

Rekursive Definition von Funktionen

über Nat oder anderen induktiv definierten Typen

Alternativen:

1. Einfache Gleichungen, wie oben

2. Bedingte Gleichungen:

$$y = nul \quad \Rightarrow \quad x + y = x$$

$$y = suc(y') \quad \Rightarrow \quad x + y = suc(x + y')$$

3. “Geschlossene” Definition, mit Hilfe bedingter Ausdrücke

– erfordert Selektorfunktion prd :

$$'+' := \lambda(x, y : Nat) : \mathbf{if} \ y = nul \ \mathbf{then} \ x \ \mathbf{else} \ suc(x + prd(y)) \ \mathbf{endif}$$

Die Alternativen sind semantisch im wesentlichen äquivalent; Unterschiede entstehen hauptsächlich durch die Art und Weise, wie Systeme die Definitionen behandeln.

Rekursive Definition von Funktionen (2)

In 1. und 2. werden die Fälle über eine Art “Mustererkennung” (*pattern matching*) ausgewählt, in 3. über die kontrollierende Bedingung.

In jedem Fall müssen *alle* Fälle der Datenstruktur abgedeckt sein.

In allen drei Formen ist die Parallelität zu beobachten zwischen

- dem induktivem Aufbau der Datenstruktur,
- der Struktur der rekursiven Funktionsdefinition, und
- dem Induktionsschema.

In PVS:

- Alle 3 Formen im Prinzip benutzt:
Form 1 als Termersetzungsregeln;
Form 2 ist ungünstig (keine “automatische” Unterstützung bedingter Gleichungen)
- PVS behandelt Definitionen dieser Formen als *Axiome*
 \rightsquigarrow Veränderung der Theorie

Rekursive Definition von Funktionen (3)

In PVS bevorzugte Syntax für Form 3:

```
add(x,y: Nat): RECURSIVE Nat =  
  IF y=nu1 THEN x ELSE suc(add(x,prd(y))) ENDIF  
  MEASURE y;
```

Man beachte:

- Expliziter Hinweis auf Rekursivität
- Erforderlich: Argument für Termierung und damit Wohldefiniertheit

In PVS (und in einigen anderen Systemen) muss für jede rekursive Funktionsdefinition nach 3. gezeigt werden, daß die Funktion wohldefiniert ist in dem Sinn, daß sie für alle Argumente einen definierten Wert liefert.

D.h. es wird gefordert nachzuweisen, daß jede Funktionsdefinition eine *konservative Erweiterung* der Theorie darstellt.

Dies kann durch Angabe einer *Maßfunktion* geschehen, mit deren Hilfe die entsprechenden Terminierungsbedingungen abgeleitet werden können.

Rekursive Definition von Funktionen (4)

Bezüglich Terminierung: Elemente von Nat sind endliche Terme (bzw. durch endliche Terme dargestellt); Argumente der rekursiven Aufrufe sind jeweils “kleiner”; Rekursion führt auf Basis-Fall zurück (entspricht dem Induktionsanfang).

Formal: das Argument für Terminierung wird auf eine *fundierte Relation* zurückgeführt (wird noch genauer behandelt; vergleiche auch Terminierung bei Termersetzung).

Lineare Listen

Abstrakter Datentyp `Liste` von linearen Listen über einem Typ A von “Atomen”:

PVS-Deklaration:

```
Liste [A: TYPE]: DATATYPE
  BEGIN
    leer: leer?
    lcons(kopf: A, rumpf: Liste): lcons?
  END Liste
```

