

## 7. Abstrakte Datentypen und Strukturelle Induktion

Abstrakte Datentypen (ADTs) sind ein wichtiges Hilfsmittel zur Modellierung (Spezifikation) von Strukturen, Operationen auf Strukturen und Eigenschaften von Strukturen.

- abstrahierend von konkreter Realisierung einer Struktur (z.B. Zeigern)
- wichtig insbesondere für *induktiv definierte* Strukturen
- Die (abstrakten) Eigenschaften der Datentypen haben vorwiegend *algebraischen* Charakter, z.B. lassen sie sich häufig in Form algebraischer Gleichungen ausdrücken.
- In PVS durch spezielle syntaktisches Konstrukt mit weitreichenden Konsequenzen unterstützt.

# Lineare Listen

Abstrakte Datentypen in PVS werden beispielhaft dargestellt mit der Struktur der *linearen Listen*.

```
list [T: TYPE]: DATATYPE
  BEGIN
    null: null?
    cons (car: T, cdr:list): cons?
  END list
```

Spezielles DATATYPE-Konstrukt für die Deklaration abstrakter Datentypen – kann separat analog dem THEORY-Konstrukt oder innerhalb einer Theorie verwendet werden.

Parametrisierung wie bei Theorien

## Lineare Listen (2)

1. Ein neuer Typ `list` von linearen Listen mit Elementen aus `T` wird definiert.
2. Angabe einer Liste von Alternativen für Elemente des Datentyps durch Angabe je eines Konstruktor-Terms und eines zugehörigen Prädikats:
  - Konstante `null` zusammen mit Prädikat `null?`
  - Konstruktor `cons` zusammen mit Prädikat `cons?`  
`cons` erfordert Argumente vom Typ `T` bzw. `list`

Die Prädikate `null?` und `cons?` sind die charakteristischen Prädikate der Bildbereiche der entsprechenden Konstruktoren.

Auch bezeichnet als *Klassifikatoren* (engl. *recognizers*) für die Konstruktoren. Die Prädikate definieren jeweils einen Untertyp von `list`.

Daher formal:

```
null?, cons?: [list -> boolean]
```

```
null: (null?)
```

```
cons: [T,list -> (cons?)]
```

## Lineare Listen (3)

3. Die Konstante `null` und der Konstruktor `cons` sind die *Generatoren* des Datentyps; sie erzeugen zusammen alle Elemente von `list`.

Die zugehörigen Prädikate ergeben eine vollständige, disjunkte Zerlegung des Datentyps.

In der Literatur über algebraische Spezifikationen abstrakter Datentypen werden diese Eigenschaften auch durch *“no junk”* und *“no confusion”* beschrieben.

```
list_inclusive: AXIOM
  FORALL (l: list): null?(l) OR cons?(l)
```

```
list_disjoint: AXIOM
  FORALL (l: list): NOT( null?(l) AND cons?(l))
```

## Lineare Listen (4)

4. `car` und `cdr` sind *Selektoren* oder *Akzessor-Funktionen* für den Zugriff auf die Komponenten einer mit `cons` gebildeten Liste (die Argumente von `cons`). Sie sind nur für nicht-leere, d.h. mit `cons` gebildete Listen definiert:

`car: [(cons?) -> T]`

`cdr: [(cons?) -> list]`

Die Beziehungen zwischen Konstruktor und Selektoren werden ausgedrückt durch die algebraischen Axiome:

`list_car: AXIOM FORALL (a:T, l:list): car(cons(a,l)) = a`

`list_cdr: AXIOM FORALL (a:T, l:list): cdr(cons(a,l)) = l`

## Lineare Listen (5)

5. “Extensionalität” der Generatoren – ausgedrückt durch die Axiome:

```
list_null: AXIOM  FORALL (l: (null?): l=null
```

```
list_cons: AXIOM  FORALL (l1, l2: (cons?):  
    (car(l1) = car(l2) AND cdr(l1) = cdr(l2) IMPLIES l1 = l2)
```

