Rechnergestützte Beweissysteme

Freiheit und Induktion

Axiome für die natürlichen Zahlen

- Spezifiziere $\mathbb{N}$ durch logische Axiome.
  - In der bisher betrachteten Prädikatenlogik 1. Stufe unmöglich
  - Ein Axiom der Prädikatenlogik 2. Stufe wird benötigt: (Induktionsaxiom)
- Spezifiziere Listen, Stacks, Queues, etc.
  - Es treten die gleichen Probleme auf
  - Aber auch die Lösung ist analog
- PVS hat dafür geeignete Mechanismen
  - `DATATYPE` , etc.

Die natürlichen Zahlen

- Menge Nat mit Operationen

  o : Nat -- eine Konstante

  s : Nat → Nat -- die Nachfolgeroperation

- Jede natürliche Zahl lässt sich damit gewinnen, aber welche Axiome brauchen wir ?

- Es gibt viele Strukturen mit einer Konstanten 0 und einer Nachfolgeroperation, z.B.:

- N = {0,1,2,3,4,5,6,7,8,9} mit
  - o := 0,
  - s(x) = (x+1) mod 10

  N = int, o := 0, s(x) = x+1

- Wir müssen diese Un-Natürlichen-Zahlen ausschließen.

Freiheitsaxiome

- Wir wollen ausdrücken, dass sich jede natürliche Zahl auf genau eine Weise aus 0 und s gewinnen lässt.
  - z.B.: 4 = s(s(s(s(0))))
- genau eine Weise zerfällt in
  1. höchstens eine Weise
  2. mindestens eine Weise
- Die erste Aussage kann man in der Logik erster Stufe ausdrücken:
  - $\forall x \in \text{Nat}.\ \neg\ 0 = s(y)$
  - $\forall x,y \in \text{Nat}.\ s(x) = s(y) \Rightarrow x=y$
- Diese Axiome nennt man „Freiheitsaxiome".
- Die zweite Aussage lässt sich nicht in der Logik erster Stufe ausdrücken !!!

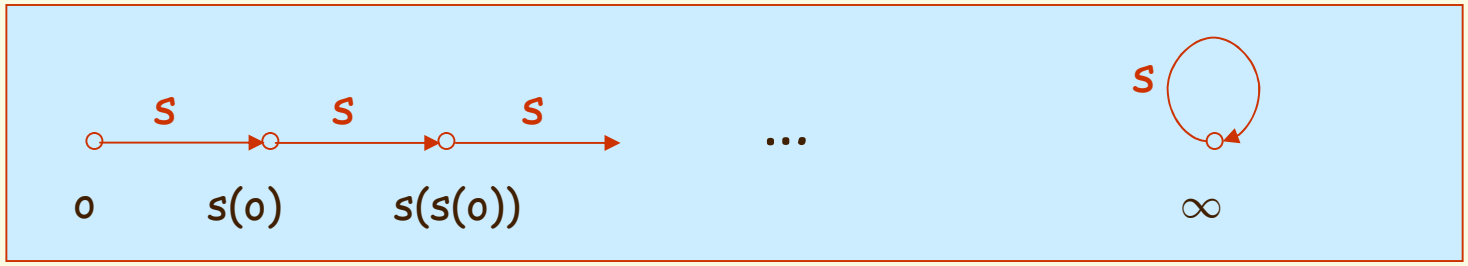
Freiheit für $\mathbb{N}$

- Die Freiheitsaxiome für $\mathbb{N}$ besagen, dass zwei Zahlen genau dann gleich sind, wenn
  - beide zero sind, oder
  - beide nicht zero und
    - `pred(l) = pred(m)`
- M.a.W.: Zwei Zahlen sind nur dann gleich, wenn sie gleich aufgebaut sind:

```
freiheit: AXIOM
  FORALL (l,m:nat): l = m
        IFF isZero?(l) = isZero?(m)
            OR    isSucc?(l)
              AND isSucc?(m)
              AND pred(l) = pred(m)
```

Modell mit Freihaitsaxiomen

- Das folgende Modell (mit $s(\infty)=\infty$) erfüllt die Freiheitsaxiome :

