

## Vollständigkeit

- Definition **Konstruktormenge** := miniale Menge von Operationen, mit denen man alle Elemente (=Instanzen) des ADT konstruieren kann.
- Für Stack ist das die Menge **{create, push}**
  - Leeren Stack anlegen
  - Um Elemente einzufügen
- Definition **vollständig**:  
Eine algebraische Definition ist **vollständig**, wenn es für jeden Konstruktor **z** und jeden Nicht-Konstruktor **o** ein Axiom der „Bauart“  $o(z(\dots)) = \dots$  gibt.
- Für Stack bräuchte man sechs Axiome, weil
  - $z \in \{create, push\}$
  - $o \in \{top, pop, isEmpty\}$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 32

- Es fehlen die Axiome:
  - $top(create()) = ?$
  - $pop(create()) = ?$
- Machen inhaltlich wenig Sinn, denn vom leeren Stack kann man weder ein oberstes Element erfragen noch es entfernen
- Offenbar sind die beiden Operationen top und pop **partiell** (d.h. nicht für alle Argumente definiert) (vergl. **partielle Funktionen**)
- Man kann dem Rechnung tragen und die Signatur ändern zu:
  - $pop : K \setminus \{create()\} \rightarrow K$
  - $top : K \setminus \{create()\} \rightarrow E$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 33

- Alternative: vervollständige partielle Operationen **top** und **pop**, indem man eine (hier nur einelementige) Menge **F** von Fehlerwerten ergänzt:
  - $F = \{stack\ underflow\}$
- dann die Signatur ändert zu:
  - $pop : K \rightarrow K \cup F$
  - $top : K \rightarrow E \cup F$
- sowie folgende Axiome hinzufügt:
  - (K5)  $top(create()) = stack\ underflow$
  - (K6)  $pop(create()) = stack\ underflow$
 (zieht evtl. aber einen „Rattenschwanz“ von Änderungen nach sich!)

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 34

## Klassifikation von Operationen

- Alternative Definition für **Konstruktor**:  
Prinzipiell jede Operation, die Wert (= Stack) aus Menge aller Stacks liefert. (bzw. generell Wert des ADT)
- Nach obiger Def. wäre auch **pop** ein "Konstruktor"; daher,
- Definition **Destruktor**:  
Operationen, die Datenmenge in einem ADT "verringern"
- Definition **Inspektor**: Alle Nicht-Konstruktoeren.
  - Bilden oft ADT auf Int oder Bool ab
  - Liefern meist Info über Status der jew. Instanz des ADT (z.B. leer?)
- Teilklassen von Inspektoren:
  - Prädikate**: liefern Bool'schen Wert
  - Selektor**: alle übrigen (z.B. **length()**)

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 35

## Modellbasierter Ansatz

- Beim **konstruktiven** (oder **modellbasierten**) Ansatz wird die Bedeutung eines ADT zurückgeführt („erklärt mit Hilfe von“) auf einen bereits bekannten ADT oder auf ein sonstiges Modell (z.B. aus der Mathematik), dessen Bedeutung als bereits bekannt gilt.
- Beispiel Stack: Ein bereits bekanntes Modell sind die Folgen über der Menge **E** der Elemente, d.h. **E\***, mit der Konkatenationsoperation „\*“. Die leere Folge wird bezeichnet durch „ε“.
- Im abstrakten Modell ist
  - $K := E^*$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 36

## Operationen im modellbasierten Ansatz

- die Operationen werden erklärt durch ( $\sigma \in E^*$ ):
  - $push(e, \sigma) \rightarrow e \cdot \sigma$  #→ "oben" im Stack = links i.d. Folge
  - $pop(e \cdot \sigma) \rightarrow \sigma$
  - $top(e \cdot \sigma) \rightarrow e$
  - $create() \rightarrow \epsilon$
  - $isEmpty(e \cdot \sigma) \rightarrow false$
  - $isEmpty(\epsilon) \rightarrow true$
- Die Operationen pop und top vervollständigt man ggfs durch:
  - $pop(\epsilon) \rightarrow stack\ underflow$
  - $top(\epsilon) \rightarrow stack\ underflow$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 37

Die Axiome des algebraischen Ansatzes lassen sich im Modell einfach „nachrechnen“:

```

top(push(e,k)) = top(e*k) = e
pop(push(e,k)) = pop(e*k) = k
isEmpty(create()) = isEmpty(ε) = true
isEmpty(push(e,k)) = isEmpty(e*k) = false

```

`isEmpty(pop(push(b,push(c,create())))) = ??`

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 38

### Formalisierung von „Liste“

- Sorten:
  - L Menge der Listen
  - E Menge der Elemente
  - B Menge der Wahrheitswerte
  - N Menge der natürlichen Zahlen
- Signatur:
  - new :  $\rightarrow L$
  - cons :  $E \times L \rightarrow L$
  - head :  $L \setminus \{new()\} \rightarrow E$
  - tail :  $L \setminus \{new()\} \rightarrow L$
  - isNew :  $L \rightarrow B$
  - length :  $L \rightarrow N$

*Konstruktoren* (new, cons)  
*Partiell* (head, tail)

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 39

### Axiome für Liste

- Ein vollständiger Satz von Axiomen für Listen:
  - (L1) `head(cons(e,l)) = e`
  - (L2) `tail(cons(e,l)) = l`
  - (L3) `isNew(new()) = true`
  - (L4) `isNew(cons(e,l)) = false`
  - (L5) `length(new()) = 0`
  - (L6) `length(cons(e,l)) = length(l)+1`
- Denn zwei der Nicht-Konstruktoren sind **partiell**:
  - $Z \in \{new, cons\}$
  - $O \in \{head, tail, isNew, length\}$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 40