Bemerkung: Quantifizierung über einen Untertyp ist im wesentlichen gleichbedeutend Quantifizierung über den Obertyp mit bedingter Aussage, z.B.

```
list_null:  AXIOM FORALL (l:list):  null?(l) IMPLIES l=null
```

## Lineare Listen (6)

*Fallunterscheidung* – CASES-Konstrukt:

anstatt einer IF-THEN-ELSE-Kaskade kann ein CASES-Ausdruck verwendet werden. Generatoren (Konstruktoren) des ADT werden als Unterscheidungsmuster (engl. *patterns*) der Fallunterscheidung benutzt.

am Beispiel der linearen Listen:

```
CASES x OF
  null:      expr1
  cons(a,l): expr2
ENDCASES
```

ist äquivalent zu

```
IF null?(x) THEN expr1
  ELSE LET (a,l) = (car(x),cdr(x)) IN expr2
ENDIF
```

Typüberprüfung stellt sicher, daß alle Fälle des Datentyps abgedeckt sind. Hierzu kann eine ELSE-Klausel helfen.

# Funktionen über linearen Listen

Das CASES-Konstrukt ist besonders nützlich bei der Definition von Funktionen über linearen Listen, die dem induktiven Aufbau der Listen folgen:

```
length(l:list): RECURSIVE nat =  
  CASES 1 OF  
    null: 0,  
    cons(x, y): length(y) + 1  
  ENDCASES  
  MEASURE ...
```

```
member(x:T, l:list): RECURSIVE bool =  
  CASES 1 OF  
    null: FALSE,  
    cons(hd, tl): x = hd  
                  OR member(x, tl)  
  ENDCASES  
  MEASURE length(l)
```

```
append(l1, l2): RECURSIVE list =  
  CASES 1 OF  
    null: l2,  
    cons(x, y): cons(x, append(y,l2))  
  ENDCASES  
  MEASURE length(l1)
```

```
reverse(l:list): RECURSIVE list =  
  CASES 1 OF  
    null: l,  
    cons(x,y): append(reverse(y),  
                       cons(x, null))  
  ENDCASES  
  MEASURE length
```

## Funktionen über linearen Listen (2)

Definition von Unterstruktur-Relationen:

```
subterm(x, y: list): boolean =  
  x = y OR  
  CASES y  
    OF null: FALSE,    cons(a,l): subterm(x, l)  
  ENDCASES;
```

```
<< : [list, list -> boolean] =  
  LAMBDA (x, y: list):  
    CASES y  
      OF null: FALSE,    cons(a,l): x = l OR x << l  
    ENDCASES;
```

subterm ist die reflexive Erweiterung der strikten Unterlisten-Relation <<.

Diese und weitere Funktionen sind im PVS-Prelude vordefiniert.

## Funktionen über linearen Listen (3)

Die gegebene Definition der Funktion `reverse` ist extrem ineffizient.  
Alternative Definition:

```
reva(l1, l2: list): RECURSIVE list =  
  CASES l1 OF  
    null: l2,  
    cons(x, y): reva(y, cons(x, l2))  
  ENDCASES  
  MEASURE length
```

```
rev2(l: list): list = reva(l,null)
```

Hier wird ein zweites Argument als 'Akkumulator' für das Ergebnis benutzt und als weiteres Argument in der Rekursion weitergereicht.

↪ 'Endrekursion' (engl. *tail recursion*), äquivalent zu einer Schleife

# Lineare Listen: Induktion

*Induktion* über linearen Listen:

```
list_induction: AXIOM
  FORALL (p: [list -> boolean]):
    (p(null) AND
     (FORALL (a: T, l1: list): p(l1) IMPLIES p(cons(a,l1))) )
    IMPLIES (FORALL (l: list): p(l))
```

Die Induktionsformel ist analog zu der für Induktion über natürlichen Zahlen aufgebaut.

Im Prinzip wird ein Induktionsbeweis angesetzt durch Instanziierung der Prädikat-Variablen  $p$  – aber: PVS-System kümmert sich um Details.