erzeugt die Theorie:

```
Liste_adt[A: TYPE]: THEORY
  BEGIN
    Liste: TYPE
    leer?, lcons?: [Liste -> boolean]
    leer: (leer?)
    lcons: [[A, Liste] -> (lcons?)]
```

```
kopf: [(lcons?) -> A]
rumpf: [(lcons?) -> Liste]
```

```
ord(x: Liste): upto(1) =
  CASES x OF
    leer: 0,
    lcons(lcons1_var, lcons2_var): 1 ENDCASES
```

```
Liste_leer_extensionality: AXIOM
  (FORALL (leer?_var: (leer?), leer?_var2: (leer?)):
    leer?_var = leer?_var2);
```

```
Liste_lcons_extensionality: AXIOM
  (FORALL (lcons?_var: (lcons?),
    lcons?_var2: (lcons?)):
    kopf(lcons?_var) = kopf(lcons?_var2)
    AND rumpf(lcons?_var) = rumpf(lcons?_var2)
    IMPLIES lcons?_var = lcons?_var2);
```

```
Liste_lcons_eta: AXIOM
  (FORALL (lcons?_var: (lcons?))):
    lcons(kopf(lcons?_var), rumpf(lcons?_var))
      = lcons?_var);
```

```
Liste_kopf_lcons: AXIOM
  (FORALL (lcons1_var: A, lcons2_var: Liste):
    kopf(lcons(lcons1_var, lcons2_var))
      = lcons1_var);
```

```
Liste_rumpf_lcons: AXIOM
  (FORALL (lcons1_var: A, lcons2_var: Liste):
    rumpf(lcons(lcons1_var, lcons2_var))
      = lcons2_var);
```

```
Liste_inclusive: AXIOM
  (FORALL (Liste_var: Liste):
    leer?(Liste_var) OR lcons?(Liste_var));
```

```

Liste_induction: AXIOM
  (FORALL (p: [Liste -> boolean]):
    p(leer)
      AND
    (FORALL (lcons1_var: A, lcons2_var: Liste):
      p(lcons2_var) IMPLIES
        p(lcons(lcons1_var, lcons2_var)))
    IMPLIES
    (FORALL (Liste_var: Liste): p(Liste_var)));

```

```

subterm(x: Liste, y: Liste): boolean =
  x = y
  OR
  CASES y OF
    leer: FALSE,
    lcons(lcons1_var, lcons2_var):
      (LAMBDA (z: Liste): subterm(x, z))(lcons2_var)
  ENDCASES;

```

```
<< (x: Liste, y: Liste): boolean =
  CASES y OF
    leer: FALSE,
    lcons(lcons1_var, lcons2_var):
      (LAMBDA (z: Liste): x = z OR x << z)(lcons2_var)
  ENDCASES;
```

Induktion auf Listen

Listen-Konkatenation (“append”)

$$app(leer, y) = y$$

$$app(a \cdot x, y) = a \cdot app(x, y)$$

Definition hat im wesentlichen gleiche Struktur wie bei Addition (+);
Beweise für Assoziativität usw. analog zu denen für +.

Listenumkehrung (“reverse”):

$$rev(leer) = leer$$

$$rev(a \cdot x) = app(rev(x), a \cdot leer)$$

Behauptung: $rev(rev(x)) = x$

Beweis durch Induktion über x :

Ind.-Anfang:

$$rev(rev(leer)) = \dots = leer$$

Ind.Annahme: $rev(rev(x_1)) = x_1$

zu zeigen: $rev(rev(a \cdot x_1)) = a \cdot x_1$

Beispiel: Listenumkehrung (Forts.)

$$\text{rev}(\text{rev}(a \cdot x_1)) = \text{rev}(\text{app}(\text{rev}(x_1), a \cdot \text{leer})) = (*)$$

wünschenswert Funktion T , so daß

$$\begin{aligned} (*) &= T(\text{rev}(\text{rev}(x_1)), a \cdot \text{leer}) \\ &= T(x_1, a \cdot \text{leer}) = a \cdot x_1 \end{aligned}$$

$T(u, v) := \text{app}(v, \text{rev}(u))$ erfüllt die Bedingungen, sofern gezeigt wird, daß

$$\text{rev}(\text{app}(x, y)) = \text{app}(\text{rev}(y), \text{rev}(x))$$

Dies erfordert eine weitere Induktion.

