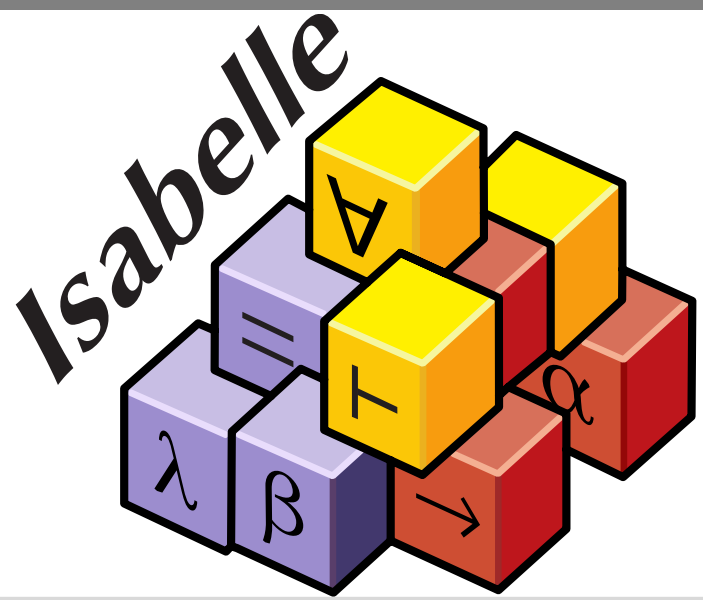
Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil XXIV

Attribute

Attribute

Allgemein: Attribute verändern Fakten.

Syntax:

theoremname[attribut1, attribut2, attribut mit optionen]

Kann überall verwendet werden, wo ein Fakt erwartet wird:

. . . **by** *(rule foo[bar])*

from *foo[bar]* **have**. . .

declare *neuer_name = foo[bar]*

note *neuer_name = foo[bar]*

. . .

Variablen in Regeln spezifizieren mittels `of`

Manchmal nötig, um Variablen vor Regelanwendung festzulegen (z.B. wenn Isabelle passende Terme nicht inferieren kann), dann:

- Attribut `of`, danach einer oder mehrere Terme
- müssen natürlich zu Typ der Variable passen
- Reihenfolge wie erstes Auftreten in Regel
- _ für Variablen, die man nicht instantiieren möchte

Beispiel:

`iffE:`	$[\![?P = ?Q;\ [\![?P \longrightarrow ?Q;\ ?Q \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of X]:`	$[\![X = ?Q;\ [\![X \longrightarrow ?Q;\ ?Q \longrightarrow X]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of _ Y]:`	$[\![?P = Y;\ [\![?P \longrightarrow Y;\ Y \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of X Y Z]:`	$[\![X = Y;\ [\![X \longrightarrow Y;\ Y \longrightarrow X]\!] \Longrightarrow Z]\!] \Longrightarrow Z$

Variablen in Regeln spezifizieren mittels `where`

Syntax:

Regel[where v=T]

Wobei

- *v* die zu spezifizierende Variable in der Regel *Regel* ist
- *T* der einzusetzende Term ist

Beispiel:

iffE: $[\![?P = ?Q; [\![?P \longrightarrow ?Q; ?Q \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$

iffE[where Q="X ∧ Y"]: $[\![?P = X \wedge Y; [\![?P \longrightarrow X \wedge Y; X \wedge Y \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$

Prämissen in Regeln spezifizieren mittels *OF*

Analog zu *of*: ganze Prämissen instantiieren

- Attribut *OF* gefolgt von Regelnamen.
- Konklusion der Regel und entspr. Prämisse müssen unifizieren.
- Entspr. Prämissen werden durch Prämissen der eingefügten Regel ersetzt.
- Mit _ werden Prämissen übersprungen.
- Gut bei Induktionshypothesen in Isar einsetzbar (*Foo.IH[OF bar]*).

