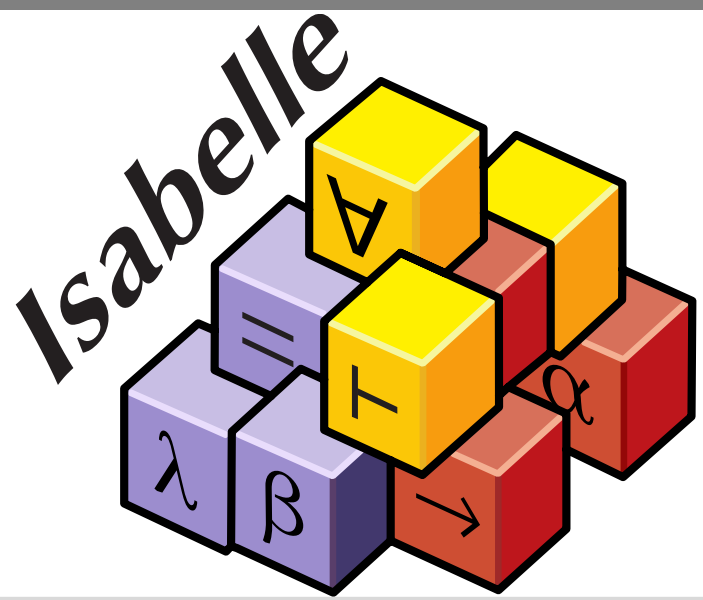
Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Karlsruher Institut für Technologie

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 ⟹ 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 ⟹ 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
```

Lemmas und Regeln: Strukturierte Lemma

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!

Lemmas und Regeln: Freie Variablen

So wie **assumes** dem $\Longrightarrow$ entspricht, entspricht **fixes** dem $\bigwedge$.
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
```

Karlsruher Institut für Technologie

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.

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.

Beispiel “mehrere Basisfälle”: Fibonacci-Zahlen

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

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.

Beispiel “mehrere Basisfälle”: Fibonacci-Zahlen

```
fun fib :: "nat ⇒ 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 ⇒ 'b list ⇒ ('a × 'b) list"
  where "zip [] [] = []"
  | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun

fun definiert Funktionen durch *Pattern Matching*.

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

fun

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 Patterns überlappen. Es wird die erste passende Regel angewandt.
Damit sind Defaultregeln möglich, die alle restlichen Fälle behandeln.

fun

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 Patterns überlappen. Es wird die erste passende Regel angewandt.
Damit sind Defaultregeln 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.

Simplifikationsregeln

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)
```

Simplifikationsregeln

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.

Induktionsregeln

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*.

Induktionsregeln

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*

```
⟦⋀a x y zs. ?P a (y # zs) ⟹ ?P a (x # y # zs); ⋀a. ?P a []; ⋀a v. ?P a [v]⟧ ⟹ ?P ?a0.0 ?a1.0
```

Induktionsregeln

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*

```
⟦⋀a x y zs. ?P a (y # zs) ⟹ ?P a (x # y # zs); ⋀a. ?P a []; ⋀a v. ?P a [v]⟧ ⟹ ?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:

Induktionsregeln

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*

```
⟦⋀a x y zs. ?P a (y # zs) ⟹ ?P a (x # y # zs); ⋀a. ?P a []; ⋀a v. ?P a [v]⟧ ⟹ ?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:

```
1. ⋀a x y zs. map f (sep a (y # zs)) = sep (f a) (map f (y # zs)) ⟹
     map f (sep a (x # y # zs)) = sep (f a) (map f (x # y # zs))
2. ⋀a. map f (sep a []) = sep (f a) (map f [])
3. ⋀a v. map f (sep a [v]) = sep (f a) (map f [v])
```

Ausblick

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

Wechselseitige Rekursion

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei mehreren Datentypen, die sich gegenseitig bei der Definition verwenden?

