Rechnergestützte Beweissysteme

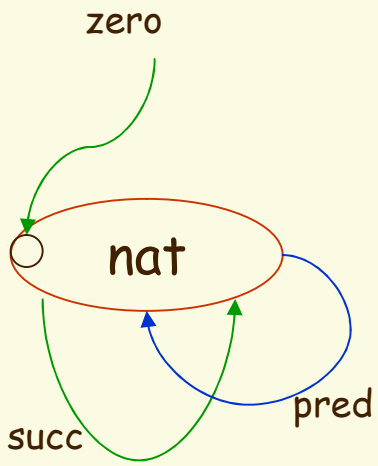
Abstrakte Datentypen

Abstrakte Datentypen

- Jeder Datentyp D besteht aus
  - Konstruktoren
    - Methoden zum Erzeugen von Elementen des Typs
  - Prädikaten
    - Sagen, mit welchem Konstruktor das Element gebaut wurde
  - Selektoren
    - Selektieren die Komponenten, aus denen das Element zusammengebaut wurde
    - Selektoren sind partielle Funktionen
- Zu jedem Datentyp D gibt es ein Induktionsprinzip

Nat als abstrakter Datentyp

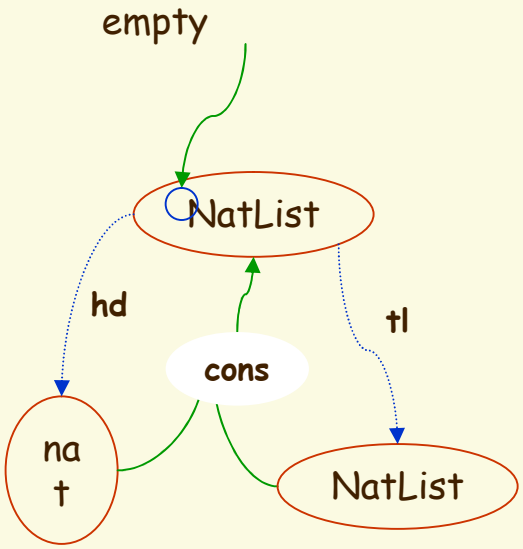
- `Nat : Type`
- Konstruktoren – konstruieren neue Elemente
  - `zero : Nat`
  - `succ : Nat -> Nat`
- Prädikate – mit welchem Konstruktor konstruiert ?
  - `isZero? : Nat -> Bool`
  - `isSucc? : Nat -> Bool`
- Selektoren – Komponenten aus denen es konstruiert wurde
  - Da `zero` keine Komponente hat, gibt es keinen Selektor
  - `pred: { n:Nat | isSucc?(n)} -> Nat`
- Einfachere Schreibweise in PVS:
  - `pred: (isSucc?) -> Nat`



Jedes Element von nat hat eine eindeutige Darstellung mittels zero und succ :
- Freiheit und
- Induktion !

NatList als Datentyp

- `NatList : Type`
  - Konstruktoren
    - `empty : NatList`
    - `cons : [nat, NatList -> NatList ]`
  - Prädikate
    - `isEmpty? : NatList -> bool`
    - `isCons? : NatList -> bool`
  - Selektoren
    - Kein Selektor für `empty`
    - Zwei Selektoren für `cons`:
      - `hd : (isCons?) -> nat`
      - `tl : (isCons?) -> NatList`



Jedes Element von NatList hat eindeutige Darstellung mittels empty und cons : Freiheit und Induktion !

Induktionsprinzip

- Sei D ein Datentyp und seien K_1, ... K_n die Konstruktoren.
  Gelte für jeden Konstruktor K_i:
  - **Wenn** :
    - P(t) für alle D-Komponenten von $K_1 \Rightarrow P(K_1(t_1,\ldots,t_{n1}))$
    - . . .
    - P(t) für alle D-Komponenten von $K_n \Rightarrow P(K_n(t_1,\ldots,t_{nk}))$
  - **Dann** gilt P für alle Elemente aus D.

