigenschaft hat *)`
(∃E)	`destruct H as [y H'].` `(* Wenn "H : exists x , P x" im Kontext ist, dann ist nach dem 'destruct' sowohl 'y' als auch "H' : P y" im Kontext *)`
(=I)	`reflexivity. (* beweist lediglich 'x = x' *)`

(=E)	`rewrite H. (* Wendet die Gleichung im Ziel an. *)`
	`rewrite H in Q. (* Wendet die Gleichung in Q an. *)`
	`rewrite ← H. (* Wendet die gespiegelte Gleichung an *)`
	`rewrite H with (x := c). (* setzt c in 'forall x' und wendet dann an *)`
	`symmetry. (* spiegelt das Ziel *)`
	`symmetry in H. (* spiegelt die Annahme H *)`

Tabelle 1: Rocq-Taktiken für Schlussregeln in Prädikatenlogik

```
Parameter M : Set.
Parameter c : M.
Parameter E D : M.
Lemma symmetrisch : E = D → D = E.
Proof. (* Ziel : E = D → D = E *)
  intro H. (* H : E = D; Ziel : D = E *)
  rewrite H. (* Ziel : D = D *)
  reflexivity.
Qed.
```

Bezüglich Vertauschung von ∃ und ∀:

```
Parameter M : Set.
Parameter c : M.
Parameter P : M → M → Prop. (* Eine zweistellige Relation *)

Lemma ex_over_all :
  (exists a , forall b , P a b) → (forall y , exists x , P x y).
Proof.
  intro H. (* H : exists a , forall b , P a b *)
  intro y. (* y : M *)
  destruct H as [c Q]. (* c : M ; Q : forall b , P c b *)
  exists c. (* c ist der Zeuge für x. *)
  apply Q. (* kurz für: apply Q with (b := y). *)
Qed.
```

Für die Äquivalenz zwischen $\exists x$ und $\neg\forall\neg$ benötigt man die klassische Regel der Doppelnegationselimination:

```
Require Export Classical_Prop.
Parameter M : Set.
Parameter c : M.
```

```
Parameter E : M → Prop.
Lemma classic_ex : (∼forall x, ∼E x) → exists y, E y.
Proof.
  intro notall. (* notall : ∼(forall x : M, ∼E x) *)
  apply NNPP.
  intro notex. (* notex : ∼(exists y : M, E y) *)
  apply notall.
  intro x.
  intro x_is_E. (* x_is_E: E x *)
  assert (exists c , E c). {
    exists x.
    exact x_is_E.
  }
  contradiction.
Qed.
```