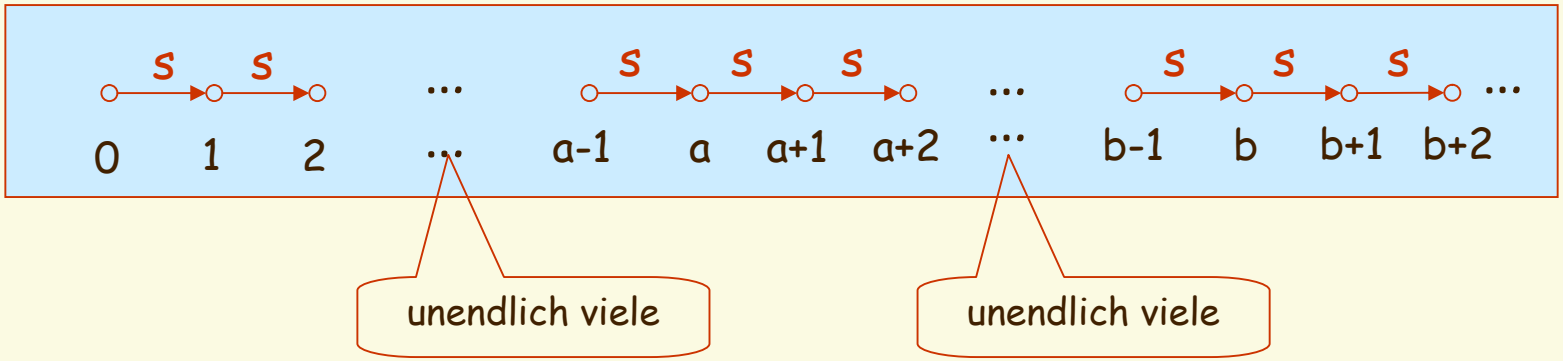


- Jedes Element lässt sich auf höchstens eine Weise aus 0 und s gewinnen, nicht aber auf mindestens eine, z.B. : ∞

- Dieses Modell könnten wir noch durch zusätzliche Axiome der PL-1 ausschliessen, z.B. durch
  - $\forall x \in \text{Nat}.\ \neg\, x = s(x)$

Brauchen wir mehr Axiome ?

- Im folgenden Modell gelten alle PL1-Aussagen, die in $\mathbb{N}$ gelten. Wir können es also nicht durch zusätzliche Axiome der PL-1 ausschließen.
- Es erfüllt genau die gleichen Axiome der PL-1 wie die richtigen natürlichen Zahlen
  - Solche Modelle heißen Nonstandard-Modelle.



Beschränktheit der Logik 1. Stufe

Wenn wir rekursiv definieren

- x + o := x
- x + s(n) = s(x+n)

folgt z.B. :

- 0 + 0 = 0
- 0 + s(0) = s(0)
- 0 + s(s(0)) = s(s(0))
- 0 + s(s(s(0))) = s(s(s(0)))

Wir können aber nicht zeigen:

- ∀ x:nat. s(o) + x = x + s(o)

Woran liegt das ?

- Sind wir zu doof ?
- Ist unser Beweiskalkül zu schwach ?
- Ist unsere logische Sprache zu beschränkt?

Grund: Die letzte Aussage gilt nicht im Nonstandardmodell

Was in PL-1 nicht geht

- Man möchte gerne sagen:

$$\forall n{:}Nat.\ n = \underbrace{s(s(\ldots s}_{n\text{-mal}}(o)\ldots))$$

  aber das kann man in der Syntax der PL-1 nicht hinschreiben

- Auch so geht es nicht:
  - $\forall n{:}nat.\exists m{:}nat.\ n = s^m(o)$, oder
  - $\forall n{:}nat.\ n = s^n(o)$,

- denn Exponentiation $s^y(x)$ ist keine Operation.

Induktionsaxiom

Das Induktionsaxiom

$$\forall P \subseteq N.\ (0 \in P \wedge (\forall\ k{:}N.\ k \in P \rightarrow s(k) \in P) \rightarrow \forall\ n{:}N.\ n \in P)$$

ist ein Ausdruck der PL-2, da über Teilmengen P quantifiziert wird. Wir benutzen es als Beweisschema:

$$\frac{P(0)\ ,\quad \forall\ k : Nat.\ P(k) \Rightarrow P(s(k))}{\forall x : Nat.\ P(x)}$$

