Teil IV

Quantoren in Isabelle/HOL

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: `∃x. P x`

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: `∀x. P x`

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

`∀x. P x ⟹ Q x x` in Konklusion nicht gebunden durch Allquantor

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: `∃x. P x`

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: `∀x. P x`

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

`∀x. P x ⟹ Q x x` in Konklusion nicht gebunden durch Allquantor

`P y ⟹ ∃y. P y y` in Prämisse nicht gebunden durch Existenzquantor

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: `∃x. P x`

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: `∀x. P x`

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

`∀x. P x ⟹ Q x` x in Konklusion nicht gebunden durch Allquantor

`P y ⟹ ∃y. P y` y in Prämisse nicht gebunden durch Existenzquantor

`⟦∀x. P x; ∃x. Q x⟧ ⟹ R`
Zwei verschiedene x in den Annahmen
gleichbedeutend mit `⟦∀y. P y; ∃z. Q z⟧ ⟹ R`
(*gebundene Namen sind Schall und Rauch*)

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: `∃x. P x`

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: `∀x. P x`

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

`∀x. P x ⟹ Q x` x in Konklusion nicht gebunden durch Allquantor

`P y ⟹ ∃y. P y` y in Prämisse nicht gebunden durch Existenzquantor

`⟦∀x. P x; ∃x. Q x⟧ ⟹ R`
Zwei verschiedene x in den Annahmen
gleichbedeutend mit `⟦∀y. P y; ∃z. Q z⟧ ⟹ R`
(*gebundene Namen sind Schall und Rauch*)

`∀x. P x ⟶ Q x` *gleiches* x für P und Q

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ [\![\ldots]\!] \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ [\![\forall y.\ P\ y \longrightarrow Q\ z\ y;\ Q\ x\ y]\!] \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ [\![\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1;\ Q\ x\ y]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ [\![\ldots]\!] \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ [\![\forall y.\ P\ y \longrightarrow Q\ z\ y;\ Q\ x\ y]\!] \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ [\![\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1;\ Q\ x\ y]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Auch $\Longrightarrow$ ist Teil der Meta-Logik, entspricht Meta-Implikation
Trennt Annahmen und Konklusion

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ [\![\ldots]\!] \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ [\![\forall y.\ P\ y \longrightarrow Q\ z\ y;\ Q\ x\ y]\!] \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ [\![\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1;\ Q\ x\ y]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Auch $\Longrightarrow$ ist Teil der Meta-Logik, entspricht Meta-Implikation
Trennt Annahmen und Konklusion

$\forall$ und $\longrightarrow$ entsprechen nicht $\bigwedge$ und $\Longrightarrow$, die ersten beiden nur in HOL!

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- `allI: (⋀x. P x) ⟹ ∀x. P x`
  Eine Aussage gilt für beliebige `x` (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- `allI: (⋀x. P x) ⟹ ∀x. P x`
  Eine Aussage gilt für beliebige `x` (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- `allE: ⟦∀x. P x; P ?x ⟹ R⟧ ⟹ R`
  Eine Aussage gilt für alle `x`, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- `allI: (⋀x. P x) ⟹ ∀x. P x`
  Eine Aussage gilt für beliebige `x` (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- `allE: ⟦∀x. P x; P ?x ⟹ R⟧ ⟹ R`
  Eine Aussage gilt für alle `x`, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann
- `exI: P ?x ⟹ ∃x. P x`
  Eine Aussage gilt für einen Term `?x`, also gibt es ein `x`, wofür sie gilt

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- `allI: (⋀x. P x) ⟹ ∀x. P x`
  Eine Aussage gilt für beliebige `x` (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- `allE: ⟦∀x. P x; P ?x ⟹ R⟧ ⟹ R`
  Eine Aussage gilt für alle `x`, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann
- `exI: P ?x ⟹ ∃x. P x`
  Eine Aussage gilt für einen Term `?x`, also gibt es ein `x`, wofür sie gilt
- `exE: ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q`
  Eine Aussage gilt für ein `x`, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen beliebigen, nicht weiter bestimmten Term zeigen kann

Quantoren in Isar

Der Befehl **fix** korrespondiert mit $\bigwedge$.

```
have "∀x. P x"
proof(rule allI)
  fix x
  show "P x" ⟨Beweis⟩
qed

from `∀x. P x`
have "Q (f x)"
proof(rule allE)
  fix x
  assume "P (f x)"
  thus "Q (f x)" ⟨Beweis⟩
qed
```

```
have "∃x. P x"
proof(rule exI)
  show "P (f 0)" ⟨Beweis⟩
qed

from `∃x. P x`
have "R"
proof(rule exE)
  fix x
  assume "P x"
  show "R" ⟨Beweis⟩
qed
```