# Lineare Listen: Induktionsbeispiel

`x,y,z: VAR list[T]`

`appassoc: LEMMA`

`append(append(x,y),z) = append(x, append(y,z))`

Im Beweiser:

|-----

{1} FORALL (x, y, z: list[T]):

append(append(x, y), z) = append(x, append(y, z))

Frage: Welche unter den 3 Variablen ist das geeignetste Induktionsargument?

## Lineare Listen: Induktionsbeispiel (2)

Das Kommando (`induct "x"`) führt zur Generierung der 2 Unterziele:

```
appassoc.1 :
```

```
|-----
```

```
{1}  FORALL (y, z: list[T]):  
      append(append(null, y), z) = append(null, append(y, z))
```

```
appassoc.2 :
```

```
|-----
```

```
{1}  FORALL (cons1_var: T, cons2_var: list[T]):  
      (FORALL (y, z: list[T]):  
        append(append(cons2_var, y), z) =  
          append(cons2_var, append(y, z)))  
      IMPLIES  
      (FORALL (y, z: list[T]):  
        append(append(cons(cons1_var, cons2_var), y), z) =  
          append(cons(cons1_var, cons2_var), append(y, z)))
```

## Lineare Listen: Induktionsbeispiel (3)

Weitere Beweisschritte:

- Auffalten der Definition von `append`  
Welche Vorkommen?
- Vereinfachung
- Für den Induktionsschritt: Versuch, die Zielformel so umzuformen, dass die Induktionsvoraussetzung zur Umformung der Zielformel genutzt werden kann.  
Faustregel: Induktionsvoraussetzung *muß* genutzt werden.

In einfachen Fällen (wie im Beispiel) reicht dies aus für einen vollständigen Beweis.  
In allgemeineren Situationen kann es notwendig sein,

- eine weitere Induktion anzusetzen,
- die zu beweisende Formel zu verallgemeinern, damit die Induktionsvoraussetzung “stark genug” ist.

# ADT-Dateien

Eine DATATYPE-Definition wie die für `list` wird von PVS umgesetzt in eine Theorie, die alle Elemente (Deklarationen, Axiome usw.) enthält, die hier (zum Teil in veränderter Form) aufgeführt wurden, und einige mehr.

Zusätzliche Elemente u.a.:

- Iteratoren über Listen, insbesondere rekursive Äquivalente für Quantifizierung über die Elemente einer Liste
- Map-Funktionale

Die genaue Theorie ist in der Datei `list_adt.pvs` wiedergegeben,

Bei Benutzung des Listentyp muß jeweils ein konkreter Elementtyp (aktueller Parameter für `T`) angegeben werden (vgl. Theorie-Parameter – wird noch näher ausgeführt).

# Binäre Bäume

Binärbäume als weiteres Beispiel für abstrakte Datentypen in PVS mit in Knoten gespeicherten Werten vom Typ T:

```
BinTree [T: TYPE]: DATATYPE
  BEGIN
    empty: empty?
    node (key: T, left:BinTree, right:BinTree): node?
  END BinTree
```

Definition ist völlig analog zu der für lineare Listen.

Unterschiede:

- 2 anstatt einer induktiven Komponente beim Aufbau der Struktur.
- Entsprechend gibt es bei der Definition rekursiver Funktionen über Binärbäumen 2 Rekursionsargumente – für jedes muss gezeigt werden, dass das Argument beim rekursiven Aufruf 'kleiner' ist.

# Parametrisierung von Abstrakten Datentypen

Eine DATATYPE-Definition kann als separates Theorie-Äquivalent auf der obersten Ebene (d.h. *nicht* innerhalb einer Theorie) gegeben werden oder innerhalb einer Theorie.

Eine separate DATATYPE-Definition kann genauso wie eine Theorie parametrisiert sein.

Eine DATATYPE-Definition *innerhalb* einer Theorie ist “automatisch” parametrisiert mit den Parametern der Theorie, in die sie eingebettet ist.

Instanziierung von ADTs: wie Instanziierung von Theorien – s. später