Natürliches Schließen

Teil 4: Programme als Beweise

Die Curry-Howard-Korrespondenz

Die Curry-Howard-Korrespondenz ist eine Beziehung zwischen Aussagen und Typen. Jedem Junktor entspricht hierbei eine ganz bestimmte Typkonstruktion.

Die Curry-Howard-Korrespondenz ist eine Beziehung zwischen Aussagen und Typen. Jedem Junktor entspricht hierbei eine ganz bestimmte Typkonstruktion.

Aussage	**Typ**
Konjunktion — $A \wedge B$	$A \times B$ — Produkt
Disjunktion — $A \vee B$	$A + B$ — Summe
Implikation — $A \to B$	$A \to B$ — Funktionentyp
Kontradiktion — $\bot$	0 — leerer Typ
Tautologie — $\top$	1 — Einheitstyp

Die Curry-Howard-Korrespondenz ist eine Beziehung zwischen Aussagen und Typen. Jedem Junktor entspricht hierbei eine ganz bestimmte Typkonstruktion.

Aussage	**Typ**
Konjunktion — $A \wedge B$	$A \times B$ — Produkt
Disjunktion — $A \vee B$	$A + B$ — Summe
Implikation — $A \to B$	$A \to B$ — Funktionentyp
Kontradiktion — $\bot$	0 — leerer Typ
Tautologie — $\top$	1 — Einheitstyp

Darüber hinaus stellt sie eine Beziehung zwischen Beweisen und Termen dar. Zu jedem Beweis gehört ein bestimmter Term. Die Schlussregeln des natürlichen Schließens spiegeln die Regeln zur Konstruktion von Termen wider.

Typen

Die Schreibweise $a : A$ drückt das Urteil aus, dass a ein Term vom Typ A sein muss.

[1] Technisches Detail: Ein Kontext soll wohlgeformt sein. Beispielsweise ist $[s : \text{string}, P : (\text{nat} \to \text{Type}), x : P(s)]$ irrsinnig. In der einfachen Typentheorie treten solche Komplikationen allerdings noch nicht auf; erst in der abhängigen.

Typen

Die Schreibweise $a : A$ drückt das Urteil aus, dass a ein Term vom Typ A sein muss.

Analog zur Logik ist ein Kontext eine Liste[1]

$$\Gamma = [(a_1 : A_1), \ldots, (a_n : A_n)],$$

wobei die a_k Variablen sind. Ist nun die Sequenz

$$\Gamma \vdash (b : B)$$

eine ableitbare, liegt das Urteil $b : B$ vor, sofern die Variablen in Γ vorausgesetzt werden dürfen. Das ist so zu verstehen, dass man mit den Variablen aus Γ den Term b zusammenbasteln kann. Eine Variable darf hierbei mehrmals benutzt werden.

Ein Objekt $b : B$ stellt einen Zeugen dafür dar, dass der Typ B von mindestens einem Objekt bewohnt ist. Anders ausgedrückt ist b ein Beweis für die Aussage B.

Die runden Klammern schreibt man nicht mit. Ich wollte nur einmal verdeutlichen, wie die Formeln zu lesen sind.

[1] Technisches Detail: Ein Kontext soll wohlgeformt sein. Beispielsweise ist $[s : \text{string}, P : (\text{nat} \to \text{Type}), x : P(s)]$ irrsinnig. In der einfachen Typentheorie treten solche Komplikationen allerdings noch nicht auf; erst in der abhängigen.

Zur Konstruktion von Termen finden sich nun Schlussregeln, die die logischen widerspiegeln.

Zur Konstruktion von Termen finden sich nun Schlussregeln, die die logischen widerspiegeln.

Axiom (Einführung von Grundsequenzen)

Aussagen	**Typurteile**
$\dfrac{}{A \vdash A}$	$\dfrac{}{\alpha : A \vdash \alpha : A}$

Schlussregeln

Konjunktion	**Paar**
$\dfrac{\Gamma \vdash A \qquad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \wedge B}$	$\dfrac{\Gamma \vdash a : A \qquad \Gamma' \vdash b : B}{\Gamma, \Gamma' \vdash (a, b) : A \times B}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$	$\dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_l(t) : A}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_r(t) : B}$

Schlussregeln

Konjunktion	**Paar**
$\dfrac{\Gamma \vdash A \qquad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \wedge B}$	$\dfrac{\Gamma \vdash a : A \qquad \Gamma' \vdash b : B}{\Gamma, \Gamma' \vdash (a, b) : A \times B}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$	$\dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_l(t) : A}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_r(t) : B}$

In Worten: Die Einführung der Konjunktion entspricht der Konstruktion des Paares. Die Beseitigung der Konjunktion entspricht der Projektion auf eine der Komponenten.

Schlussregeln

Implikation

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}$$

Funktion

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (a \mapsto b) : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma' \vdash a : A}{\Gamma, \Gamma' \vdash f(a) : B}$$

[2] Alonzo Church: *The Calculi of Lambda-Conversion*. In: *Annals of Mathematics Studies*. Nr. 6. Princeton University Press, Princeton 1941.