Letzteres geht auch ohne neuen Scope:

```
from `∃x. P x`
obtain x where "P x" by (rule exE)
```

Karlsruher Institut für Technologie

Teil V

Fallunterscheidung

Fallunterscheidung

In (klassischen) Beweisen Fallunterscheidung wichtiges Hilfsmittel

$$\frac{\dfrac{P}{R} \qquad \dfrac{\neg P}{R}}{R}$$

Fallunterscheidung

In (klassischen) Beweisen Fallunterscheidung wichtiges Hilfsmittel

$$\frac{\dfrac{P}{R} \qquad \dfrac{\neg P}{R}}{R}$$

In Isabelle: Mit der Regel `case_split:` $[\![P \Longrightarrow R;\ \neg P \Longrightarrow R]\!] \Longrightarrow R$

Beispiel:

```
have "BB ∨ ¬ BB"
proof(rule case_split)
  assume "BB"
  thus "BB ∨ ¬BB"..
next
  assume "¬ BB"
  thus "BB ∨ ¬BB"..
qed
```

Statt **proof** `(rule case_split)` geht auch **proof** `(cases "BB")`. Vorteil: Die Annahmen sind gleich die richtigen (sonst erfährt Isabelle erst beim ersten **show** was `P` sein sollte – das klappt ggf. nicht zuverlässig).

Fallunterscheidung: Beispiel

Dieses Lemma wäre ohne Fallunterscheidung so nicht (einfach) lösbar!

```
lemma "B ∧ C ⟶ (A ∧ B) ∨ (¬ A ∧ C)"
proof
  assume "B ∧ C" hence "B"..
  from `B ∧ C`   have "C"..

  show "(A ∧ B) ∨ (¬ A ∧ C)"
  proof(cases A)
    assume A
    from `A` `B` have "A ∧ B"..
    thus ?thesis..
  next
    assume "¬A"
    from `¬A` `C` have "¬A ∧ C"..
    thus ?thesis..
  qed
qed
```

Karlsruher Institut für Technologie

Teil VI

Definition

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

```
definition solution :: "nat"
 where "solution = 42"
```

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

```
definition solution :: "nat"
 where "solution = 42"
```

Erzeugt Regel: `solution_def: solution = 42`

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

```
definition solution :: "nat"
 where "solution = 42"
```

Erzeugt Regel: `solution_def: solution = 42`

So können auch Funktionen definiert werden:

Beispiel:

```
definition nand :: "bool ⇒ bool ⇒ bool"
 where "nand A B = (¬ (A ∧ B))"
```

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

```
definition solution :: "nat"
 where "solution = 42"
```

Erzeugt Regel: `solution_def: solution = 42`

So können auch Funktionen definiert werden:

Beispiel:

```
definition nand :: "bool ⇒ bool ⇒ bool"
 where "nand A B = (¬ (A ∧ B))"
```

Erzeugt Regel: `nand_def: nand A B = (¬ (A ∧ B))`

definition – Syntaxdefinition

`nand` ist binärer Operator

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

Syntaxdefinition (Infix-Notation)

Schreibe `(`**infixl** *`"Operatorsymbol" n`*`)` an die Deklarationszeile, wobei

- **infixl** für linksgebundenen Infixoperator steht,
  infixr für rechtsgebundene
- *`Operatorsymbol`* ein beliebig wählbares Symbol für den Operator ist,
- *n* eine Zahl ist, welche die Präzedenz dieses Operators angibt

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

Syntaxdefinition (Infix-Notation)

Schreibe *(***infixl** *"Operatorsymbol" n)* an die Deklarationszeile, wobei

- **infixl** für linksgebundenen Infixoperator steht,
  infixr für rechtsgebundene
- *Operatorsymbol* ein beliebig wählbares Symbol für den Operator ist,
- *n* eine Zahl ist, welche die Präzedenz dieses Operators angibt

Beispiel: Operator `nand`

definition *nand :: "bool ⇒ bool ⇒ bool"* ***(infixl*** *"⋈" 36)*
where *"A ⋈ B = (¬ (A ∧ B))"*

Jetzt: *A ⋈ B ⋈ C* gleichbedeutend mit *nand (nand A B) C*

Karlsruher Institut für Technologie

Teil VII

Gleichungen

Gleichungen

Das Arbeiten mit Gleichungen ist eine sehr wichtige Beweistechnik. Die Hauptregel dabei ist die Substitution:

$$\frac{s = t \qquad P[s/x]}{P[t/x]}$$

Diese Regel gibt es auch in Isabelle:

```
subst: s = t ⟹ P s ⟹ P t
```

Gleichungen

Das Arbeiten mit Gleichungen ist eine sehr wichtige Beweistechnik. Die Hauptregel dabei ist die Substitution:

$$\frac{s = t \qquad P[s/x]}{P[t/x]}$$

Diese Regel gibt es auch in Isabelle:

```
subst: s = t ⟹ P s ⟹ P t
```

Beispiel:

```
assume "correct(solution)"
with solution_def
have "correct(42)" by (rule subst)
```