Verallgemeinerung von Induktionsannahmen

Heuristik für Fälle, in denen eine Induktionsannahme zu schwach ist für den Beweis eines Induktionsschritts

Typisches (kanonisches?) Beispiel: Gleichheit von rekursiv und iterativ definierten Listenumkehrungen:

$$rev_2(x) = reva(x, leer)$$

$$reva(leer, y) = y$$

$$reva(a \cdot x, y) = reva(x, a \cdot y)$$

Behauptung: $rev(x) = rev_2(x)$

erster Versuch: Induktion über x

$$rev(leer) = leer = reva(leer, leer) = rev_2(leer)$$

Ind. Ann.: $rev(x_1) = reva(x_1, leer)$

zu zeigen: $rev(a \cdot x_1) = reva(a \cdot x_1, leer)$

li.S.: $rev(a \cdot x_1) = app(rev(x_1), a \cdot leer)$

$$= app(reva(x_1, leer), a \cdot leer) \quad \text{mit Ind. Ann.}$$

re.S.: $reva(a \cdot x_1, leer) = reva(x_1, a \cdot leer)$

\rightsquigarrow Terme lassen sich nicht zusammenbringen.

↷ **Verallgemeinerung** notwendig:

$$\forall x, y. \text{app}(\text{rev}(x), y) = \text{reva}(x, y)$$

Ind.Anf. trivial

$$\text{Ind. Ann.: } \forall y. \text{app}(\text{rev}(x_1), y) = \text{reva}(x_1, y)$$

zu zeigen: $\text{app}(\text{rev}(a \cdot x_1), y) = \text{reva}(a \cdot x_1, y)$

$$\text{app}(\text{rev}(a \cdot x_1), y) = \text{app}(\text{app}(\text{rev}(x_1), a \cdot \text{leer}), y)$$

$$= \text{app}(\text{rev}(x_1), a \cdot y) \quad \text{mit Assoz. etc.}$$

$$= \text{reva}(x_1, a \cdot y) \quad \text{mit Ind. Ann. instantiiert für } a \cdot y$$

$$= \text{reva}(a \cdot x_1, y)$$

Bemerkungen zum Beispiel:

Generalisiertes Argument ist “Akkumulator”

Verallgemeinerung im Beispiel entspricht dem Finden einer Schleifen-Invariante für ein imperatives Programm.

Andere typische Situationen für Verallgemeinerungen:

- Entkopplung von Vorkommen derselben Variablen in Induktions- und Nicht-Induktionspositionen

Beispiel (für $+$ mit 2. Argument als Rekursionsargument):

$$y + (y + y) = (y + y) + y \quad \rightarrow \quad y + (y + x) = (y + y) + x$$

Verallgemeinerungen sind gefährlich (gültige Formeln können zu nicht gültigen Formeln generalisiert werden), daher nur sehr kontrolliert zu verwenden (d.h. im allgemeinen *nicht automatisch*).

Richtige Verallgemeinerung erfordert ein gutes Verständnis des zu beweisenden Problems.

Binäre Bäume

Abstrakter Datentyp BBaum für binäre Bäume mit Blättern, die Werte aus einem Typ A von “Atomen” enthalten:

Deklaration in PVS:

```
BBaum [A: TYPE]: DATATYPE
  BEGIN
    blatt(val: A): blatt?
    comp(left, right: BBaum): comp?
  END Liste
```

PVS generiert hierfür eine Theorie BBaum_adt analog zu Liste_adt.

Der Hauptunterschied ist, dass in BBaum die Rekursion in zwei Richtungen absteigt; entsprechend stehen z.B. in der Induktionsvoraussetzung des Induktionsaxioms 2 Annahmen.

Fundierte Relationen

Eine Relation \prec auf einer Menge S heißt *fundiert* (engl. *well-founded*), falls jede nichtleere Teilmenge M von S wenigstens ein bezgl. \prec minimales Element m enthält (d.h. es gibt kein $x \in M$ mit $x \prec m$).

Eine Menge S mit einer fundierten Relation \prec heißt *fundiert* (oder *wohlfundiert*) bezgl. \prec .