Schlussregeln

Implikation	**Funktion**
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$	$\dfrac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (a \mapsto b) : A \to B}$
$\dfrac{\Gamma \vdash A \to B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}$	$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma' \vdash a : A}{\Gamma, \Gamma' \vdash f(a) : B}$

In Worten: Die Einführung der Implikation entspricht der Einführung einer anonymen Funktion (Abstraktion). Die Beseitigung der Implikation entspricht der Applikation einer Funktion.

[2] Alonzo Church: *The Calculi of Lambda-Conversion*. In: *Annals of Mathematics Studies*. Nr. 6. Princeton University Press, Princeton 1941.

Schlussregeln

Implikation	**Funktion**
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$	$\dfrac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (a \mapsto b) : A \to B}$
$\dfrac{\Gamma \vdash A \to B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}$	$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma' \vdash a : A}{\Gamma, \Gamma' \vdash f(a) : B}$

In Worten: Die Einführung der Implikation entspricht der Einführung einer anonymen Funktion (Abstraktion). Die Beseitigung der Implikation entspricht der Applikation einer Funktion.

Alonzo Church schrieb $\lambda x.\, t$ anstelle von $x \mapsto t$. Für $f(x)$ schrieb er $(f\, x)$ oder kurz $f\, x$.[2] Im Laufe der Zeit verbreitete sich diese Notation in der Informatik. Die Regeln, die wir hier aufstellen, bilden den einfach typisierten λ-Kalkül mit Erweiterung um Produkte von zwei Faktoren und Summen von zwei Summanden.

[2] Alonzo Church: *The Calculi of Lambda-Conversion*. In: *Annals of Mathematics Studies*. Nr. 6. Princeton University Press, Princeton 1941.

Eine kurze Ableitung:

Aussagen

$$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A}}{\vdash A \wedge B \to A}$$

Programmterme

$$\dfrac{\dfrac{\dfrac{}{t : A \times B \vdash t : A \times B}}{t : A \times B \vdash \pi_l(t) : A}}{\vdash (t \mapsto \pi_l(t)) : A \times B \to A}$$

Eine kurze Ableitung:

Aussagen	**Programmterme**
$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A}}{\vdash A \wedge B \to A}$	$\dfrac{\dfrac{\dfrac{}{t : A \times B \vdash t : A \times B}}{t : A \times B \vdash \pi_l(t) : A}}{\vdash (t \mapsto \pi_l(t)) : A \times B \to A}$

Bemerkung. Anstelle von $t \mapsto \pi_l(t)$ kann man auch $(a, b) \mapsto a$ schreiben. Streng genommen müsste man hierfür allerdings ein erweitertes formales System definieren. Technisch wird ein unabweisbarer Musterabgleich durchgeführt. Ist t das Argument, wird t mit (a, b) abgeglichen, weshalb $a = \pi_l(t)$ sein muss.

Implementierung der Konstruktion

Der konstruierte Programmterm liefert den Beweis, dass der Typ $A \times B \rightarrow A$ ein bewohnter ist. Man kann den Term nun in Gallina formulieren und durch den Beweisassistent Coq prüfen lassen, ob die Konstruktion fehlerfrei durchgeführt wurde:

Implementierung der Konstruktion

Der konstruierte Programmterm liefert den Beweis, dass der Typ $A \times B \rightarrow A$ ein bewohnter ist. Man kann den Term nun in Gallina formulieren und durch den Beweisassistent Coq prüfen lassen, ob die Konstruktion fehlerfrei durchgeführt wurde:

Gallina

```
Definition proof1 (A B: Type): A*B -> A
  := fun t => fst t.
```

Implementierung der Konstruktion

Der konstruierte Programmterm liefert den Beweis, dass der Typ $A \times B \rightarrow A$ ein bewohnter ist. Man kann den Term nun in Gallina formulieren und durch den Beweisassistent Coq prüfen lassen, ob die Konstruktion fehlerfrei durchgeführt wurde:

Gallina

```
Definition proof1 (A B: Type): A*B -> A
  := fun t => fst t.
```

Es ginge auch so:

Gallina

```
Definition proof2 (A B: Type): A*B -> A
  := fun t => match t with (a, b) => a end.
```

Oder kurz:

Gallina

```
Definition proof3 (A B: Type): A*B -> A
  := fun '(a, b) => a.
```

Aussagen bekommen eigentlich den speziellen Typ `Prop`. Infolgedessen verändern sich Syntax und genutzte Funktionen, wobei die Mechanismen aber äquivalent bleiben.

Gallina

```
Definition proof4 (A B: Prop): A /\ B -> A
  := fun h => proj1 h.
```

Aussagen bekommen eigentlich den speziellen Typ `Prop`. Infolgedessen verändern sich Syntax und genutzte Funktionen, wobei die Mechanismen aber äquivalent bleiben.

Gallina

```
Definition proof4 (A B: Prop): A /\ B -> A
  := fun h => proj1 h.
```

Auch hier ist eine Zerlegung per Musterabgleich durchführbar:

Gallina

```
Definition proof5 (A B: Prop): A /\ B -> A
  := fun h => match h with conj a b => a end.
```

Aussagen bekommen eigentlich den speziellen Typ `Prop`. Infolgedessen verändern sich Syntax und genutzte Funktionen, wobei die Mechanismen aber äquivalent bleiben.