Beispiel:

conjI:	$\llbracket ?P;\ ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$
ccontr:	$(\neg\ ?P \Longrightarrow False) \Longrightarrow ?P$
conjI[OF ccontr]:	$\llbracket \neg\ ?P \Longrightarrow False;\ ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$
conjI[OF ccontr,of X]:	$\llbracket \neg\ X \Longrightarrow False;\ ?Q \rrbracket \Longrightarrow X \wedge ?Q$

Konklusion umdrehen mit *symmetric*

Wenn die Konklusion einer Regel eine Gleichheit falsch herum hat, hilft *foo[symmetric]*:

Beispiel:

drop_all: $length\ ?xs \leq ?n \Longrightarrow drop\ ?n\ ?xs = []$

drop_all[symmetric]: $length\ ?xs \leq ?n \Longrightarrow [] = drop\ ?n\ ?xs$

Definitionen falten mit *folded* und *unfolded*

Man kann eine Gleichung (meist eine Definition) in einer Regel substituieren, je nach Richtung mit *foo[folded equality]* oder *foo[unfolded equality]*:

Beispiel:

solution_def:	*solution = 42*
foo:	*?P solution* $\Longrightarrow$ *?Q 42*
foo[unfolded solution_def]:	*?P 42* $\Longrightarrow$ *?Q 42*
foo[folded solution_def]:	*?P solution* $\Longrightarrow$ *?Q solution*

Regeln vereinfachen mit *simplified*

Das Attribut *[simplified]* lässt den Simplifier eine Regel vereinfachen. Das sollte man bei bewiesenen Lemmas eigentlich nicht brauchen (die kann man direkt „richtig“ formulieren), aber in Kombination mit *OF* oder *of* ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. *auto intro:* arbeiten kann.

(Sehr konstruiertes) Beispiel:

```
take_add:
  take (?i + ?j) ?xs = take ?i ?xs @ take ?j (drop ?i ?xs)
take_add[of 5 10]:
  take (5 + 10) ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
take_add[of 5 10, simplified]:
  take 15 ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
```

Regeln vereinfachen mit *simplified*

Das Attribut *[simplified]* lässt den Simplifier eine Regel vereinfachen. Das sollte man bei bewiesenen Lemmas eigentlich nicht brauchen (die kann man direkt „richtig“ formulieren), aber in Kombination mit *OF* oder *of* ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. *auto intro:* arbeiten kann.

(Sehr konstruiertes) Beispiel:

```
take_add:
  take (?i + ?j) ?xs = take ?i ?xs @ take ?j (drop ?i ?xs)
take_add[of 5 10]:
  take (5 + 10) ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
take_add[of 5 10, simplified]:
  take 15 ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
```

Das Attribut kann auch in der Form *[simplified regel1 regel2...]* verwendet werden. Dann verwendet der Simplifier nur die angegebenen Regeln.

Karlsruher Institut für Technologie

Teil XXV

Universelle Fallunterscheidung

Fallunterscheidungsregeln

Wir kennen bereits Fallunterscheidung

- klassisch (mit `case_split`),
- nach Datentypkonstruktor (Bsp. `list.exhaust`),
- als Regelinversion bei induktiven Prädikaten (Bsp. `palin.cases`),
- nach Pattern-Matching bei **fun**-Definitionen (Bsp. `BigNat.add'.cases`).

Alle diese Regeln folgen dem Muster:

$$(Fall1 \Longrightarrow P) \Longrightarrow$$
$$(Fall2 \Longrightarrow P) \Longrightarrow$$
$$(Fall3 \Longrightarrow P) \Longrightarrow$$
$$\ldots \Longrightarrow P$$

Im Allgemeinen kann jede Regel dieser Form als Fallunterscheidungsregel verwendet werden. Z.B.:

$$(even\ n \Longrightarrow P) \Longrightarrow (odd\ n \Longrightarrow P) \Longrightarrow P$$

Eigene Fallunterscheidungsregeln anwenden

```
lemma even_odd_cases:
  assumes "even n ⟹ P"
    and "odd n ⟹ P"
  shows "P"
```

Freie Variablen der Regel müssen instanziiert werden:

```
have "P (n::nat)"
proof (cases n rule: even_odd_cases)
  case 1
  ...
qed
```

Eigene Fallunterscheidungsregeln anwenden

```
lemma even_odd_cases [case_names even odd]:
  assumes "even n ⟹ P"
    and "odd n ⟹ P"
  shows "P"
```

Freie Variablen der Regel müssen instanziiert werden:

```
have "P (n::nat)"
proof (cases n rule: even_odd_cases)
  case even
  ...
qed
```

Eigene Fallunterscheidungsregeln anwenden

```
lemma even_odd_cases:
  obtains (even) "even n" | (odd) "odd n"
```