Satz: Es sind äquivalent:

- (i) \prec ist fundierte Relation auf S .
- (ii) Es gibt in S keine unendliche absteigende Kette

$$\dots \prec x_n \prec \dots \prec x_2 \prec x_1.$$

Beweis: (mit $x \succ y \Leftrightarrow y \prec x$)

(i) \Rightarrow (ii): Angenommen, es gibt eine unendliche Kette $x_1 \succ x_2 \succ \dots$. Sei $M = \{x_1, x_2, \dots\}$. Für jedes $x_i \in M$ gibt es ein kleineres Element (x_{i+1}), daher hat M kein minimales Element.

Fundierte Relationen (Forts.)

Beweis: (i) \Leftarrow (ii):

Angenommen, \prec ist auf S nicht fundiert. Dann gibt es wenigstens eine nichtleere Teilmenge M von S ohne minimales Element. Eine unendliche absteigende Kette kann wie folgt konstruiert werden:

(a) x_1 willkürlich aus M gewählt (existiert, da M nicht leer ist).

(b) Für x_i wähle ein x_{i+1} , so daß $x_i \succ x_{i+1}$ – muß jeweils existieren, da es kein minimales Element gibt. \square

Lemma: Eine fundierte Relation ist irreflexive und asymmetrisch.

Bemerkung: Eine fundierte Relation muß nicht transitiv sein, d.h. keine Ordnungsrelation sein.

Die meisten hier benutzten fundierten Relationen sind jedoch Ordnungsrelationen.

Beispiele für fundierte Relationen:

(1) Die leere Relation ist fundiert.

(2) Die übliche Ordnungsrelation $<$ auf natürlichen Zahlen Nat

(3) Die *lexikographische Ordnung* $<^2$ auf $Nat \times Nat$:

$$(m_1, m_2) <^2 (n_1, n_2) \text{ g.d.w.}$$

$$m_1 < n_1 \text{ oder } m_1 = n_1 \wedge m_2 < n_2$$

Allgemein läßt sich eine lexikographische Ordnung auf Tupeln angeben, wenn jeweils eine fundierte Ordnung für jede Komponenten-Sorte gegeben ist.

(4) Die Teillisten-Relation auf linearen Listen:

$$l_1 < l_2 \text{ falls } l_1 \text{ ein "Endstück" von } l_2 \text{ ist}$$

$$l_1 < l_2 \Leftrightarrow (\exists n : Nat. n > 0 \wedge l_1 = rumpfn(l_2))$$

(5) Ordnungsrelation auf Binärbäumen:

$$t_1 < t_2 \text{ g.d.w. } t_1 \text{ Teilbaum von } t_2$$

insbesondere gilt:

$$t_1 < comp(t_1, t_2) \quad t_2 < comp(t_1, t_2)$$

Beispiele für fundierte Relationen (Forts.)

(6) Allgemein sind Teilstruktur-Relationen auf induktiv definierten Datenstrukturen fundiert. In PVS wird automatisch eine solche Teilstruktur-Relation \ll für jeden abstrakten Datentyp deklariert.

(7) Die folgende Relation auf Nat ist fundiert:

$$n \prec m \text{ g.d.w. } m = suc(n)$$

Diese Relation ist nicht transitiv, aber fundiert!

Fundierte Relationen (2)

Nachweis der Fundiertheit mit Hilfe von Abbildungen:

Gegeben Mengen S und M mit einer fundierten Relation $<_M$ auf M , weiterhin eine Abbildung $f : S \rightarrow M$ [“Maßfunktionen”].

Eine Relation $<_S$ auf S wird definiert (induziert) durch

$$x <_S y \iff f(x) <_M f(y) \text{ für alle } x, y \in S$$

Satz: Die Relation $<_S$ ist fundiert.