Gallina

```
Definition proof4 (A B: Prop): A /\ B -> A
  := fun h => proj1 h.
```

Auch hier ist eine Zerlegung per Musterabgleich durchführbar:

Gallina

```
Definition proof5 (A B: Prop): A /\ B -> A
  := fun h => match h with conj a b => a end.
```

Äquivalent zum Musterabgleich ist das Induktionsprinzip der Konjunktion:

Gallina

```
Definition proof6 (A B: Prop): A /\ B -> A
  := fun h => and_ind (fun a b => a) h.
```

Der Beweis einer Konjunktion wird hier in seine beiden Teile zerlegt, um einen oder beide nutzen zu können. Die Zerlegung wird durch eine Rückruffunktion dargestellt, die `and_ind` als erstes Argument bekommt.

Zum Verständnis der Summe $A + B$ zweier Typen A, B betrachten wir zunächst die Analogie für Mengen. Seien L, R zwei Konstanten, als *Tags* oder *Diskriminatoren* bezeichnet. Man definiert die disjunkte Vereinigung als

$$A + B := (\{\mathrm{L}\} \times A) \cup (\{\mathrm{R}\} \times B).$$

Zum Verständnis der Summe $A + B$ zweier Typen A, B betrachten wir zunächst die Analogie für Mengen. Seien L, R zwei Konstanten, als *Tags* oder *Diskriminatoren* bezeichnet. Man definiert die disjunkte Vereinigung als

$$A + B := (\{\mathrm{L}\} \times A) \cup (\{\mathrm{R}\} \times B).$$

Nun definiert man die beiden Injektionen

$$\iota_l : A \to A + B, \quad \iota_l(a) := (\mathrm{L}, a),$$
$$\iota_r : B \to A + B, \quad \iota_r(b) := (\mathrm{R}, b).$$

Weil es sich um Injektionen handelt, kann zu einem Wert aus $A + B$ eine Fallunterscheidung per Musterabgleich vorgenommen werden. Es wird dabei schlicht geprüft, welches der beiden Tags vorliegt.

Zum Verständnis der Summe $A + B$ zweier Typen A, B betrachten wir zunächst die Analogie für Mengen. Seien L, R zwei Konstanten, als *Tags* oder *Diskriminatoren* bezeichnet. Man definiert die disjunkte Vereinigung als

$$A + B := (\{\mathrm{L}\} \times A) \cup (\{\mathrm{R}\} \times B).$$

Nun definiert man die beiden Injektionen

$$\iota_l : A \to A + B, \quad \iota_l(a) := (\mathrm{L}, a),$$
$$\iota_r : B \to A + B, \quad \iota_r(b) := (\mathrm{R}, b).$$

Weil es sich um Injektionen handelt, kann zu einem Wert aus $A + B$ eine Fallunterscheidung per Musterabgleich vorgenommen werden. Es wird dabei schlicht geprüft, welches der beiden Tags vorliegt. Sinnloses Beispiel:

$$f : \mathbb{Z} + \mathbb{Z} \to \mathbb{Z}, \quad f(n) := \textbf{match } n \begin{cases} \iota_l(a) \mapsto a, \\ \iota_r(b) \mapsto b^2. \end{cases}$$

Schlussregeln

Disjunktion

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

Term der Summe

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \iota_l(a) : A + B}$$

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash \iota_r(b) : A + B}$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, a : A \vdash c : C \quad \Gamma, b : B \vdash c' : C}{\Gamma \vdash \textbf{match}\ t\ \{\iota_l(a) \mapsto c, \iota_r(b) \mapsto c'\} : C}$$

Einen Term vom Typ $A + B \to B + A$ erhält man beispielsweise wie folgt:

Einen Term vom Typ $A + B \to B + A$ erhält man beispielsweise wie folgt:

$$t \mapsto \textbf{match}\ t \begin{cases} \iota_l(a) \mapsto \iota_r(a), \\ \iota_r(b) \mapsto \iota_l(b). \end{cases}$$

Einen Term vom Typ $A + B \to B + A$ erhält man beispielsweise wie folgt:

$$t \mapsto \textbf{match}\ t \begin{cases} \iota_l(a) \mapsto \iota_r(a), \\ \iota_r(b) \mapsto \iota_l(b). \end{cases}$$

Gallina

```
Definition proof7 (A B: Type): A + B -> B + A
  := fun t =>
    match t with
    | inl(a) => inr(a)
    | inr(b) => inl(b)
    end.

Definition proof8 (A B: Prop): A \/ B -> B \/ A
  := fun t =>
    match t with
    | or_introl(a) => or_intror(a)
    | or_intror(b) => or_introl(b)
    end.
```

Abhängige Typen

Zur Ausweitung der Korrespondenz auf die Prädikatenlogik muss die Typentheorie um *abhängige Typen* erweitert werden. Ein Typ darf hier von einem Term abhängen, analog dazu wie eine Menge als indizierte Familie von einem Index abhängt. Die logische Entsprechung eines solchen Typs ist ein Prädikat. Ist P ein Prädikat, dann kann je x die Aussage $P(x)$ eine wahre oder eine falsche sein. Ein abhängiger Typ kann gleichermaßen bewohnt oder unbewohnt sein.

Im Fortgang vermitteln zwei neue Konstruktionen die Kodierung der Quantoren:

- *Pi-Typen* entsprechen Universalquantifizierungen,
- *Sigma-Typen* entsprechen Existenzquantifizierungen.

