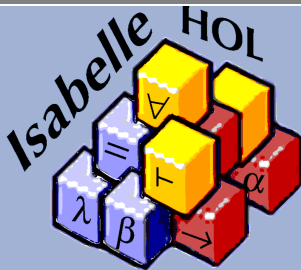


Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil XI

Eigene Lemmas als Regeln

Lemmas und Regeln

Man kann bewiesene Lemmas als Regeln verwenden!
(Wenig überraschend)

```
lemma mylemma: "even (42::nat)" by simp
```

```
lemma "∃ x. even (x::nat)"
```

```
proof
```

```
  show "even (42::nat)" by (rule mylemma)  
qed
```

Lemmas und Regeln: Annahmen

Man kann bewiesene Lemmas als Regeln verwenden!
Dabei sollten Annahmen mit der Meta-Implikation angegeben werden.

```
lemma mylemma2: "even n  $\implies$  even (3 * n)" by simp
```

```
lemma "even 126"
```

```
proof-
```

```
  have "even 42" by (rule mylemma)
```

```
  then have "even (3 * 42)" by (rule mylemma2)
```

```
  also have "3 * 42 = (126::nat)" by simp
```

```
  finally show ?thesis.
```

```
qed
```

Lemmas und Regeln: Annahmen vs. Isar

Man kann bewiesene Lemmas als Regeln verwenden!
Dabei sollten Annahmen mit der Meta-Implikation angegeben werden.
Das ist in Isar-beweisen nicht so schön (Annahmen müssen zweimal genannt werden):

```
lemma "even n  $\implies$  even (9 * n)"  
proof-  
  assume "even n"  
  have "even (3 * (3 * n))"  
  proof(rule mylemma2)  
    from (even n)  
    show "even (3 * n)" by (rule mylemma2)  
  qed  
  then show ?thesis by simp  
qed
```

In Isar sind mit den Schlüsselwörter **assumes** und **shows** die Annahmen direkt verfügbar, sie können benannt werden und Attribute wie `[simp]` gesetzt werden.

Außerdem bezeichnet `assms` immer alle Annahmen.

```
lemma times9:
  assumes n_is_even: "even n"
  shows "even (9 * n)"
proof-
  have "even (3 * (3 * n))"
  proof(rule mylemma2)
    from n_is_even — oder from ⟨even n⟩ oder from assms
    show "even (3 * n)" by (rule mylemma2)
  qed
then show ?thesis by simp
qed
```

Will man die Annahme(n) der ersten Beweismethode übergeben, so fügt man vor **proof** noch **using** `assms` ein. Wichtig bei Induktionsbeweisen!

So wie **assumes** dem \implies entspricht, entspricht **fixes** dem \wedge .
Damit lassen sich die freien Variablen des Lemmas besser betonen und ihr Typ kann festgelegt werden:

```
lemma times9:
  fixes n :: nat
  assumes n_is_even: "even n"
  shows "even (9 * n)"
proof-
  from n_is_even
  have "even (3 * n)" by (rule mylemma2)
  then have "even (3 * (3 * n))" by (rule mylemma2)
  also have "3 * (3 * n) = 9 * n" by simp
  finally show ?thesis.
qed
```

Teil XII

Allgemeine Rekursion

Allgemeine Rekursion

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Beispiel “mehrere Basisfälle”: Fibonacci-Zahlen

```
fun fib :: "nat  $\Rightarrow$  nat"  
  where "fib 0 = 1"  
        | "fib (Suc 0) = 1"  
        | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Beispiel “mehrere Basisfälle”: Fibonacci-Zahlen

```
fun fib :: "nat  $\Rightarrow$  nat"  
  where "fib 0 = 1"  
        | "fib (Suc 0) = 1"  
        | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Beispiel “Rekursion in mehreren Parametern”: Zippen von Listen

```
fun zip :: "'a list  $\Rightarrow$  'b list  $\Rightarrow$  ('a  $\times$  'b) list"  
  where "zip [] [] = []"  
        | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur “lineare Patterns” unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur “lineare Patterns” unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

Es ist erlaubt, dass sich Pattern überlappen. Es wird die erste passende Regel angewandt.

Damit sind default-Regeln möglich, die alle restlichen Fälle behandeln.

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur “lineare Patterns” unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

Es ist erlaubt, dass sich Pattern überlappen. Es wird die erste passende Regel angewandt.

Damit sind default-Regeln möglich, die alle restlichen Fälle behandeln.

Beispiel: Separatorzeichen zwischen je zwei Elemente einer Liste

