

6. Induktion

Warum braucht man Induktion?

1. *Ein Induktionsaxiom/-schema schließt Nichtstandard-Modelle aus*

(s.o. Logik höherer Stufe)

Ein *Induktionsschema* steht für eine (unendliche) Menge von (PL1-)Instanzen des Induktionsprinzips; ein Induktionsaxiom (in der Form einer PL2-Formel) drückt dasselbe etwas direkter (und sauberer) aus.

2. Mit Hilfe von *Induktion als Beweisprinzip* können Aussagen bewiesen werden, die anders (d.h. in PL1) nicht bewiesen werden können.

Z.B. Prinzip der *mathematischen Induktion* über den natürlichen Zahlen Nat

Beispiel: Peano-Arithmetik

Die Peano-Arithmetik (elementare Arithmetik; nach dem ital. Mathematiker Peano, 1889) ist die Theorie der natürlichen Zahlen mit Addition und Multiplikation, $\mathcal{N}_{+,*}$, definiert durch das Axiomensystem PA :

$$\forall x. \neg(0 = s(x)) \quad (\text{constr})$$

$$\forall x, y. s(x) = s(y) \Rightarrow x = y \quad (\text{inj})$$

$$\forall x. x + 0 = x \quad (\text{add0})$$

$$\forall x, y. x + s(y) = s(x + y) \quad (\text{add1})$$

$$\forall x. x * 0 = 0 \quad (\text{mul0})$$

$$\forall x, y. x * s(y) = x * y + x \quad (\text{mul1})$$

$$\forall P : (\mathcal{N} \rightarrow \text{Bool}). \quad (\text{induct})$$

$$P(0) \wedge (\forall x. P(x) \Rightarrow P(s(x))) \Rightarrow \forall x. P(x)$$

plus Axiome über Gleichheit

Peano-Arithmetik (2)

Das Axiom (induct) definiert das *Induktionsprinzip* für natürliche Zahlen:

Um eine Formel $\forall x.Q(x)$ zu beweisen, muß gezeigt werden

- (i) $Q(0)$ (Induktions-Anfang)
- (ii) $Q(s(x))$ unter der Annahme $Q(x)$, für beliebiges x (Induktions-Schritt)

Man kann zeigen, daß obige Axiomatisierung \mathcal{N}_{+*} bis auf Isomorphie charakterisiert.

Die in PA gültigen Sätze sind nicht rekursiv aufzählbar.

Insbesondere ist also PA nicht entscheidbar.

Es gibt entscheidbare Untertheorien von PA , z.B. die *Presburger-Arithmetik*, die nur lineare Arithmetik (keine Multiplikation) enthält.

Peano-Arithmetik: Beispielbeweis

Beweise die Formel $Q := \forall x. (x + 1 = 1 + x)$ aus den Axiomen PA :

Beweisstruktur:

$$\frac{\frac{\frac{PA \vdash IA \quad PA \vdash IS}{PA \vdash IA \wedge IS}}{PA, (IA \wedge IS \Rightarrow Q) \vdash Q}}{PA, \forall P. P(0) \wedge (\forall x. Px \Rightarrow P(s(x))) \Rightarrow \forall x. P(x) \vdash Q}^{(*)}$$

Bei dem Schritt $(*)$ ist für P einzusetzen die Funktion

$$\sigma := \lambda z. (z + 1 = 1 + z)$$

Dadurch wird $P(0)$ zum Induktions-Anfang:

$$IA := 0 + 1 = 1 + 0$$

$\forall x. P(x) \Rightarrow P(s(x))$ wird zum Induktions-Schritt

$$IS := \forall x. (x + 1 = 1 + x \Rightarrow s(x) + 1 = 1 + s(x))$$

$\forall x. P(x)$ wird gleich der zu beweisenden Formel Q :

$$\forall x. x + 1 = 1 + x$$

Peano-Arithmetik: Beispielbeweis (2)

Die einzigen verbleibenden Beweisziele sind:

$$PA \vdash IA$$

und

$$PA \vdash IS$$

Sie lassen sich mit Hilfe der übrigen Axiome beweisen, d.h. ohne Anwendung von Induktion.

Bemerkung: Formal gesehen ist der Beweis nur in Logik höherer Ordnung möglich, aber Elemente der Logik höherer Ordnung werden nur für die Instantiierung der Induktionsformel (und für die zugehörigen Inferenzen) benötigt; der Rest des Beweises ist dann wie in einem “normalen” PL1-Beweis.