Pi-Typen

Erklärung anhand der Analogie in der Mengenlehre. Ein Produkt $A \times B$ enthält Paare (a, b) mit $a \in A$ und $b \in B$. Nun fasst man das Paar als endliche Folge auf, also als Abbildung

$$f : \{0, 1\} \to A \cup B, \quad f(0) := a, \quad f(1) := b.$$

Pi-Typen

Erklärung anhand der Analogie in der Mengenlehre. Ein Produkt $A \times B$ enthält Paare (a, b) mit $a \in A$ und $b \in B$. Nun fasst man das Paar als endliche Folge auf, also als Abbildung

$$f : \{0, 1\} \to A \cup B, \quad f(0) := a, \quad f(1) := b.$$

Allgemein definiert man insofern

$$\prod_{x \in X} A_x := \{f : X \to \bigcup_{x \in X} A_x \mid \forall x \in X : f(x) \in A_x\}.$$

Pi-Typen

Erklärung anhand der Analogie in der Mengenlehre. Ein Produkt $A \times B$ enthält Paare (a, b) mit $a \in A$ und $b \in B$. Nun fasst man das Paar als endliche Folge auf, also als Abbildung

$$f : \{0, 1\} \to A \cup B, \quad f(0) := a, \quad f(1) := b.$$

Allgemein definiert man insofern

$$\prod_{x \in X} A_x := \{f : X \to \bigcup_{x \in X} A_x \mid \forall x \in X : f(x) \in A_x\}.$$

Bei Typen verhält es sich analog.

Eine Term f vom Typ $\prod_{x:X} A_x$ heißt *abhängige Funktion*. Ihr Wert $f(x)$ ist vom Typ A_x. Der Typ des Wertes hängt also vom Argument der Funktion ab.

Pi-Typen

Erklärung anhand der Analogie in der Mengenlehre. Ein Produkt $A \times B$ enthält Paare (a, b) mit $a \in A$ und $b \in B$. Nun fasst man das Paar als endliche Folge auf, also als Abbildung

$$f : \{0, 1\} \to A \cup B, \quad f(0) := a, \quad f(1) := b.$$

Allgemein definiert man insofern

$$\prod_{x \in X} A_x := \{f : X \to \bigcup_{x \in X} A_x \mid \forall x \in X : f(x) \in A_x\}.$$

Bei Typen verhält es sich analog.

Eine Term f vom Typ $\prod_{x:X} A_x$ heißt *abhängige Funktion*. Ihr Wert $f(x)$ ist vom Typ A_x. Der Typ des Wertes hängt also vom Argument der Funktion ab.

So wie das zweistellige Produkt die Konjunktion kodiert, kodiert das allgemeine Produkt die Allquantifizierung. Wurde die Funktion f konstruiert, erbringt dies für jedes x einen Zeugen bzw. Beweis $f(x)$, dass der Typ A_x bewohnt ist.

Sigma-Typen

Betrachten wir zunächst die disjunkte Vereinigung von Mengen. Für $t \in A + B$ ist entweder $t = (\mathrm{L}, a)$ mit $a \in A$ oder $t = (\mathrm{R}, b)$ mit $b \in B$. Möchte man den Begriff nun auf beliebig viele Summanden ausdehnen, findet sich

$$\sum_{x \in X} A_x := \bigcup_{x \in X} (\{x\} \times A_x).$$

Sigma-Typen

Betrachten wir zunächst die disjunkte Vereinigung von Mengen. Für $t \in A + B$ ist entweder $t = (\mathrm{L}, a)$ mit $a \in A$ oder $t = (\mathrm{R}, b)$ mit $b \in B$. Möchte man den Begriff nun auf beliebig viele Summanden ausdehnen, findet sich

$$\sum_{x \in X} A_x := \bigcup_{x \in X} (\{x\} \times A_x).$$

Analog verhält es sich bei der Summe von Typen. Ein Term t vom Typ $\sum_{x:X} A_x$ heißt *abhängiges Paar*. Für $t = (x, a)$ ist a vom Typ A_x. Der Typ der rechten Komponente des Paars hängt also von der linken Komponente ab.

Sigma-Typen

Betrachten wir zunächst die disjunkte Vereinigung von Mengen. Für $t \in A + B$ ist entweder $t = (\mathrm{L}, a)$ mit $a \in A$ oder $t = (\mathrm{R}, b)$ mit $b \in B$. Möchte man den Begriff nun auf beliebig viele Summanden ausdehnen, findet sich

$$\sum_{x \in X} A_x := \bigcup_{x \in X} (\{x\} \times A_x).$$

Analog verhält es sich bei der Summe von Typen. Ein Term t vom Typ $\sum_{x:X} A_x$ heißt *abhängiges Paar*. Für $t = (x, a)$ ist a vom Typ A_x. Der Typ der rechten Komponente des Paars hängt also von der linken Komponente ab.

So wie die zweistellige Summe die Disjunktion kodiert, kodiert die allgemeine Summe die Existenzquantifizierung. Ein konstruiertes Paar (x, a) bezeugt, dass die Summe bewohnt ist. Man bezeichnet auch x selbst als den Zeugen, wobei zugehörige Beweis a belegt, dass der Summand A_x bewohnt ist. Der Zeuge besteht also eigentlich nicht nur in x, sondern in der gesamten Information (x, a).