Wechselseitige Rekursion

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei mehreren Datentypen, die sich gegenseitig bei der Definition verwenden?

```
datatype
  'a aexp =    — arithmetische Ausdrücke
    Const nat
  | Var 'a
  | Sum "'a aexp" "'a aexp"
  | Diff "'a aexp" "'a aexp"
  | IF "'a bexp" "'a aexp" "'a aexp"
and
  'a bexp =    — boole'sche Ausdrücke
    Less "'a aexp" "'a aexp"
  | And "'a bexp" "'a bexp"
  | Neg "'a bexp"
```

Wechselseitige Rekursion

Definitions eines Größenmaßes für arithmetische Ausdrücke

Ansatz:

```
fun asize :: "'a aexp ⇒ nat"


  where "asize (Const n) = 1"
  | "asize (IF b a1 a2) = 1 + asize a1 + asize a2 + ?"
  | ...
```

Wechselseitige Rekursion

Definitions eines Größenmaßes für arithmetische Ausdrücke

Ansatz:

```
fun asize :: "'a aexp ⇒ nat"


  where "asize (Const n) = 1"
  | "asize (IF b a1 a2) = 1 + asize a1 + asize a2 + bsize b"
  | ...
```

Wir brauchen eine Definition für die Größe von boole'schen Ausdrücken!

Wechselseitige Rekursion

Definitions eines Größenmaßes für arithmetische Ausdrücke

Ansatz:

```
fun asize :: "'a aexp ⇒ nat"
  and bsize :: "'a bexp ⇒ nat"

  where "asize (Const n) = 1"
  | "asize (IF b a1 a2) = 1 + asize a1 + asize a2 + bsize b"
  | ...

  | "bsize (Neg b) = 0"
  | "bsize (Less a1 a2) = 1 + asize a1 + asize a2"
```

Wir brauchen eine Definition für die Größe von boole'schen Ausdrücken!

Dazu gleichzeitige Definition der Funktion `asize` für `'a aexp`
und `bsize` für `'a bexp`.

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
1. Versuch:

lemma *"asize a > 0"*
proof *(induction a)*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
1. Versuch:

lemma `"asize a > 0"`
proof `(induction a)`

Komisches subgoal: `?P2.0`

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
1. Versuch:

lemma *"asize a > 0"*
proof *(induction a)*

Komisches subgoal: *?P2.0*
Wir haben keine Aussage über die Größe von boole'schen Ausdrücken!

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
2. Versuch:

```
lemma "asize a > 0" and "bsize b > 0"
proof (induction a)
```

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
2. Versuch:

```
lemma "asize a > 0" and "bsize b > 0"
proof (induction a and b)
```

müssen beide Parameter getrennt durch **and** angeben

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
2. Versuch:

```
lemma "asize a > 0" and "bsize b > 0"
proof (induction a and b)
```

Problem: Wegen automatischer Verallgemeinerung des Lemmas haben `a` und `b` unterschiedliche Variablentypen! `a::"'a aexp", b::"'b bexp"`

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Größe jedes arithmetischen Ausdrucks mindestens 1 ist
3. Versuch:

```
lemma
  fixes a::"'a aexp" and b::"'a bexp"
  shows "asize a > 0" and "bsize b > 0"
proof (induction a and b) qed auto
```

Wechselseitige Rekursion und 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**

Wechselseitige Rekursion und 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**

Wechselseitige Rekursion und 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

abbreviation

Das Schlüsselwort **abbreviation** führt eine rein syntaktische Abkürzung ein, die bei der Eingabe verwendet werden kann und automatisch bei der Ausgabe berücksichtigt wird.

abbreviation *"sort ≡ sort_key (λx. x)"*

Auf Typebene funktioniert das mit **type_synonym**.

type_synonym *'a env = "'a ⇒ nat"*

let

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: **define** führt eine neue (lokale) Konstante ein.

```
  define S where "S = {x. x ∉ f x}"
```

Die definierende Gleichung von `S` heißt (standardmäßig) `S_def`.