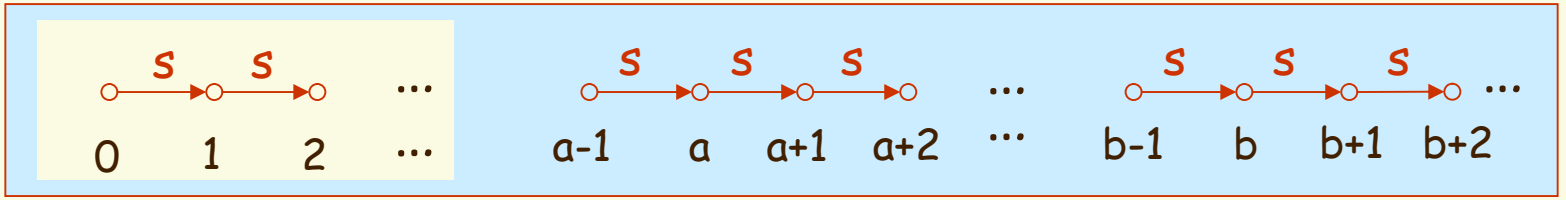
- PVS erlaubt Logik höherer Stufe, daher können wir es dort axiomatisieren (Teilmengen werden durch char. Fkt. ersetzt):

```
FORALL (X:[Nat -> bool]):
      ( X(o) AND (FORALL (k:Nat): X(k) IMPLIES X(k+1))
               IMPLIES  (FORALL (n:Nat): X(n)
```

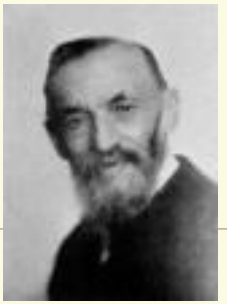
Im Nichtstandard Modell

- Im Nichtstandard-Modell bleiben nur 0 und die Elemente der Form s(s(…s(o)…) übrig, denn diese Menge erfüllt die Prämisse des Induktionsaxioms

$$\frac{P(0)\ ,\ \forall\ k : Nat.\ P(k) \Rightarrow P(s(k))}{\forall x : Nat.\ P(x)}$$



Peano Axiome


Guiseppe Peano

- Die Peano-Axiome bestehen aus
  - den Axiomen
    - $\forall$ x:nat. $\neg$ s(x) = 0
    - $\forall$ x,y:nat. s(x) = s(y) $\Rightarrow$ x = y
    - $\forall$ x:nat. x + o := x
    - $\forall$ x,y:nat. x + s(y) = s(x+y)
  - dem Induktionsaxiom
- Mit dem Induktionsaxiom folgt u.a.
  - Jede nat. Zahl ist von der Gestalt s(s(…s(o)…))
  - Zu jeder Zahl n gibt es nur endlich viele Zahlen k < n
  - Addition und Multiplikation sind assoziativ und kommutativ
- Durch diese Axiome sind die natürlichen Zahlen bis auf Isomorphie eindeutig festgelegt.
  - das können Logiker beweisen …

Weitere Ergebnis der Logik

- Löwenheim-Skolem:
  - Sei Ax eine Menge von Aussagen der PL1. Wenn Ax ein unendliches Modell hat, dann hat Ax auch ein Modell jeder größeren Kardinalität.
- Folgerung:
  - Sei Th_{Nat} die Menge aller PL-1-Aussagen, die in den *„richtigen"* natürlichen Zahlen gelten. Dann gibt es auch eine Struktur mit überabzählbar vielen Elementen, in der ebenfalls alle Aussagen aus Th_{Nat} wahr ist.
- Konsequenz :
  - Wir können die natürlichen Zahlen in der Prädikatenlogik erster Stufe nicht eindeutig festlegen
  - In der PL-2 geht das – mit dem Induktionsaxiom.

Natürliche Zahlen - selbstgemacht

```
Natuerlich  : THEORY
BEGIN
  Nat : TYPE+
  O : Nat
  s : [Nat -> Nat]

  s_injektiv: AXIOM
    FORALL (m,n:Nat):
      s(m) = s(n) IMPLIES m=n

  O_ungleich_s : AXIOM
    FORALL (m: Nat) : NOT s(m)=O

  % Induktionsaxiom

  ind_Nat: AXIOM
    FORALL (X:[Nat -> bool]):
      X(O) AND
      (FORALL (k:Nat): (X(k) IMPLIES X(s(k))))
        IMPLIES (FORALL (n:Nat): X(n))

END Natuerlich
```

Konstruktoren : o und s

Freiheitsaxiome
Zwei Objekte sind nur gleich, wenn sie gleich erzeugt wurden

Induktionsaxiom: Axiom 2. Stufe !
Hier wird über Mengen bzw. Abbildungen quantifiziert

Anwendung des Induktionsaxioms

- Mit dem Induktionsaxiom folgt u.A. dass die Addition kommutativ ist:

```
FORALL (n:Nat):  x + y = y + x
```

- Vorher empfiehlt es sich ein Lemma zu zeigen:

```
FORALL (n:Nat):  0 + n = n
```