Wir schreiben ab jetzt $\forall x.\, P(x)$ und $\exists x.\, P(x)$ anstelle von $\forall x : P(x)$ und $\exists x : P(x)$, um Verwirrung zu vermeiden.

Der Aussage

$$A \wedge (\forall x.\, P(x)) \to \forall x.\, A \wedge P(x)$$

entspricht

$$A \times (\prod_x P(x)) \to \prod_x (A \times P(x)).$$

Es ist also ein Paar (a, f) mit $a : A$ und $f : \prod_x P(x)$ gegeben. Zu konstruieren ist eine abhängige Funktion, die jedem x ein Paar (a, y) mit $a : A$ und $y : P(x)$ zuordnet.

Wir schreiben ab jetzt $\forall x.\, P(x)$ und $\exists x.\, P(x)$ anstelle von $\forall x : P(x)$ und $\exists x : P(x)$, um Verwirrung zu vermeiden.

Der Aussage

$$A \wedge (\forall x.\, P(x)) \to \forall x.\, A \wedge P(x)$$

entspricht

$$A \times (\prod_x P(x)) \to \prod_x (A \times P(x)).$$

Es ist also ein Paar (a, f) mit $a : A$ und $f : \prod_x P(x)$ gegeben. Zu konstruieren ist eine abhängige Funktion, die jedem x ein Paar (a, y) mit $a : A$ und $y : P(x)$ zuordnet.

Schlicht $y := f(x)$ setzen. Die gesuchte Funktion ist also $x \mapsto (a, f(x))$.

Gallina

```
Definition proof9 (A: Type) (X: Type) (P: X -> Type):
  A*(forall x, P x) -> forall x, A*(P x)
  := fun t => match t with (a, f) => fun x => (a, f x) end.
```

Taktiken

Als Alternative zur direkten Formulierung von Termen bietet Coq außerdem noch Taktiken an, darunter versteht man Prozeduren zur automatischen Erstellung von Teilbeweisen.

Als Alternative zur direkten Formulierung von Termen bietet Coq außerdem noch Taktiken an, darunter versteht man Prozeduren zur automatischen Erstellung von Teilbeweisen.

Die Erstellung des Beweisbaums verläuft hierbei rückwärts. Man arbeitet sich also von der Wurzel aus bis zu den Blättern durch. Ein noch unbestätigter Knoten stellt in diesem Zusammenhang ein zu erreichendes *Ziel* dar, dessen Kindknoten die *Unterziele*.

Es stehen Basistaktiken zur Verfügung, die im Prinzip die Schlussregeln des natürlichen Schließens widerspiegeln. Dies darf allerdings nicht immer allzu genau verstanden werden. Man kann ja zu einigen Regeln alternative Formulierungen in Form von zulässigen Schlussregeln herleiten. Im gleichen Sinne steht ein Sammelsurium von Taktiken zur Verfügung.

Taktik zum Axiom

Regel	**Taktik**
$\dfrac{}{\Gamma, h : A, \Gamma' \vdash h : A}$	`exact h`

Bemerkung. Es gestattet `exact` nicht nur Variablen für die Blätter des Baums. Die Erstellung beliebiger korrekt konstruierter Terme ist ebenfalls möglich.

Taktiken zu den Einführungsregeln

Regel	**Taktik**
$\dfrac{\Gamma, h : A \vdash ? : B}{\Gamma \vdash ? : A \to B}$	`intro h`
$\dfrac{\Gamma \vdash ? : A \qquad \Gamma \vdash ? : B}{\Gamma \vdash ? : A \wedge B}$	`split`
$\dfrac{\Gamma \vdash ? : A}{\Gamma \vdash ? : A \vee B}$	`left`
$\dfrac{\Gamma \vdash ? : B}{\Gamma \vdash ? : A \vee B}$	`right`

Taktiken zu den Beseitigungsregeln

Regel	**Taktik**
$\dfrac{\Gamma \vdash h : A \to B \qquad \Gamma \vdash ? : A}{\Gamma \vdash ? : B}$	`apply h`
$\dfrac{\Gamma, a : A, b : B, \Gamma' \vdash ? : C}{\Gamma, h : A \wedge B, \Gamma' \vdash ? : C}$	`destruct h as (a, b)`

Um $A \wedge B \to A$ zu zeigen, können wir also wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{}{a : A, b : B \vdash a : A}\ \texttt{exact a}}{h : A \wedge B \vdash a : A}\ \texttt{destruct h as (a, b)}}{\vdash (h \mapsto a) : A \wedge B \to A}\ \texttt{intro h}$$

Um $A \wedge B \to A$ zu zeigen, können wir also wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{}{a : A, b : B \vdash a : A}\ \texttt{exact a}}{h : A \wedge B \vdash a : A}\ \texttt{destruct h as (a, b)}}{\vdash (h \mapsto a) : A \wedge B \to A}\ \texttt{intro h}$$

Die Implementierung:

Gallina

```
Theorem projl (A B: Prop): A /\ B -> A.
Proof.
  intro h.
  destruct h as (a, b).
  exact a.
Qed.
```