- Beispiel: Konstruktoren von Nat: Zero, Succ
  - **Wenn** :
    - P(Zero)
    - P(n) $\Rightarrow$ P(Succ(n))
  - **Dann** gilt P für alle Elemente aus Nat

NatList-Induktion

- Konstruktoren von NatList: `empty`, `cons`
  - **Wenn** :
    - P(`empty`)
    - P(`l`) $\Rightarrow$ P(`cons(t,l)`)
  - **Dann** gilt P für alle Elemente aus NatList

$$\frac{P(empty), \quad \forall\, n{:}nat, \forall\, l{:}NatList{:}\ P(l) \Rightarrow P(cons(n,l))}{\forall\, l\ {:}NatList{:}\ P(l)}$$

Listeninduktion

- Konstruktoren von Liste[T]: empty, cons
  - **Wenn** :
    - P(empty)
    - P(l) $\Rightarrow$ P(cons(t, l)
  - **Dann** gilt P für alle Elemente aus Liste[T]

$$\frac{P(empty), \quad \forall\, t{:}T, \forall\, l{:}List[T]{:}\ P(l) \Rightarrow P(cons(t,l))}{\forall\, l{:}List[T]{:}\ P(l)}$$

Binärbäume mit Knoten vom Typ T

- Sei `T: TYPE` irgendein Typ, dann definieren wir Datentyp BinTree[T]
  - `BinTree[T] : Type`
  - Konstruktoren
    - `empty : BinTree[T]`
    - `mkTree : [BinTree[T], T, BinTree[T] -> BinTree[T]]`
  - Prädikate
    - `isEmpty? : BinTree[T] -> bool`
    - `isTree? : BinTree[T] -> bool`
  - Selektoren
    - Kein Selektor für `Empty`
    - Drei Selektoren für `aTree`:
      - `left : (isTree?) -> BinTree[T]`
      - `right : (isTree?) -> BinTree[T]`
      - `content : (isTree?) -> T`

BinTree-Induktion

- Konstruktoren von `BinTree[T]: empty, mkTree`
  - **Wenn** :
    - P(`empty`)
    - P(`l`), P(`r`) $\Rightarrow$ P(`mkTree(l,t,r)`)
  - **Dann** gilt P für alle Elemente aus `BinTree[T]`

$$\frac{P(\texttt{empty}),\ \forall \texttt{t:T}\ .\forall\ \texttt{l,r:BinTree[T]}.\ P(\texttt{l}) \wedge P(\texttt{r}) \Rightarrow P(\texttt{mkTree(l,t,r)})}{\forall\ \texttt{l:BinTree[T]}:\ P(\texttt{l})}$$

Freiheit für BinTrees

- Das Freiheitsaxiom für Bintree ist:

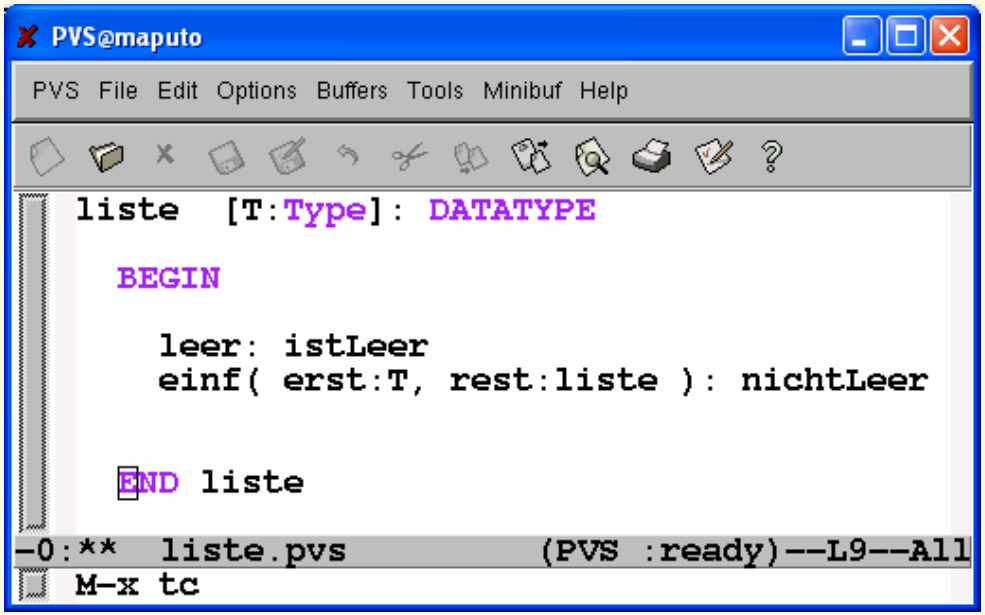
- ```
  binTreeFreiheit_1: AXIOM
    FORALL (l,m:BinTree): l = m
          IFF isEmpty?(l) AND isEmpty?(m)
                OR
                     isTree?(l) AND isTree?(m)
                     AND left(l)    = left(m)
                     AND right(l)   = right(m)
                     AND content(l)= content(m)
  ```
  ```

EZBaum[T]

- Aufgabe: Ein Knoten eine EZBaum[T] soll 1 oder 2 Söhne haben und ein Datenelement aus T:
  - `EZBaum[T] : Type`
  - **Konstruktoren**
    - `Leaf    : Baum[T]`
    - `EKnoten : [T, EZBaum[T] -> EZBaum}`
    - `ZKnoten : [EZBaum[T],T,EZBaum[T] -> EZBaum[T]}`
  - **Geben Sie Prädikate und Selektoren an**
  - **Schreiben Sie das Induktionsaxiom für EZBaum[T] auf**
  - **Geben Sie das Freiheitsaxiom für EZBaum[T] an**

Umsetzung in PVS

- Die Konstruktion von frei erzeugten Datentypen geht nach *„Schema F"*:
  Zu jedem Konstruktor gibt man gleich das zugehörige Prädikat an:
  - `Konstruktor : Prädikat`
    - `leer : istLeer?`
  - Die Bezeichner der Argumente werden zu Selektoren:
    - `einf(erst:T,rest:Liste) : nichtLeer?`



DATATYPE in PVS

- Aus der DATATYPE - Deklaration Liste.pvs generiert PVS eine Theorie Liste_adt.pvs.

- Diese enthält
  - alle Deklarationen
  - das Induktionsaxiom
  - … und vieles mehr..

Die Datentypen **`nat`**, **`list`**, und viele andere sind schon vordefiniert

- Will man einen Datentyp benutzen, so muss eine IMPORTING - Anweisung gegeben werden, z.B.:

  - **`IMPORTING list_adt`**

liste_adt – die Spezifikation

liste.pvs