```
fun sep :: "'a ⇒ 'a list ⇒ 'a list"  
where "sep a (x#y#zs) = x#a#sep a (y#zs)"  
      | "sep a xs      = xs"
```

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

Beispiel: *fib.simps:*

```
fib 0 = 1
```

```
fib (Suc 0) = 1
```

```
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```


In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

Beispiel: *fib.simps*:

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Beispiel: *sep.simps*

```
sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
sep ?a [] = []
sep ?a [?v] = [?v]
```

Beachte: Die Defaultregel (*sep a xs = xs*) generiert **zwei** Regeln, damit das Pattern-Matching vollständig ist.

Analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Diese kann man im Induktionsbeweis verwenden:

proof(*induction Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*.

Analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Diese kann man im Induktionsbeweis verwenden:

proof(*induction Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

$$\begin{aligned} & \llbracket \bigwedge a \ x \ y \ zs. \ ?P \ a \ (y \ \# \ zs) \implies \ ?P \ a \ (x \ \# \ y \ \# \ zs); \bigwedge a. \ ?P \ a \ []; \\ & \bigwedge a \ v. \ ?P \ a \ [v] \rrbracket \implies \ ?P \ ?a0.0 \ ?a1.0 \end{aligned}$$

Analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Diese kann man im Induktionsbeweis verwenden:

proof(*induction Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

```
[[ $\wedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \implies ?P\ a\ (x\ \#\ y\ \#\ zs); \wedge a.\ ?P\ a\ [];$   
 $\wedge a\ v.\ ?P\ a\ [v]] \implies ?P\ ?a0.0\ ?a1.0$ 
```

lemma "map f (sep x ys) = sep (f x) (map f ys)"

proof(*induction x ys rule:sep.induct*) generiert folgende 3 subgoals:

Analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Diese kann man im Induktionsbeweis verwenden:

proof(*induction Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

$$\begin{aligned} & \llbracket \bigwedge a \ x \ y \ zs. \ ?P \ a \ (y \ \# \ zs) \implies \ ?P \ a \ (x \ \# \ y \ \# \ zs); \ \bigwedge a. \ ?P \ a \ []; \\ & \bigwedge a \ v. \ ?P \ a \ [v] \rrbracket \implies \ ?P \ ?a0.0 \ ?a1.0 \end{aligned}$$

lemma "map f (sep x ys) = sep (f x) (map f ys)"

proof(*induction x ys rule:sep.induct*) generiert folgende 3 subgoals:

1. $\bigwedge a \ x \ y \ zs. \ \text{map } f \ (\text{sep } a \ (y \ \# \ zs)) = \text{sep } (f \ a) \ (\text{map } f \ (y \ \# \ zs)) \implies \text{map } f \ (\text{sep } a \ (x \ \# \ y \ \# \ zs)) = \text{sep } (f \ a) \ (\text{map } f \ (x \ \# \ y \ \# \ zs))$
2. $\bigwedge a. \ \text{map } f \ (\text{sep } a \ []) = \text{sep } (f \ a) \ (\text{map } f \ [])$
3. $\bigwedge a \ v. \ \text{map } f \ (\text{sep } a \ [v]) = \text{sep } (f \ a) \ (\text{map } f \ [v])$

fun ist sehr mächtig und in den meisten Fällen ausreichend, um rekursive Funktionen zu definieren.

Aber: auch mit **fun** kann es Probleme geben, z.B. bei wechselseitiger Rekursion oder falls **fun** die Termination nicht selbst beweisen kann.

Lösung: function

Braucht jedoch selbstgeschriebenen Vollständigkeits- und Terminationsbeweis...