Um $(A \to B) \wedge A \to B$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{\overline{f : A \to B \vdash f : A \to B} \quad \overline{a : A \vdash a : A}}{f : A \to B, a : A \vdash f(a) : B}}{h : (A \to B) \wedge A \vdash f(a) : B}}{\vdash (h \mapsto f(a)) : (A \to B) \wedge A \to B}$$

```
exact a
apply f
destruct h as (f, a)
intro h
```

Um $(A \to B) \wedge A \to B$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{\overline{f : A \to B \vdash f : A \to B} \quad \overline{a : A \vdash a : A}}{f : A \to B, a : A \vdash f(a) : B}}{h : (A \to B) \wedge A \vdash f(a) : B}}{\vdash (h \mapsto f(a)) : (A \to B) \wedge A \to B}$$

exact a
apply f
destruct h as (f, a)
intro h

Die Implementierung:

Gallina

```
Theorem mp (A B: Prop): (A -> B) /\ A -> B.
Proof.
  intro h.
  destruct h as (f, a).
  apply f.
  exact a.
Qed.
```

Um $A \wedge B \to B \wedge A$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{}{b : B \vdash b : B}\ \texttt{exact b} \quad \dfrac{}{a : A \vdash a : A}\ \texttt{exact a}}{a : A, b : B \vdash \mathrm{conj}(b)(a) : B \wedge A}\ \texttt{split}}{h : A \wedge B \vdash \mathrm{conj}(b)(a) : B \wedge A}\ \texttt{destruct h as (a, b)}$$
$$\dfrac{}{\vdash (h \mapsto \mathrm{conj}(b)(a)) : A \wedge B \to B \wedge A}\ \texttt{intro h}$$

Um $A \wedge B \to B \wedge A$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{\dfrac{}{b : B \vdash b : B}\ \texttt{exact b} \quad \dfrac{}{a : A \vdash a : A}\ \texttt{exact a}}{a : A, b : B \vdash \mathrm{conj}(b)(a) : B \wedge A}\ \texttt{split}}{h : A \wedge B \vdash \mathrm{conj}(b)(a) : B \wedge A}\ \texttt{destruct h as (a, b)}$$
$$\vdash (h \mapsto \mathrm{conj}(b)(a)) : A \wedge B \to B \wedge A\ \texttt{intro h}$$

Die Implementierung:

Gallina

```
Theorem conjunction_commutes (A B: Prop): A /\ B -> B /\ A.
Proof.
  intro h.
  destruct h as (a, b).
  split.
  - exact b.
  - exact a.
Qed.
```

Um $A \vee B \to B \vee A$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{}{h : A \vee B \vdash h : A \vee B} \quad \dfrac{\dfrac{}{a : A \vdash a : A}}{a : A \vdash r(a) : B \vee A} \quad \dfrac{\dfrac{}{b : B \vdash b : B}\ \texttt{exact b}}{b : B \vdash l(b) : B \vee A}\ \texttt{left}}{\dfrac{h : A \vee B \vdash s : B \vee A}{\vdash (h \mapsto s) : A \vee B \to B \vee A}\ \texttt{intro h}}\ \texttt{destruct h as [a | b]}$$

mit $l :=$ or_introl, $r :=$ or_intror und

$$s := \textbf{match}\ h \begin{cases} l(a) \mapsto r(a), \\ r(b) \mapsto l(b). \end{cases}$$

Um $A \vee B \to B \vee A$ zu zeigen, können wir wie folgt vorgehen:

$$\dfrac{\dfrac{}{h : A \vee B \vdash h : A \vee B} \quad \dfrac{\dfrac{}{a : A \vdash a : A}}{a : A \vdash r(a) : B \vee A} \quad \dfrac{\dfrac{}{b : B \vdash b : B}\ \texttt{exact b}}{b : B \vdash l(b) : B \vee A}\ \texttt{left}}{h : A \vee B \vdash s : B \vee A}\ \texttt{destruct h as [a | b]}$$
$$\vdash (h \mapsto s) : A \vee B \to B \vee A \quad \texttt{intro h}$$

mit $l :=$ or_introl, $r :=$ or_intror und

$$s := \textbf{match}\ h \begin{cases} l(a) \mapsto r(a), \\ r(b) \mapsto l(b). \end{cases}$$

Die Implementierung:

Gallina

```
Theorem disjunction_commutes (A B: Prop): A \/ B -> B \/ A.
Proof.
  intro h.
  destruct h as [a | b].
  - right. exact a.
  - left. exact b.
Qed.
```

Nun wird man irgendwann höhere kognitive Sprünge vollführen und mühseligen Kleinkram auslassen wollen. Hierfür stehen höhere Taktiken zur Verfügung, die einfache Beweise gänzlich automatisch konstruieren. Ein Beispiel hierfür ist `tauto`.

Nun wird man irgendwann höhere kognitive Sprünge vollführen und mühseligen Kleinkram auslassen wollen. Hierfür stehen höhere Taktiken zur Verfügung, die einfache Beweise gänzlich automatisch konstruieren. Ein Beispiel hierfür ist `tauto`.

Gallina

```
Theorem projl (A B: Prop): A /\ B -> A.
Proof.
  tauto.
Qed.
```

Nun wird man irgendwann höhere kognitive Sprünge vollführen und mühseligen Kleinkram auslassen wollen. Hierfür stehen höhere Taktiken zur Verfügung, die einfache Beweise gänzlich automatisch konstruieren. Ein Beispiel hierfür ist `tauto`.