- Dabei muss das X im Induktionsaxiom instanziiert werden mit der char. Funktion der Menge `{n | 0 + n = n}`:

- Eine einfache Möglichkeit wäre, einen *Hilfsterm*

```
ind_hilf: [Nat -> bool]
```

einzuführen mit einem Axiom

```
hilf_ax : AXIOM  FORALL (n:Nat): ind_hilf(n) = (0+n=n)
```

- Eine bessere Möglichkeit wäre, eine *Hilfsfunktion* einzuführen mit einer Definition

```
hilf_ax(n:Nat): bool = (0+n=n)
```

Funktionsobjekte – Beta Reduktion

Statt neue Funktionen oder Terme einzuführen, kann man das Induktionsaxiom auch direkt mit Funktionsobjekten instanziieren:

```
(LAMBDA (n:T) expr )
```

ist die Funktion , welche einem `n` vom Typ `T` den Wert `expr` zuordnet.

Die Anwendung eines Funktionsobjektes auf ein Argument nennt man beta-Reduktion

```
(LAMBDA (n) (0+n=n))(5) → (0+5=5) → true
```

PVS hat dafür den Befehl

```
(beta)
```

Axiome der Addition

```
  add       : [Nat,Nat -> Nat]

  add_null : AXIOM  FORALL (n:Nat)    : add(n,0)     = n
  add_succ : AXIOM  FORALL (m,n: Nat): add(n,s(m)) = s(add(n,m))

%Testing
   links_0 : LEMMA  FORALL (n:Nat): add(0,n)=n

%1. Möglichkeit:
    hilf_def: AXIOM FORALL (n:Nat): hilf(n) iff  add(0,n) = n

%2. Möglichkeit
    hilf :[Nat -> bool] = (Lambda (n:Nat): add(0,n)=n) ;

%3. Möglichkeit:   ind_Nat instanziieren mit
%     (inst -1 "(LAMBDA (n:Nat): add(0,n)=n)")
%       (Lambda (n: Nat): add(0,n))

% Übung: Beweisen Sie:
    kommutativ : THEOREM   FORALL (m,n: Nat) : add(m,n) = add(n,m)
```

Eingebaute natürliche Zahlen

Der Benutzer muss die natürlichen Zahlen und ihre Axiome nicht in PVS definieren. Sie sind „eingebaut"