```
liste  [T:Type]: DATATYPE

  BEGIN

    leer: istLeer
    einf( erst:T, rest:liste ): nichtLeer


  END liste
-0:**  liste.pvs          (PVS :ready)--L9--All
```

```
%%% ADT file generated from liste

liste_adt[T: TYPE]: THEORY
 BEGIN

  liste: TYPE

  istLeer, nichtLeer: [liste -> boolean]

  leer: (istLeer)

  einf: [[T, liste] -> (nichtLeer)]

  erst: [(nichtLeer) -> T]

  rest: [(nichtLeer) -> liste]
```

Der Befehl

M-x tc

erzeugt automatisch:

liste_adt.pvs

liste_adt – AXIOME

PVS@maputo

PVS File Edit Options Buffers Tools Help

```
liste_einf_extensionality: AXIOM
  FORALL (nichtLeer_var: (nichtLeer), nichtLeer_var2: (nichtLeer)):
    erst(nichtLeer_var) = erst(nichtLeer_var2) AND
     rest(nichtLeer_var) = rest(nichtLeer_var2)
     IMPLIES nichtLeer_var = nichtLeer_var2;

liste_einf_eta: AXIOM
  FORALL (nichtLeer_var: (nichtLeer)):
    einf(erst(nichtLeer_var), rest(nichtLeer_var)) = nichtLeer_var;

liste_erst_einf: AXIOM
  FORALL (einf1_var: T, einf2_var: liste):
    erst(einf(einf1_var, einf2_var)) = einf1_var;

liste_rest_einf: AXIOM
  FORALL (einf1_var: T, einf2_var: liste):
    rest(einf(einf1_var, einf2_var)) = einf2_var;