Induktion in PVS

In PVS ist der Typ `nat` vorgegeben, ebenso die zugehörige Formel für Induktion über `nat`.

(Streng genommen handelt es sich in PVS hierbei *nicht* um ein Axiom – s. später.)

```
p: VAR pred[nat]
```

```
nat_induction: LEMMA
```

```
(p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))  
  IMPLIES (FORALL i: p(i))
```

Alternativ könnte die Formel über `p` all-quantifiziert geschrieben werden.

Induktionsbeweise in PVS

In PVS ist es (in der Regel) nicht notwendig, die Induktionsformel explizit zu instantiieren mit einem passenden Prädikat.

Stattdessen: Beweiserkommando `induct`

`(induct "n")`

– generiert für eine Zielformel die entsprechenden Unterziele Induktionsanfang und Induktionsschritt, entsprechend der zugehörigen Induktionsformel (Beispiele für andere Typen kommen später)

Für einen Induktionsbeweis ist die wichtigste Entscheidung: welches ist das Induktionsargument?

Im einfachsten Fall: `trivial` (wenn es nur einen Kandidaten gibt)

Rekursive Funktionen

Rekursive Funktionen können in PVS nicht einfach wie nicht-rekursive Funktionen definiert werden, da der zu definierende Name im Rumpf der Definition (der rechten Seite nach dem =) nicht verwendet werden kann.

Alternative 1: Deklaration des Funktionsnamens als uninterpretiertes Funktionssymbol, gefolgt von einer *Formel*, die die Rekursionsgleichung enthält und als Axiom anzusehen ist.

```
fakt : [nat -> nat]
fakt_ax : AXIOM
      fakt(n) = IF n=0 THEN 1 ELSE n * fakt(n-1) ENDIF
```

Alternative 2: Explizite rekursive Definition mit Angabe des Schlüsselworts RECURSIVE vor dem Resultat-Typ

```
fakt(n: nat): RECURSIVE nat =
      IF n=0 THEN 1 ELSE n * fakt(n-1) ENDIF
...

```

Rekursive Funktion: Axiom oder Definition?

Axiome sind einfacher hinzuschreiben, sind aber potentiell gefährlich:

Axiome können inkonsistent sein!

Es ist daher sicherer, eine definatorische Form der Rekursion zu benutzen.

Definitionen sollen *konservative Erweiterungen* einer Theorie sein:

Theorie , d.h. die Menge der beweisbaren Aussagen ("Theoreme"), soll durch Hinzu-
nahme einer Definition nicht vergrößert werden.

PVS erzwingt bei Definitionen diese Eigenschaft, insbesondere durch die Forderung,
dass alle Funktionen als *total* nachgewiesen werden.

Totale rekursive Funktionen

In der 'normalen' Prädikatenlogik werden alle Funktionen als *total* vorausgesetzt. Partialität entsteht in natürlicher Weise bei rekursiven Funktionen.

~> Logik muß mit partiellen Funktionen umgehen können.

Mögliche Alternativen: z.B.

- Mehrwertige Logik (z.B. 3-wertig: wahr, falsch, unbestimmt ('bottom')); siehe "Logic of Computable Functions", LCF)
- Logik partieller Funktionen
Problem z.B.: was bedeutet All-Quantifizierung über ein nur partiell definiertes Prädikat?

Die Alternativen haben ihre eigenen Probleme; PVS unterstützt (direkt) keine.

Frage: wie sieht es aus mit Funktionen, die inhärent partiell sind?

Z.B. für Division über reellen Zahlen: Division durch 0?

Rekursive Funktionen: Terminierung

PVS erlaubt es in vielen Fällen, inhärent partielle Funktionen auf ihren tatsächlichen Definitionsbereich einzuschränken und damit total zu machen.

↪ Erweiterung des Typsystems um *Sub-* oder *Untertypen* (kommt später)

Für nicht-rekursive Funktionen ist die Forderung nach Totalität unproblematisch (warum?).