Mehr dazu im **function**-Tutorial unter

<http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf>

Teil XIII

Wechselseitige Rekursion

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei

- mehreren Datentypen, die sich gegenseitig oder
- Datentypen, die eine Liste ihres eigenen Typs bei der Definition verwenden?

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei

- mehreren Datentypen, die sich gegenseitig oder
- Datentypen, die eine Liste ihres eigenen Typs bei der Definition verwenden?

Beispiel:

Bäume mit beliebigem Verzweigungsgrad, d.h. jeder Knoten verwaltet eine Liste von Nachfolgebäumen. Der Datentyp wird wie bisher definiert:

```
datatype 'a tree = Leaf 'a  
  | Node 'a "'a tree list"
```

Damit ist der Datentyp wechselseitig rekursiv definiert für sich und Liste seiner selbst.

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree ⇒ nat"
```

```
  where "height (Leaf l) = 1"  
        | "height (Node n ts) = ?"
```

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree  $\Rightarrow$  nat"
```

```
  where "height (Leaf l) = 1"
```

```
    | "height (Node n ts) = heights ts + 1"
```

Wir brauchen eine Definition der Höhe für Liste von Bäumen!

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree  $\Rightarrow$  nat"  
  and heights :: "'a tree list  $\Rightarrow$  nat"  
  
  where "height (Leaf l) = 1"  
        | "height (Node n ts) = heights ts + 1"  
  
        | "heights [] = 0"  
        | "heights (t#ts) = max (height t) (heights ts)"
```

Wir brauchen eine Definition der Höhe für Liste von Bäumen!

Dazu gleichzeitige Definition der Funktion *height* für *'a tree* und *heights* für *'a tree list*.

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

lemma

"height t > 0"

proof(*induction t*)

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

lemma

"height t > 0"

proof(*induction t*)

komisches subgoal: ?P2.0 []

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

lemma

"height t > 0"

proof(*induction t*)

komisches subgoal: ?P2.0 []

wir haben keine Aussage über die Höhe von Baumlisten!

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma

"height t > 0" and "heights ts ≥ 0"

proof(*induction t*)

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma

"height t > 0" and "heights ts ≥ 0"

proof(*induction t and ts*)

müssen beide Parameter getrennt durch **and** angeben

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma

"height t > 0" and "heights ts ≥ 0"

proof(*induction t and ts*)

Problem: wegen generischem Elementyp 'a werden *t* und *ts*
unterschiedliche Typen zugeordnet! *t::"'a tree"*, *ts::"'b tree list"*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
3. Versuch:

```
lemma fixes t::"'a tree" and ts::"'a tree list"  
shows "height t > 0" and "heights ts ≥ 0"  
proof(induction t and ts) qed auto
```

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

lemma "P t" "Q a t ts"

proof (induction t **and** ts)

q braucht a in der Induktionshypothese quantifiziert, also in einem **arbitrary**

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

lemma "P t" "Q a t ts"

proof (induction t **and** ts **arbitrary**: **and** a)

q braucht a in der Induktionshypothese quantifiziert, also in einem **arbitrary**

Auch hinter **arbitrary** werden die zu quantifizierenden Variablen für jedes Lemma mit **and** getrennt.

Teil XIV

Abkürzungen

Innerhalb eines Isar-Beweises können Terme **let**-gebunden werden.

Beispiel „Cantors Theorem“

```
theorem "∃ S. S ∉ range (f :: 'a ⇒ 'a set)"
```

```
proof
```

```
  let ?S = "{x. x ∉ f x}"
```

```
  show "?S ∉ range f"
```

```
  ...
```

Auch möglich: **def** führt eine neue (lokale) Konstante ein.

```
def S ≡ "{x. x ∉ f x}"
```

Die definierende Gleichung von s heißt s_def .

type_synonym

Auch Typen kann man neu benamen:

```
type_synonym 'a env = "'a  $\Rightarrow$  nat"
```

Ab jetzt sind 'a \Rightarrow nat und 'a env äquivalent/austauschbar.