Auch nützlich:

```
ssubst: s = t ⟹ P t ⟹ P s
arg_cong: x = y ⟹ f x = f y
```

Transitivität

Gleichheit ist transitiv, auch in Isabelle:

trans: $r = s \Longrightarrow s = t \Longrightarrow r = t$

Aber umständlich:

```
lemma "foo = qux"
proof(rule trans)
  show "foo = bar" ⟨Beweis 1⟩
next
  show "bar = qux"
  proof(rule trans)
    show "bar = baz" ⟨Beweis 2⟩
  next
    show "baz = qux" ⟨Beweis 3⟩
  qed
qed
```

Transitivität

Gleichheit ist transitiv, auch in Isabelle:

trans: $r = s \Longrightarrow s = t \Longrightarrow r = t$

Aber umständlich:

```
lemma "foo = qux"
proof(rule trans)
  show "foo = bar" ⟨Beweis 1⟩
next
  show "bar = qux"
  proof(rule trans)
    show "bar = baz" ⟨Beweis 2⟩
  next
    show "baz = qux" ⟨Beweis 3⟩
  qed
qed
```

Besser mit **also** und **finally**:

```
lemma "foo = qux"
proof-
  have "foo = bar" ⟨Beweis 1⟩
  also
  have "bar = baz" ⟨Beweis 2⟩
  also
  have "baz = qux" ⟨Beweis 3⟩
  finally show ?thesis.
qed
```

Gleichungsketten

Aber oft **proof** *(rule trans)* zu schreiben wäre sehr umständlich.
Statt dessen: Gleichungsketten!

- Die Befehle **also** und **finally** sollten jeweils einer Aussage (**assume** oder **have**) folgen, die eine Gleichung ist.
- Das abschließende **finally** kombiniert die Aussagen per Transitivität und stellt das Ergebnis (wie **from**) bereit.
- In Ausdrücken steht ... für die rechte Seite der letzten Aussage.
- Ist ein Lemma *foo* falsch herum, kann man *foo[symmetric]* verwenden.
- Die Abkürzung *?thesis* steht für die Konklusion des aktuell zu beweisende Lemmas (vor Anwendung von Regeln!).
  Typisches Muster:

```
proof -
  ⟨...⟩
  finally show ?thesis.
qed
```

Gleichungsketten (Beispiel)

```
lemma "(A ∧ (A ∨ B)) = A"
proof-
  from conj_disj_distribL
  have "(A ∧ (A ∨ B)) = ((A ∧ A) ∨ (A ∧ B))".
  also
  from conj_absorb
  have "((A ∧ A) ∨ (A ∧ B)) = (A ∨ (A ∧ B))" by (rule arg_cong)
  also
  have "(A ∨ (A ∧ B)) = A"
  proof
    ⟨a nested proof⟩
  qed
  finally
  show "(A ∧ (A ∨ B)) = A".
qed
```

Gleichungsketten (Beispiel mit Abkürzungen)

```
lemma "(A ∧ (A ∨ B)) = A"
proof-
  from conj_disj_distribL
  have "(A ∧ (A ∨ B)) = ((A ∧ A) ∨ (A ∧ B))".
  also
  from conj_absorb
  have "... = (A ∨ (A ∧ B))" by (rule arg_cong)
  also
  have "... = A"
  proof
    ⟨a nested proof⟩
  qed
  finally
  show ?thesis.
qed
```

Randbemerkung: moreover und ultimately

Die **also**..**finally**-Struktur hat einen kleinen Bruder: **moreover**..**ultimately**. Hier werden die Aussagen nicht per Transitivität verbunden, sondern einfach gesammelt.

Randbemerkung: moreover und ultimately

Die **also**..**finally**-Struktur hat einen kleinen Bruder: **moreover**..**ultimately**. Hier werden die Aussagen nicht per Transitivität verbunden, sondern einfach gesammelt.

Damit können Verschachtelungen vermieden werden, die Beweise natürlicher aufgebaut und Wiederholungen in **from**-Befehlen verringert werden:

have *"A ∧ B"*
proof
  show *A*
    ⟨*Beweis A*⟩
next
  show *B*
    ⟨*Beweis B*⟩
qed

have *A*
  ⟨*Beweis A*⟩
moreover
have *B*
  ⟨*Beweis B*⟩
ultimately
  have *"A ∧ B"*..