Beispiele:

- Jede Abbildung in Nat mit Ordnungsrelation $<$
- Längenfunktion über linearen Listen
- Höhenfunktion über Bäumen
- Anzahl der Blätter von Bäumen

All diese Funktionen sind “Zählfunktionen”

Beispiel für eine Nicht-Zählfunktion: $flt : BB \rightarrow Liste$

Noethersche Induktion

Satz (Noethersche Induktion): Sei \prec eine fundierte Relation auf einer Menge S . Um eine Eigenschaft P für alle $x \in S$ nachzuweisen (d.h. $\forall x : S. P(x)$), genügt es zu zeigen:

$$\forall x : S. (\forall y : S. y \prec x \Rightarrow P(y)) \Rightarrow P(x)$$

Beweis: Angenommen, die Menge A der Elemente aus S , für die P nicht gilt, ist nicht-leer. Dann hat sie ein kleinstes Element, m . Nach Definition gilt $P(y)$ für alle $y \prec m$, nach Voraussetzung muß dann auch $P(m)$ gelten, im Widerspruch zur Annahme. Daher muß A leer sein, d.h. P gilt für alle $x \in S$. \square

Beispiele für Noethersche Induktion:

“vollständige Induktion” über natürlichen Zahlen:

Um $\forall x : \text{Nat}. P(x)$ zu beweisen, genügt es zu zeigen

$$\forall x : \text{Nat}. (\forall y : \text{Nat}. y < x \Rightarrow P(y)) \Rightarrow P(x)$$

Bemerkung: der Basis-Fall $P(0)$ ist “automatisch” abgedeckt.

Vollständige Induktion auf binären Bäumen (mit $<$ als Teilbaum-Relation):

$$\forall t : \text{BB}. P(t) \quad \text{falls} \quad \forall t : \text{BB}. (\forall tt : \text{BB}. tt < t \Rightarrow P(tt)) \Rightarrow P(t)$$

Bemerkungen:

Strukturelle Induktion ist ein Spezialfall der Noetherschen Induktion: die Teilstruktur-Relation (die sich für jede induktiv definierte Struktur einführen läßt) ist jeweils fundiert.

Fundierte (Ordnungs-)Relationen sind wichtig für

- den Nachweis der Terminierung für rekursiv definierte Funktionen (s. frühere Folien),
- als Grundlage für Induktionsbeweise.

Die Charakterisierung fundierter Relationen über minimale Elemente kann als Beweisprinzip statt eines Induktionsbeweises genutzt werden.

Terminierung

notwendig für den Nachweis, daß

- eine “richtige”, d.h. insbesondere *totale*, Funktion definiert wird (notwendig in einem Kontext wie PL1, in dem es nur totale Funktionen gibt, aber auch in PVS und ähnlichen Systemen);
- keine Inkonsistenz entsteht durch Hinzunahme der Gleichungen, die eine Funktion definieren.
~> rekursive Funktionsdefinitionen sollen *konservative Erweiterungen* sein: Theorie wird konsistent und nur durch Aussagen erweitert, die die neuen Funktionssymbole enthalten.

Nachweis der Terminierung von Funktionen über fundierte Relationen entspricht der gebräuchlichsten Art der informellen Begründung, weshalb ein Programm terminiert (z.B.: ein Zähler wird so lange monoton verändert, bis ein Abbruchkriterium erfüllt ist).

Beispiel: Größter gemeinsamer Teiler (ggT)

Für $x, y > 0$

$$ggT(x, x) = x$$

$$x > y \Rightarrow ggT(x, y) = ggT(x - y, y)$$

$$y > x \Rightarrow ggT(x, y) = ggT(x, y - x)$$

Terminierung: mit lexikographischer Ordnung $<^2$ auf Nat^2 sind Argumente der rechten Seiten jeweils kleiner als die der linken.

Beweise von Eigenschaften des ggT mit (vollständiger) Induktion (und lexikogr. Ordnung) – Beispiel Kommutativität des ggT:
um zu schließen

$$\forall x, y : Nat. \quad ggT(x, y) = ggT(y, x)$$

ist als Induktionsschritt zu zeigen:

$$\begin{aligned} \forall x, y : Nat. \quad [\forall u, v : Nat. \quad (u, v) <^2 (x, y) \Rightarrow ggT(u, v) = ggT(v, u)] \\ \Rightarrow ggT(x, y) = ggT(y, x) \end{aligned}$$

Beispiel: (ggT) (Forts.)