Gallina

```
Theorem projl (A B: Prop): A /\ B -> A.
Proof.
  tauto.
Qed.
```

Gut, das wäre im ersten Semester nicht gestattet. Nun kann man trotzdem Schummeln, indem man sich den erzeugten Beweis einfach anschaut:

Gallina

```
Print projl.

(* Ausgabe: *)
projl =
fun (A B: Prop) (H: A /\ B) => and_ind (fun (H0: A) (_: B) => H0) H
     : forall A B: Prop, A /\ B -> A
```

Für $(A \wedge \forall x.\, P(x)) \rightarrow \forall x.\, A \wedge P(x)$ findet sich:

Für $(A \wedge \forall x.\, P(x)) \to \forall x.\, A \wedge P(x)$ findet sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash a : A}\ \texttt{exact a} \quad \dfrac{}{f : \forall x.\, P(x), x : X \vdash f(x)}\ \texttt{apply f}}{a : A, f : \forall x.\, P(x), x : X \vdash y : A \wedge P(x)}\ \texttt{split}}{a : A, f : \forall x.\, P(x) \vdash (x \mapsto y) : \forall x.\, A \wedge P(x)}\ \texttt{intro x}}{h : A \wedge \forall x.\, P(x) \vdash (x \mapsto y) : \forall x.\, A \wedge P(x)}\ \texttt{destruct h as (a, f)}$$

$$\vdash (h \mapsto x \mapsto y) : (A \wedge \forall x.\, P(x)) \to \forall x.\, A \wedge P(x)\ \texttt{intro h}$$

Hierbei gilt $y := \mathrm{conj}(a)(f(x))$. Die Implementierung:

Für $(A \wedge \forall x.\, P(x)) \to \forall x.\, A \wedge P(x)$ findet sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash a : A}\ \texttt{exact a} \quad \dfrac{}{f : \forall x.\, P(x), x : X \vdash f(x)}\ \texttt{apply f}}{a : A, f : \forall x.\, P(x), x : X \vdash y : A \wedge P(x)}\ \texttt{split}}{a : A, f : \forall x.\, P(x) \vdash (x \mapsto y) : \forall x.\, A \wedge P(x)}\ \texttt{intro x}}{h : A \wedge \forall x.\, P(x) \vdash (x \mapsto y) : \forall x.\, A \wedge P(x)}\ \texttt{destruct h as (a, f)}$$

$$\vdash (h \mapsto x \mapsto y) : (A \wedge \forall x.\, P(x)) \to \forall x.\, A \wedge P(x)\ \texttt{intro h}$$

Hierbei gilt $y := \mathrm{conj}(a)(f(x))$. Die Implementierung:

Gallina

```
Theorem const_factor (A: Prop) (X: Type) (P: X -> Prop):
  A /\ (forall x, P x) -> forall x, A /\ P x.
Proof.
  intro h.
  destruct h as (a, f).
  intro x.
  split.
  - exact a.
  - apply f.
Qed.
```

Für $A \wedge (\exists x.\, P(x)) \rightarrow \exists x.\, A \wedge P(x)$ findet sich:

Für $A \wedge (\exists x.\, P(x)) \to \exists x.\, A \wedge P(x)$ findet sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash a : A}\ \texttt{exact a} \quad \dfrac{}{x : X, p : P(x) \vdash p : P(x)}\ \texttt{exact p}}{a : A, x : X, p : P(x) \vdash \mathrm{conj}(a)(p) : A \wedge P(x)}\ \texttt{split}}{a : A, x : X, p : P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{exists x} \quad \dfrac{}{s : \exists x.\, P(x) \vdash s : \exists x.\, P(x)}}{a : A, s : \exists x.\, P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{destruct s as (x, p)}}{h : A \wedge \exists x.\, P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{destruct h as (a, s)}}{\vdash (h \mapsto s') : A \wedge (\exists x.\, P(x)) \to \exists x.\, A \wedge P(x)}\ \texttt{intro h}$$

Für $A \wedge (\exists x.\, P(x)) \to \exists x.\, A \wedge P(x)$ findet sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash a : A}\ \texttt{exact a} \quad \dfrac{}{x : X, p : P(x) \vdash p : P(x)}\ \texttt{exact p}}{a : A, x : X, p : P(x) \vdash \mathrm{conj}(a)(p) : A \wedge P(x)}\ \texttt{split}}{a : A, x : X, p : P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{exists x} \quad \dfrac{}{s : \exists x.\, P(x) \vdash s : \exists x.\, P(x)}}{a : A, s : \exists x.\, P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{destruct s as (x, p)}}{h : A \wedge \exists x.\, P(x) \vdash s' : \exists x.\, A \wedge P(x)}\ \texttt{destruct h as (a, s)}}{\vdash (h \mapsto s') : A \wedge (\exists x.\, P(x)) \to \exists x.\, A \wedge P(x)}\ \texttt{intro h}$$

Die Implementierung:

Gallina

```
Theorem distribute (A: Prop) (X: Type) (P: X -> Prop):
  A /\ (exists x, P x) -> exists x, A /\ P x.
Proof.
  intro h.
  destruct h as (a, s).
  destruct s as (x, p).
  exists x.
  split.
  - exact a.
  - exact p.
Qed.
```

Zum Abschluss Zahlentheorie

Gleichungen werden durch Typen ausgedrückt. Wir schreiben $p : (n = 2k)$ für das Urteil, dass p vom Typ $n = 2k$ ist. Es stellt also p einen Beweis für die Gleichheit $n = 2k$ dar.