Freie Variablen der Regel müssen instanziiert werden:

```
have "P (n::nat)"
proof (cases n rule: even_odd_cases)
  case even
  ...
qed
```

lokale Fallunterscheidung

Fallunterscheidung ist auch lokal in einem Isar-Beweis mit dem Kommando **consider** möglich:

```
consider (even) "even n" | (odd) "odd n" by blast
then show ?thesis
proof cases
  case even
  ...
next
  case odd
  ...
qed
```

lokale Fallunterscheidung

Fallunterscheidung ist auch lokal in einem Isar-Beweis mit dem Kommando **consider** möglich:

```
consider (even) "even n" | (odd) "odd n" by blast
then show ?thesis
proof cases
  case even
  ...
next
  case odd
  ...
qed
```

Dabei auch „obtainen“ von Variablen möglich:

```
consider (zero) "n = 0" | (succ) x where "n = Suc x"
then have "even n"
proof cases
  case (succ x) ...
```

Karlsruher Institut für Technologie

Teil XXVI

Strukturierte Zwischenziele

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

```
have "P x"
  if "Q x"
  and "R x"
```

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable `that`, welche alle **if**-Annahmen enthält.

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

```
have "P x"
  if meine_annahme_1: "Q x"
  and meine_annahme_2: "R x"
```

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable `that`, welche alle **if**-Annahmen enthält.
- Benamte Annahmen

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

```
have "P x"
  if meine_annahme_1: "Q x"
  and meine_annahme_2: "R x"
  for x :: nat
```

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable `that`, welche alle **if**-Annahmen enthält.
- Benamte Annahmen
- (Meta-)Allquantifizierte Variablen

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

```
have "P x"
  if meine_annahme_1: "Q x"
  and meine_annahme_2: "R x"
  for x :: nat
```

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable `that`, welche alle **if**-Annahmen enthält.
- Benamte Annahmen
- (Meta-)Allquantifizierte Variablen

if und **for** auch mit **assume** möglich – nicht aber mit **assumes**. Also nur innerhalb eines Beweises.

Teil XXVII

Locales

Beweise modularisieren

$\Longrightarrow$ Verwende **Locales**

locale: Definiert neuen Beweiskontext

fixes: Legt Funktionssymbol fest (wird zu Parameter der Locale)

assumes: Macht Annahmen über die Locale-Parameter

context <locale> begin ... end: Öffnet Beweiskontext

Beispiel:

```
locale Magma =
 fixes M :: "'a set"
 fixes bop :: "'a ⇒ 'a ⇒ 'a"
 assumes closed: "a ∈ M ⟹ b ∈ M ⟹ bop a b ∈ M"

context Magma begin <Definitionen, Beweise, ...> end
```

Locales erweitern

Locales lassen sich mit “+” erweitern:

Beispiel:

```
locale Semigroup = Magma +
 assumes assoc: "..."
```

Auch “Verschmelzen” von Locales möglich:

Beispiel:

```
locale Ring = AbelianGroup "M" "add" "zero" + Magma "M" "mul"
 for M :: "'a set"
 and add :: "'a ⇒ 'a ⇒ 'a"
 and zero :: "'a"
 and mul :: "'a ⇒ 'a ⇒ 'a"
+ assumes assoc: "..."
```

Interpretationen

Instanziierung der Locales mit

interpretation: im Theoriekontext

interpret: in Beweiskontexten

Vorgehen:

- Angabe der konkreten Parameter
- Locale-Definition “auspacken” mit Taktik **unfold_locales**
- Beweis der Locale-Annahmen

Beispiel:

```
interpretation Mod3:
  Ring "{0::nat,1,2}" "λa b. a + b mod 3" "0" "λa b. a * b mod 3"
   by (unfold_locales) auto
```

Typklassen

Spezielle Locales mit genau einem Typparameter

Beispiel:

```
class Magma =
 fixes M :: "'a set"
 fixes bop :: "'a ⇒ 'a ⇒ 'a"
 assumes closed: "a ∈ M ⟹ b ∈ M ⟹ bop a b ∈ M"

context Magma begin <Definitionen, Beweise, ...> end
```

Instanziierung mittels **instantiation**

Vorteil: Alle Deklarationen innerhalb der Typklasse sind auch auf dem Top-Level verfügbar, die notwendige Instanz wird bei Anwendung durch *Typklasseninferenz* ermittelt.

Mehr dazu: Tutorial **classes** in der Isabelle-Dokumentation