liste_inclusive: AXIOM
  FORALL (liste_var: liste): istLeer(liste_var) OR nichtLeer(liste_var);
```

--:%% liste_adt.pvs   (PVS :ready)--L41--11%----------

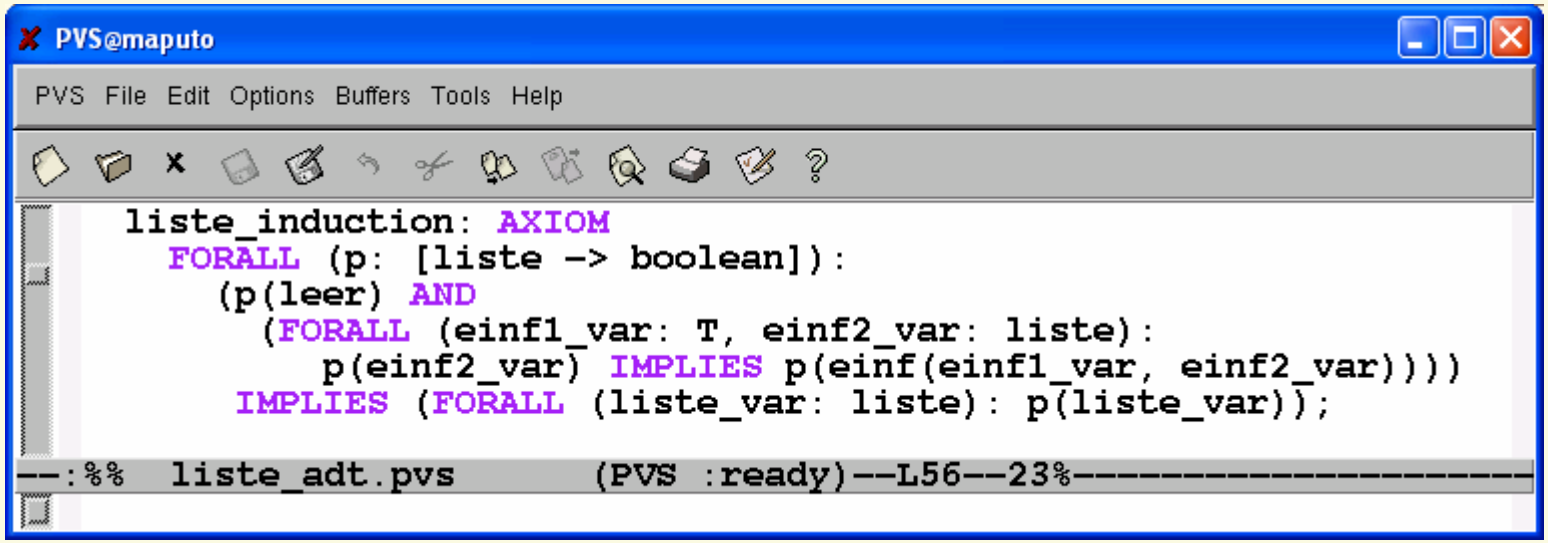
Induktionsbefehl

- Der Befehl
- `( induct … )`
- erkennt anhand des Typs seines Arguments, welches Induktionsaxiom benötigt wird.
- Besonders effizient ist der Befehl
- `( induct-and-simplify … )`
- **Beispiel:**

```
    |---
[1] FORALL (l:list[T]): app(l,empty) = l
  (induct-and-simplify "l")
```

liste_adt - Induktionsaxiom

- Auch das Induktionsaxiom wird automatisch erzeugt



liste_adt - Iteratoren

PVS@maputo

PVS File Edit Options Buffers Tools Help

```
every(p: PRED[T], a: liste): boolean =
    CASES a
      OF leer: TRUE,
         einf(einf1_var, einf2_var): p(einf1_var) AND every(p, einf2_var)
      ENDCASES;

some(p: PRED[T])(a: liste): boolean =
    CASES a
      OF leer: FALSE,
         einf(einf1_var, einf2_var): p(einf1_var) OR some(p)(einf2_var)
      ENDCASES;

some(p: PRED[T], a: liste): boolean =
    CASES a
      OF leer: FALSE,
         einf(einf1_var, einf2_var): p(einf1_var) OR some(p, einf2_var)
      ENDCASES;

subterm(x, y: liste): boolean =
    x = y OR
     CASES y
       OF leer: FALSE, einf(einf1_var, einf2_var): subterm(x, einf2_var)
       ENDCASES;
```

--:%% liste_adt.pvs   (PVS :ready)--L77--30%----------

Bäume mit variabel vielen Söhnen

- Bäume mit
  - einer beliebigen Anzahl von Söhnen
  - mit Information vom Typ T in den Blättern

```
busch [T : Type] : DATATYPE
BEGIN
   blatt(inhalt:T):istBlatt
   knoten(soehne:list[busch]):hatSoehne
END busch
```

Pattern und CASES-Ausdruck

- Jeder abstrakte Datentyp definiert Muster (pattern) mit denen man auf Teile verweisen kann
  - Muster sind Konstruktoren mit Variablen
- Mit einem CASES-Ausdruck kann man
  - entsprechend dem Aufbau eine Fallunterscheidung machen
  - unmittelbar Teile der Datenelemente referenzieren
- ```
  CASES x OF
     blatt(i)   : ergebnisVonInhalt(i)
     knoten(l)  : ergebnisVonSohnliste(l)
  ```
- Beispielsweise könnte man die Tiefe eines Baumes definieren als

```
tiefe(b:baum) : RECURSIVE nat =
   CASES baum OF
    blatt(i)   : 0
     knoten(l)   : maximum(map(tiefe,l))
```
```