Für rekursiv definierte Funktionen erzwingt PVS den Nachweis der Terminierung:

In der Syntax: In der rekursiven Definition muß eine *Maßfunktion* angegeben werden:

```
fakt(n: nat): RECURSIVE nat =  
    IF n=0 THEN 1 ELSE n * fakt(n-1) ENDIF  
MEASURE x
```

x steht hier für die Identität (LAMBDA (x:nat): x)

Generell: bei Angabe der Maßfunktion kann Lambda-Bindung fortgelassen werden, es genügt Angabe des Rumpfs der Funktion.

Fundierte Relationen

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

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

Satz: Es sind äquivalent:

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

$$x_1 \succ x_2 \succ \dots \succ x_n \succ \dots$$

Beweis:

(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 bezgl. \succ kleineres Element (x_{i+1}), daher hat M kein minimales Element.

Fundierte Relationen (Forts.)

Beweis: (i) \Leftarrow (ii):

Angenommen, \succ 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 wird 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 folgende Relation auf Nat ist fundiert, aber nicht transitiv:

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

- (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 Komponente gegeben ist.

- (4) Die Teillisten-Relation auf linearen Listen:

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

Allgemein sind Teilstruktur-Relationen auf induktiv definierten Datenstrukturen fundiert (mehr hierzu später).

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”.

Fundierte Relationen (3)

Fundierte (Ordnungs-)Relationen sind wichtig für

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

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).

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

Rekursive Funktionen: Terminierung (2)

In der Semantik: Die Maßfunktion bildet das Rekursionsargument in eine geordnete Menge ab.

Im einfachsten Fall (wie hier): Abbildung in die Menge der natürlichen Zahlen.

Bezüglich der Maßfunktion müssen die Argumente der rekursiven Aufrufe im Funktionsrumpf kleiner werden.

Wenn es ein kleinstes Element in der Maßmenge gibt, folgt aus der Bedingung, dass die Rekursion terminiert.

PVS generiert hierfür während der Typüberprüfung eine *Terminierungs-TCC* - hier:

```
fakt_TCC: OBLIGATION
  FORALL (n: nat): NOT n=0 IMPLIES n-1 < n
```

Die TCC enthält als Vorbedingung die zutreffende Bedingung

~> Herstellen des richtigen Kontexts für den Beweis

~> TCCs (“*type correctness conditions*”) werden in Kürze allgemein behandelt.

Induktion über fundierten Relationen

Eine fundierte Relation kann herangezogen werden zur Definition eines Induktionssprinzips:

fundierte oder Noethersche Induktion

benannt nach der Mathematikerin Amalie Emmy (“Emily”) Noether (1882–1935)

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 A 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

Induktion über fundierte Relationen (2)

Beispiel 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 (wieso?).

Weitere Beispiele kommen bei der Behandlung induktiver Datenstrukturen.

Die allgemeinste Form der Induktion ist *fundierte Induktion* (engl. *well-founded induction*), die eine fundierte Relation als Grundlage erfordert.

Andere Formen der Induktion können daraus abgeleitet werden.

Fundierte Relationen in PVS

Die Eigenschaft, fundiert zu sein, kann in PVS als Prädikat (höherer Stufe) über Relationen ausgedrückt werden:

```
< : VAR pred[[T,T]]
```

```
p : VAR pred[T]
```

```
x, y, z : VAR T
```

```
well_founded?(<): bool =
```

```
  (FORALL p: (EXISTS x: p(x))
```

```
    IMPLIES (EXISTS y: p(y) AND
```

```
      (FORALL z: p(z) IMPLIES (NOT z < y))))
```

Der PVS-Prelude enthält allgemeine Formulierungen für fundierte Relationen und fundierte Induktion.

Einschub: Prädikative Untertypen in PVS

PVS erlaubt es, einen Untertyp eines existierenden Typs zu definieren durch Angabe eines Prädikats, das die dem Untertyp zugrundeliegende Teilmenge bestimmt:

```
p: [T -> bool]
sub: TYPE = { x:T | p(x) }
```

Die Definition eines Untertyps ist immer relativ zu einem Basistyp ('Obertyp').
sub ist der Typ all derjenigen Elemente des Typs T, die das Prädikat p erfüllen.

Alternative, abkürzende Schreibweise:

(p) bezeichnet denselben Untertyp wie $\{x:T \mid p(x)\}$

d.h. *jedes* Prädikat kann zur Definition eines Untertyps herangezogen werden.

Untertypen können in PVS wie Typen benutzt werden.

Sie sind insbesondere nützlich bei der Spezifikation von Funktionen, die andernfalls nur partiell wären.

