

Teil XXI

Attribute

Attribute

Allgemein: Attribute verändern Theoreme.

Attribute

Allgemein: Attribute verändern Theoreme.

Syntax:

```
theoremname[attribut1, attribut2, attribut mit optionen]
```

Allgemein: Attribute verändern Theoreme.

Syntax:

```
theoremname[attribut1, attribut2, attribut mit optionen]
```

Kann überall verwendet werden, wo ein Theorem referenziert 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

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:

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

Syntax:

Regel [*where* $v=T$]

Wobei

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

Beispiel:

iffE: $\llbracket ?P = ?Q; \llbracket ?P \longrightarrow ?Q; ?Q \longrightarrow ?P \rrbracket \implies ?R \rrbracket$
 $\implies ?R$

iffE[*where* $Q="X \wedge Y"$]: $\llbracket ?P = X \wedge Y;$
 $\llbracket ?P \longrightarrow X \wedge Y; X \wedge Y \longrightarrow ?P \rrbracket \implies ?R \rrbracket$
 $\implies ?R$

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 Überspringen.
- Gut bei Induktionshypothesen in Isar einsetzbar ($Foo.IH[OF bar]$).

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 Überspringen.
- Gut bei Induktionshypothesen in Isar einsetzbar (*Foo.IH[of bar]*).

Beispiel:

```
conjI:                [[?P; ?Q]] ==> ?P & ?Q
ccontr:               (¬ ?P ==> False) ==> ?P
conjI[of ccontr]:    [[¬ ?P ==> False; ?Q]] ==> ?P & ?Q
conjI[of ccontr, of X]: [[¬ X ==> False; ?Q]] ==> X & ?Q
```

Konklusion umdrehen mit *symmetric*

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

Konklusion umdrehen mit *symmetric*

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

Beispiel:

<i>drop_all</i> :	$\text{length } ?xs \leq ?n \implies \text{drop } ?n ?xs = []$
<i>drop_all[symmetric]</i> :	$\text{length } ?xs \leq ?n \implies [] = \text{drop } ?n ?xs$

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]*:

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:

<i>solution_def</i>	<i>solution = 42</i>
<i>foo:</i>	<i>?P solution \implies ?Q 42</i>
<i>foo[unfolded solution_def]:</i>	<i>?P 42 \implies ?Q 42</i>
<i>foo[folded solution_def]:</i>	<i>?P solution \implies ?Q solution</i>

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 *DF* oder *of* ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. *auto intro*: arbeiten kann.

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 `DF` 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 `[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 `DF` 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.

Teil XXII

Typedef

Eigene Typen in HOL definieren

In HOL, und damit in Isabelle, können eigene Datentypen definiert werden. Dazu benötigt man

- eine Teilmenge eines existierenden Typs sowie
- ein Beweis, dass diese Teilmenge nicht leer ist.

(Leere Typen würden HOL inkonsistent machen, d.h. man könnte *False* beweisen.)

Syntax

```
typedef typename = "Menge" morphisms rep_fun abs_fun by proof
```

- *typename* ist der Name des neuen Typs. Hier dürfen auch Typvariablen verwendet werden ($(\text{'a}, \text{'b}) \textit{typename}$).
- *Menge* ist ein Ausdruck vom Typ *irgendwas set*.
- Morphismen konvertieren zwischen der Menge und dem neuen Typ:
 $\textit{rep_fun} :: \textit{typename} \Rightarrow \textit{irgendwas}$ und $\textit{abs_fun} :: \textit{irgendwas} \Rightarrow \textit{typename}$
- Default-Morphismennamen: *Rep_typname* und *Abs_typname*.
- Das Beweisziel ist $\exists x. x \in \textit{Menge}$.
- Erzeugt (u. a. und v. a.) diese Lemmas:
 $\textit{rep_fun}: \quad \textit{rep_fun} \textit{?x} \in \textit{Menge}$
 $\textit{rep_fun_inverse}: \quad \textit{abs_fun} (\textit{rep_fun} \textit{?x}) = \textit{?x}$
 $\textit{rep_abs_inverse}: \quad \textit{?y} \in \textit{Menge} \implies \textit{rep_fun} (\textit{abs_fun} \textit{?y}) = \textit{?y}$