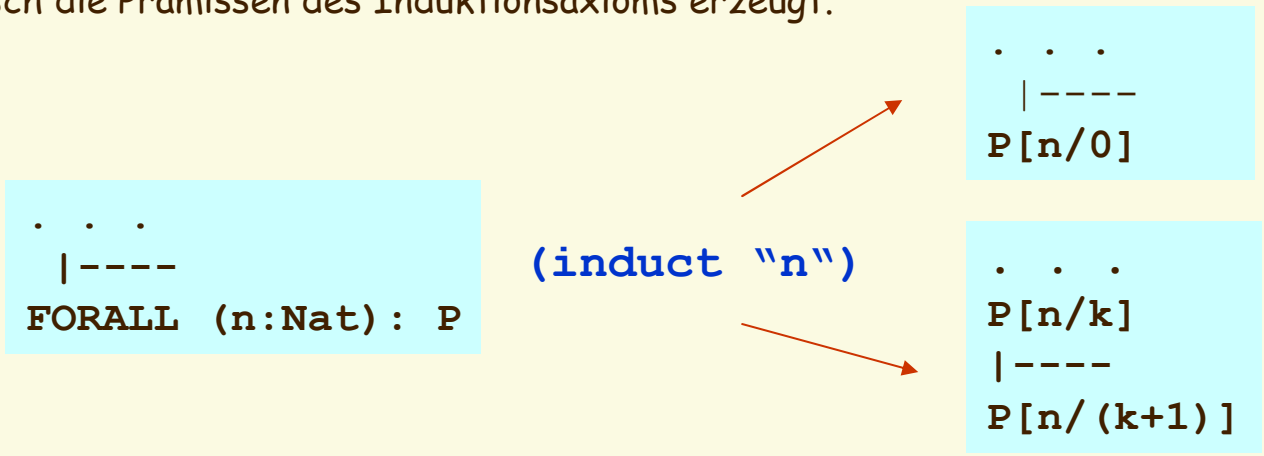
Will man eine Sequenz der Form

Δ |- Γ, `FORALL (n:Nat): P(n)`

beweisen, so werden durch den Befehl

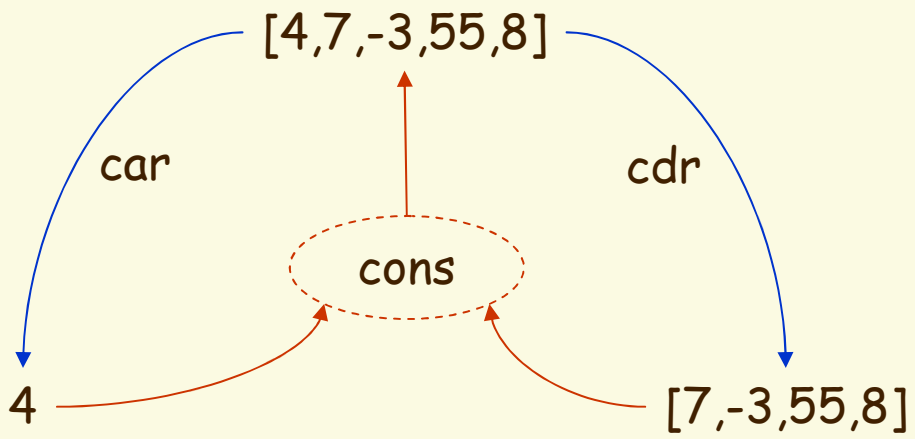
`induct`

automatisch die Prämissen des Induktionsaxioms erzeugt.



Listen

- Listen werden als parametrisierter Typ `list[T]` definiert
- Ein `list[nat]` mit 5 Elementen:



- `car` und `cdr` sind altmodische Bezeichnungen für `head` und `tail`
- Listen sind induktiv definiert:
  - `die leere Liste [] ist eine Liste`
  - `Ist x ein Objekt und l eine Liste, dann ist cons(x,l) eine Liste`

Listen in PVS

- Listen sind als parametrisierte Theorie in **`prelude.pvs`** definiert
- Sei T: TYPE irgendein Typ, dann wird der Datentyp list[T] folgendermaßen gebildet:
  - list[T] : Type
  - Konstruktoren
    - `null  : Liste[T]`
    - `cons  : [T , Liste[T] -> Liste[T]]`
  - Prädikate
    - `null?  : Liste[T] -> bool`
    - `cons?  : Liste[T] -> bool`
  - Selektoren
    - Zwei Selektoren für Cons (entsprechen den beidenArgumenten):
      - `car : (cons?) -> T`
      - `cdr : (cons?) -> Liste[T]`

Listenfunktionen

- Die Länge einer Liste

```
length(l:list[T]): RECURSIVE nat =
   IF (null?(l)) THEN 0
   ELSE 1 + length(cdr(l)) ENDIF
```

- Konkatenieren

```
append(l,m:list[T]): RECURSIVE list[T] =
   IF (null?(l)) THEN m
   ELSE cons(car(l),append(cdr(l),m)) ENDIF
MEASURE length(l)
```

Listenfunktionen

- Umdrehen einer Liste

```
reverse(l:list[T]): RECURSIVE list[T] =
   IF (null?(l)) THEN null
   ELSE append(reverse(l),cons(car(l),null) ENDIF
Measure length(l)
```

- Eine Behauptung

```
RevApp : THEOREM
 FORALL (l,m:list[T]):
   reverse(append(l,m)) =
      append(reverse(m),reverse(l))
```

- Der Induktionsbeweis benötigt das Lemma

```
appNull : LEMMA
       FORALL (l:list[T]):append(l,null) = l
```

Listeninduktion

- In prelude.pvs gibt es auch ein Induktionsaxiom für Listen.
- Es besagt:

```
FORALL (P : set[list[T]]) :
      ( (P)(null) AND
        FORALL(t:T, l:list[T]):
              (P)(l) IMPLIES (P)(cons(t,l))  )
      IMPLIES
        FORALL(l:list[T]): (P)(l)
      )
```

Freiheit – in PVS: Extensionalität

- Die Freiheitsaxiome besagen, dass zwei Listen l und m genau dann gleich sind, wenn
  - beide null sind, oder
  - beide nicht null und
    - `car(l) = car(m)` sowie
    - `cdr(l)` gleich `cdr(m)`
- M.a.W.: Zwei Listen sind nur dann gleich, wenn sie gleich aufgebaut sind:

```
freiheit: AXIOM
  FORALL (l,m:list[T]): l = m
        IFF null?(l) = null?(m)
              OR
                  cons?(l)
                AND cons?(m)
                AND car(l) = car(m)
                AND cdr(l) = cdr(m)
```