Untertypen (2)

Beispiele:

`nznat: TYPE = { n: nat | n > 0 }`

`evennat: TYPE = { n: nat | even(n) } [= (even)]`

`non_zero(x: num): boolean = (x /= 0)`

`div(x: num, y: (non_zero)): num = ...`

Untertypen (3)

Konsequenzen der Einführung von prädikativen Untertypen:

- Typüberprüfung ist nicht mehr rein statisch (“syntaktisch”) durchführbar, sondern ist im allgemeinen Fall unentscheidbar und erfordert Beweise; ‘Typ’ wird zu einem *semantischen* Begriff.
 \rightsquigarrow Integration von Typüberprüfung und Beweisen in PVS
- Es werden *Typkorrektheitsbedingungen* (TCCs) erzeugt, sobald z.B.
 - die Existenz eines Elements im Untertyp behauptet wird;
 - als Funktionsargument ein Element eines Untertyps gefordert wird, für das aktuelle Argument aber nur Zugehörigkeit zum Basistyp sichergestellt ist.

TCCs müssen wie andere Lemmata, auf denen ein Beweis aufbaut, bewiesen werden, damit ein Beweis vollständig ist (bzw. damit eine Spezifikation als konsistent akzeptiert werden kann).

Parametrisierte Typen in PVS

Typen können von Werten abhängen, d.h. parametrisiert sein.
Beispiele aus dem `prelude`:

```
m: VAR nat
upto(m): NONEMPTY_TYPE = {n: nat | n <= m} CONTAINING m
below(m): TYPE = {n: nat | n < m}
```

auch mit mehreren Parametern:

```
i, j: VAR int
subrange(i, j): TYPE = {k: int | i <= k AND k <= j}
```

Ein `subrange`-Typ kann offensichtlich auch leer sein.

Fundierte Induktion in PVS

Zur Erinnerung: Definition eines Prädikats `well_founded?(<)` über Relationen
(jetzt in Notation mit prädikativen Subtypen)

(Aus der `prelude`-Theorie `orders`)

```
< : VAR pred[[T,T]]
```

```
p : VAR pred[T]
```

```
x, y, z : VAR T
```

```
well_founded?(<): bool =
```

```
  (FORALL p: (EXISTS x: p(x))
```

```
    IMPLIES (EXISTS (y:(p)): (FORALL (x:(p)): (NOT x < y))))
```

Fundierte Induktion in PVS (2)

```
% wf_induction defines induction for any type that  
% has a well-founded relation.
```

```
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
```

Fundierte Induktion in PVS (3)

Induktion mit Maßfunktion in PVS, abgeleitet von fundierter Induktion

```
% 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
```

Induktive Definitionen

Ein Prädikat kann induktiv definiert werden und damit die Basis für einen Induktionsbeweis abgeben.

PVS unterstützt induktive Definition durch die Generierung der entsprechenden Induktionsformeln.

Beispiel: gerade Zahlen

```
even(n:nat) INDUCTIVE bool =  
  n=0 OR (n>1 AND even(n-2))
```

Für induktive Definitionen gelten Einschränkungen bzgl. Terminierung wie für rekursive Funktionen.

Induktive Definitionen (2)

Die Definition von `even` kann aufgefasst werden als die Zusammenfassung von 2 'Regeln':

1. `even(0)`

2. `even(n) => even(n+2)`

und der "Abschlußeigenschaft": `even(x)` gilt genau dann, wenn es aufgrund der Regeln (1) und (2) der Fall ist – die Menge $\{n : nat \mid even(n)\}$ ist die *kleinste* Menge, die unter den Regeln abgeschlossen ist.

Diese Struktur wird ausgenutzt, um induktiv eine Eigenschaft für alle Elemente der Menge, die durch das (induktiv definierte) Prädikat charakterisiert wird, zu beweisen.

Induktive Definitionen (3)

Zugehöriges Induktionsaxiom in PVS:

```
even_weak_induction: AXIOM
  (FORALL (P: [nat->bool]):
    (FORALL (n:nat): n=0 OR (n>1 AND P(n-2)) IMPLIES P(n))
    IMPLIES (FORALL (n:nat): even(n) IMPLIES P(n)))
```

In einer zweiten Form des Induktionsaxioms steht die Eigenschaft $\text{even}(n)$ in der Voraussetzung des Induktionsschritts zur Verfügung:

```
even_induction: AXIOM
  (FORALL (P: [nat->bool]):
    (FORALL (n:nat): n=0 OR (n>1 AND even(n-2) AND P(n-2)) IMPLIES P(n))
    IMPLIES (FORALL (n:nat): even(n) IMPLIES P(n)))
```


Einschub II: Abhängige Typen in PVS

Ein Typ ist ein *abhängiger Typ* (engl. *dependent types*), wenn eine Komponente vom *Wert* einer vorherigen Komponente abhängt.