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für **nicht-leere Listen** und beginnen mit der Typ-Definition:

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für **nicht-leere Listen** und beginnen mit der Typ-Definition:

```
typedef 'a ne = "{xs :: 'a list . xs ≠ []}"  
  by (rule exI[where x = "[undefined]"], simp)
```

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für **nicht-leere Listen** und beginnen mit der Typ-Definition:

```
typedef 'a ne = "{xs :: 'a list . xs ≠ []}"
  by (rule exI[where x = "[undefined]"], simp)
```

Weiter ein paar Funktionen auf nicht-leeren Listen:

```
definition singleton :: "'a ⇒ 'a ne"
  where "singleton x = Abs_ne [x]"
```

```
definition append :: "'a ne ⇒ 'a ne ⇒ 'a ne"
  where "append l1 l2 = Abs_ne (Rep_ne l1 @ Rep_ne l2)"
```

```
definition head :: "'a ne ⇒ 'a"
  where "head l = hd (Rep_ne l)"
```

```
definition tail :: "'a ne ⇒ 'a ne"
  where "tail l = Abs_ne (tl (Rep_ne l))"
```

Beispiel: Lemmas zu Nicht-Leeren Liste

Bei Append kommt der Head der Liste immer von der linken Liste (für allgemeine Listen nicht wahr!):

```
lemma "head (append l1 l2) = head l1"  
  unfolding head_def append_def  
  apply (subst Abs_ne_inverse)  
  using Rep_ne[of l1] apply simp  
  using Rep_ne[of l1] apply simp  
  done
```

Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"  
  unfolding head_def append_def singleton_def tail_def  
  apply (subst Abs_ne_inverse)  
  apply simp  
  apply (subst Abs_ne_inverse)  
  defer  
  using Rep_ne[of l]  
  apply simp  
  apply (rule Rep_ne_inverse)  
  apply simp  
oops
```


Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"  
  unfolding head_def append_def singleton_def tail_def  
  apply (subst Abs_ne_inverse)  
  apply simp  
  apply (subst Abs_ne_inverse)  
  defer  
  using Rep_ne[of l]  
  apply simp  
  apply (rule Rep_ne_inverse)  
  apply simp  
oops
```

Problem: Das Lemma ist „eigentlich“ richtig, aber `tail [a]` ist undefiniert, da keine nicht-leere Liste.

Beispiel: Richtige Lemmas zu Nicht-Leeren Liste

Tail muss eine „normale“ Liste zurückgeben:

```
definition tail' :: "'a ne  $\Rightarrow$  'a list"
  where "tail' l = tl (Rep_ne l)"
```

```
definition append' :: "'a ne  $\Rightarrow$  'a list  $\Rightarrow$  'a ne"
  where "append' l1 l2 = Abs_ne (Rep_ne l1 @ l2)"
```

```
lemma "append' (singleton (head l)) (tail' l) = l"
  unfolding head_def append'_def singleton_def tail'_def
  apply (subst Abs_ne_inverse, simp)
  using Rep_ne[of l, simplified]
  apply simp
  apply (rule Rep_ne_inverse)
  done
```

Das Beweisen mit den Abstraktions- und Repräsentationsfunktionen ist mühsam und unnatürlich: So wird die Erhaltung einer Invariante beim Verwenden der Funktion bewiesen, und nicht beim Definieren (siehe *tail*).

Die Isabelle-Pakete Lifting und Transfer erlauben es, Funktionen einmal bei der Definition als „korrekt“ zu beweisen und Lemmas mit einem Methodenaufruf in die Welt der zugrundeliegenden Repräsentation zu übertragen und dann dort zu beweisen.

Teil XXIII

Locales

⇒ Verwende **Locales**

locale: Definiert neuen Beweiskontext

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

assumes: Macht Annahmen über die Locale-Parameter

context ... begin: Öffnet Beweiskontext

end: Schließt 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 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: "..."
```

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
```