Gleichungen werden durch Typen ausgedrückt. Wir schreiben $p : (n = 2k)$ für das Urteil, dass p vom Typ $n = 2k$ ist. Es stellt also p einen Beweis für die Gleichheit $n = 2k$ dar.

Nun darf die Ersetzungsregel zur Anwendung kommen. Man schreibt dies als

$$\frac{\Gamma \vdash p : n = 2k}{\Gamma \vdash y : f(n) = f(2k)} \quad \text{mit } y := \mathsf{f\_equal}(f)(p).$$

Gleichungen werden durch Typen ausgedrückt. Wir schreiben $p : (n = 2k)$ für das Urteil, dass p vom Typ $n = 2k$ ist. Es stellt also p einen Beweis für die Gleichheit $n = 2k$ dar.

Nun darf die Ersetzungsregel zur Anwendung kommen. Man schreibt dies als

$$\frac{\Gamma \vdash p : n = 2k}{\Gamma \vdash y : f(n) = f(2k)} \quad \text{mit } y := \mathsf{f\_equal}(f)(p).$$

Hiermit findet sich:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{h : (\exists k.\, n = 2k) \vdash h : (\exists k.\, n = 2k)} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash p : n = 2k}}{\Gamma \vdash y : n^2 = (2k)n}}{\Gamma \vdash y' : n^2 = 2(kn)}}{\Gamma \vdash y'' : \exists l.\, n^2 = 2l}\ \texttt{apply (f\_equal (}x \mapsto xn\texttt{) }p\texttt{)}}{h : \exists k.\, n = 2k \vdash y'' : \exists l.\, n^2 = 2l}\ \texttt{rewrite Nat.mul\_assoc}}{(h \mapsto y'') : (\exists k.\, n = 2k) \to (\exists l.\, n^2 = 2l)}\ \texttt{exists (}kn\texttt{)}\ \texttt{destruct h as (k, p)}\ \texttt{intro h}$$

mit $\Gamma := [k : \mathsf{nat}, p : n = 2k]$.

Die Implementierung:

Gallina

```
Require Import Coq.Arith.PeanoNat.

Theorem even_square (n: nat):
  (exists k, n = 2*k) -> (exists l, n*n = 2*l).
Proof.
  intro h.
  destruct h as (k, p).
  exists (k*n).
  rewrite Nat.mul_assoc.
  apply (f_equal (fun x => x*n) p).
Qed.
```

Möchte man den Beweisterm von Hand schreiben, erhält man zunächst separat $n^2 = (2k)n$ per Ersetzungsregel und $2(kn) = (2k)n$ per Assoziativgesetz. Um aus den beiden Gleichungen nun $n^2 = 2(kn)$ zu gewinnen, muss noch explizit das Transitivgesetz angewendet werden, wofür die Funktion `eq_trans_r` zur Verfügung steht.

Möchte man den Beweisterm von Hand schreiben, erhält man zunächst separat $n^2 = (2k)n$ per Ersetzungsregel und $2(kn) = (2k)n$ per Assoziativgesetz. Um aus den beiden Gleichungen nun $n^2 = 2(kn)$ zu gewinnen, muss noch explizit das Transitivgesetz angewendet werden, wofür die Funktion `eq_trans_r` zur Verfügung steht.

Gallina

```
Require Import Coq.Arith.PeanoNat.

Theorem even_square (n: nat):
  (exists k, n = 2*k) -> (exists l, n*n = 2*l).
Proof.
  exact (fun t =>
    match t with ex_intro _ k p => ex_intro _ (k*n)
      (let y := f_equal (fun x => x*n) p in
        eq_trans_r y (Nat.mul_assoc 2 k n))
    end).
Qed.
```

Alternativ kann man sich aber auch vom Fluss der verbleibenden Ziele leiten lassen und Taktiken zur Termersetzung nutzen:

Gallina

```
Require Import Coq.Arith.PeanoNat.

Theorem even_square (n: nat):
  (exists k, n = 2*k) -> (exists l, n*n = 2*l).
Proof.
  intro h.
  destruct h as (k, p).
  exists (k*n).
  rewrite Nat.mul_assoc.
  replace (2*k) with n.
  reflexivity.
Qed.
```

Literatur

- Philip Wadler: *Propositions as Types*. In: *Communications of the ACM*. Band 58, Nr. 12, 2015. S. 75–84. doi:10.1145/2699407. —Der klassische Einführungsartikel in die Thematik.
- Samuel Mimram: *Program = Proof*. Link (Open Access).
- Christine Paulin-Mohring: *Introduction to the Coq proof-assistant for practical software verification*. Laboratoire de recherche en informatique, Université Paris-Saclay, 2011. Link (Open Access).
- Benjamin C. Pierce u. a.: *Software Foundations. Volume 1: Logical Foundations.* University of Pennsylvania, 2022. Link (Open Access).

Ende.

November 2022