Beispiel:

```
datum: TYPE = [# jj: jahr, mm: monat,  
               tt: {t: posnat | t <= tage(mm,jj) } #]
```

Der Typ der Tag-Komponente `tt` des Datums hängt von den Werten für das Jahr und den Monat ab, hier ausgedrückt durch eine (geeignet definierte) Funktion `tage`.

Abhängige Typen können beitragen zur Verschärfung bzw. genaueren Spezifikation von Definitions- und Wertebereich von Funktionen:

~> Vermeidung von Unbestimmtheit (“Partialität”) von Funktionen

Abhängige Typen (2)

Eine endliche Folge der Länge n aus Elementen des Typs T kann repräsentiert werden durch eine Funktion über dem Index-Bereich $\{0, \dots, n-1\}$; als Typ in PVS:

```
seq: [below(n) -> T]
```

Typ endlicher Folgen *beliebiger* Länge (engl. *finite sequences*):

```
finseq: TYPE = [# length: nat, seq: [below(length) -> T] #]
```

Beispiel: Zugriff auf n -tes Element einer Folge

```
nth(s:finseq, n: below(s'length)): T = s'seq(n)
```

Typüberprüfung stellt sicher, daß nicht versucht wird, auf nicht-existierende Elemente der Folge zuzugreifen.

↪ Einschränkung des Definitionsbereichs der Funktion `nth` durch geeignete Typisierung

Abhängige Typen (3)

Operationen auf endlichen Folgen: z.B. Verkettung

```
o(fs1, fs2: finseq): finseq =  
    LET l1    = fs1'length,  
        lsum = l1 + fs2'length  
    IN (# length := lsum,  
        seq      := (LAMBDA (n:below(lsum)):  
                      IF n < l1  
                        THEN fs1'seq(n)  
                        ELSE fs2'seq(n-l1)  
                      ENDIF) #);
```

Typkorrektheit der rechten Seite?

Abhängige Typen (4)

Weiteres Beispiel:

```
mod: THEORY
  i,k: VAR int
  j:   VAR nonzero_int
  ...
  mod(i,j): { k | abs(k) < abs(j) } = ...
```

- Wertebereich ist eingeschränkt in dem Sinn, dass der *Rumpf* der Funktionsdefinition einen Wert bestimmen muß, der dem angegebenen Typ angehört.
- Bei Aufruf der Funktion `mod` wird dann garantiert, daß der Funktionswert die angegebene Eigenschaft hat.
- Typinformation für `mod` bzw. einen Aufruf `mod(i,j)` kann dem Beweiser übermittelt werden mit dem Kommando `TYPEPRED`.

Parametrisierte Theorien

Eine Theorie kann parametrisiert werden mit

- Typen
- Konstanten (einschließlich Funktionskonstanten)

Theorie-Parameter ergeben zum Teil denselben Effekt wie *polymorphe* oder *generische Typen*

Beispiel: generischer Typ von endlichen Folgen über einem beliebigen Elementtyp T

```
finseqs [T: TYPE]: THEORIE
BEGIN
  finseq: TYPE = [# length: nat, seq: [below(length) -> T] #]
  ...
END
```

Parameter können im Theorie-Rumpf genauso benutzt werden wie “normal” deklarierte Typen und Konstanten.

Parametrisierte Theorien (2)

In PVS ist die Parametrisierung von `finseq` nur über den Umweg der parametrisierten Theorie möglich, nicht direkt.

Streng genommen gibt es in PVS keine *Typ-Variablen*, in dem Sinne wie es Variable für Grundobjekte, Funktionen, Prädikate usw. gibt; es gibt auch keine *Quantifizierung* über Typen.

Dies erfordert eine mächtigere und komplexere Logik: eine *Typentheorie*.

Instanziierung einer parametrisierten Theorie: durch Import

~> wird in Kürze behandelt