Induktionsannahme wird instantiiert für $(u, v) \rightsquigarrow (x - y, y)$ bzw. $(u, v) \rightsquigarrow (x, y - x)$

Abwandlung des Induktionsschemas, so daß nur die aktuell vorkommenden “Vorgänger” in der Prämisse auftauchen – durch vorherigen Terminierungsbeweis begründet:

$\forall x, y : \text{Nat.}$

$$(x > y \wedge y > 0 \Rightarrow \text{ggT}(x - y, y) = \text{ggT}(y, x - y)) \quad \wedge$$

$$(x > 0 \wedge y > x \Rightarrow \text{ggT}(x, y - x) = \text{ggT}(y - x, x))$$

$$\Rightarrow \text{ggT}(x, y) = \text{ggT}(y, x)$$

Terminierung von allgemein-rekursiven Funktionen

Beispiel: Ackermann-Funktion:

$$A : \text{Nat}^2 \rightarrow \text{Nat}$$

definiert durch Gleichungen:

$$A(0, y) = y + 1$$

$$A(x + 1, 0) = A(x, 1)$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y))$$

Mit lexikographischer Ordnung $<^2$ auf Nat^2 läßt sich zeigen, daß die Argument-Paare auf der rechten Seite jeweils kleiner sind.

Für die 3. Gleichung müssen beide Aufrufe betrachtet werden:

innerer Aufruf: $(x + 1, y) <^2 (x + 1, y + 1)$

äußerer Aufruf: $(x, z) <^2 (x + 1, y + 1)$ für beliebiges $z \in \text{Nat}$

Die Ackermann-Funktion ist das Standard-Beispiel für eine totale rekursive Funktion, die nicht primitiv-rekursiv ist.

Beispiel einer nicht immer terminierenden Funktion (über Nat):

$$\begin{aligned} F(0, y) &= 1 \\ F(x + 1, y) &= F(x, F(x + 1, y)) \end{aligned}$$

Für den inneren Aufruf sind die Argumente nicht kleiner (unter *jeder* möglichen Ordnung).

Aber: das zweite Argument kann ignoriert werden, wenn immer nur der äußere rekursive Aufruf ausgewertet wird (*call-by-name*): das Ergebnis hängt von der benutzten *Berechnungsregel* ab.

Fundierte Induktion in PVS

```
% PVS prelude:
% aus der Theorie "orders [T: TYPE]", mit
%   < : VAR pred[[T,T]]
%   p : VAR pred[T]

well_founded?(<): bool =
  (FORALL p: (EXISTS y: p(y))
    IMPLIES (EXISTS (y:(p)):
      (FORALL (x:(p)): (NOT x < y))))
```

Fundierte Induktion in PVS (2)

```
wf_induction [T: TYPE, <: (well_founded?[T])]: THEORY
BEGIN

  wf_induction: LEMMA
    (FORALL (p: pred[T]):
      (FORALL (x: T):
        (FORALL (y: T): y<x IMPLIES p(y))
          IMPLIES p(x))
      IMPLIES
        (FORALL (x:T): p(x)))

END wf_induction
```

Induktion mit Maßfunktion in PVS

```
% measure_induction builds on well-founded induction.  
% It allows induction over a type T for which a measure  
% function m is defined.
```

```
measure_induction [T: TYPE, M: TYPE,  
    m: [T -> M],  
    <: (well_founded?[M])]: THEORY
```

```
BEGIN
```

```
measure_induction: LEMMA  
  (FORALL (p: pred[T]):  
    (FORALL (x: T):  
      (FORALL (y: T): m(y) < m(x) IMPLIES p(y))  
      IMPLIES p(x))  
    IMPLIES (FORALL (x: T): p(x)))
```

```
END measure_induction
```