- Beispiel:
 

```
tail([5,7,9]) = tail(cons(5, [7,9])) = [7,9]
```

*[5,7,9] ist abkürzende Schreibweise für cons(5, cons(7, cons(9, empty())))*
- Bemerkung: Hätten wir **cons** so definiert, daß **e hinten** angefügt wird, so hätten wir **Rekursion** in den Axiomen gebraucht
  - (L1) `head(cons(l,e)) = { e, head(l) ; 1 = new(), sonst`
  - (L2) `tail(cons(l,e)) = { new(), cons(tail(l),e) ; 1 = new(), sonst`
- Beispiel:
 

```
tail([5,7,9]) = tail(cons([5,7],9)) =
cons(tail([5,7]), 9) = cons(cons([5],7), 9) =
cons(cons(tail([5]), 7), 9) = ... = [7,9]
```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 41

### Beispiel: „Liste“ als Modell für „Stack“

- Rückführung:  $L \approx K, E \approx E$ 
  - `push(e,k) → cons(e,k)`
  - `pop(k) → tail(k)`
  - `top(k) → head(k)`
  - `create → new`
  - `isEmpty(k) → isNew(k)`
- Gelten die Stackaxiome?
- Wir rechnen (K1) nach:
 

```

top(push(e,k)) = top(cons(e,k))
                = head(cons(e,k))
                = e

```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 42

### Zusicherungen-basierte Spezifikation von ADTs

- Weiteres Stack-Modell
 
$$Stack = (S, c), \quad c \in \mathbb{N}, S \subseteq E \times \mathbb{N}$$
- Stack muß folgenden **Invarianten** genügen:
  - (I1)  $\forall (e_1, n_1) \in S : (n_1 = n_2) \Rightarrow (e_1 = e_2)$
  - (I2)  $\forall (e, n) \in S : c > n$
- Die Operationen werden jeweils durch **Vorbedingungen** (*pre-condition*) und **Nachbedingungen** (*post-condition*) spezifiziert, in denen  $(s, c)$  den Zustand des Modells **vor** Ausführung der Operation bezeichnet,  $(s', c')$  den Zustand **nach** der Ausführung.

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 43

```

push(e,k)   pre : true
            post : S' = {(e, c)} ∪ S ∧  $\overset{c'}{c} > c$ 
            wa(I2)

top(k)      pre : |S| > 0
            post : S' = S ∧
                  ∃(t, m) ∈ S : top(k) = t ∧
                  ∀(e, n) ∈ S : n ≤ m

pop(k)      pre : |S| > 0
            post : ∃(t, m) ∈ S ∀(e, n) ∈ S :
                  n ≤ m ∧ S' = S - {(t, m)}

create()    pre : true   post : S' = ∅

isEmpty(k)  pre : true
            post : S' = S ∧
                  isEmpty(k) = (|S| = 0)

```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 44

### Spezifikation von Funktionen

- Mit Pre- und Post-Conditions lassen sich auch einfache Funktionen (losgelöst von ADT / Klasse) formal spezifizieren:
  - Zulässigen Bereich der Parameter spezifizieren, für den Funktion korrekt arbeitet (arbeiten soll)
  - Prädikat angeben, das gültig sein muß, wenn Fkt korrekte Ausgabe produziert hat
  - Veränderungen der Parameter spezifizieren (falls Call-by-Reference)

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 45

### Beispiel

- Funktion `search` liefert ersten Index von Element in `A`, das gleich Key `k` ist

```

def search ( A, k )

pre : typ(A) = array
     typ(k) = int
     ∃i ∈ [0, len(A) - 1] : A[i] = k

post : typ(search) = int
      A[search(A, k)] = k

```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 46

### Fehlerbedingungen

- Ann.: Funktion ist korrekt, d.h., Nachbedingung gültig, wenn Vorbedingung erfüllt
- Leider: Nachbedingung *nur* dann gültig, wenn Vorbed. gültig!
- Also: auch spezifizieren, was passiert, wenn Vorbed. nicht erfüllt, d.h., Funktion erhält unerwartete Eingaben
- Beispiel `search` :

```

def search ( A, k )

pre : ∃i ∈ [0, len(A) - 1] : A[i] = k
post : A[search(A, k)] = k
      err : search(A, k) = -1

```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 47

- Weitere Alternative: Tupel mit (Index,Err-Code) zurückliefern

```

def search ( A, k )

pre : typ(A) = array
     typ(k) = int

post : typ(search) = (int, bool)
      (i, e) = search(A, k)
      ∃j ∈ [0, len(A) - 1] : A[j] = k ⇒ i = j ∧ e = false
      ∀j ∈ [0, len(A) - 1] : A[j] = k ⇒ e = true

```

- Weitere Alternative: Exception werfen (später in SW-Engineering)
  - Problem: läßt sich nicht mehr so einfach mit Prädikaten fassen

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 48

### Formalisierung von „Menge“

- Relevante Mengen („Sorten“):
  - `M` Sorte der Mengen
  - `E` Sorte der Elemente
  - `B` Sorte der Wahrheitswerte
- Signatur (Syntax der Operationen):
  - `empty` :  $\rightarrow M$
  - `insert` :  $E \times M \rightarrow M$
  - `delete` :  $E \times M \rightarrow M$
  - `isEmpty` :  $M \rightarrow B$
  - `contains` :  $E \times M \rightarrow B$
  - `union` :  $M \times M \rightarrow M$
  - `inter` :  $M \times M \rightarrow M$

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 52

▪ Axiome:

```
(M1) delete(e, empty()) = empty()
(M2) delete(e, insert(e,m)) = delete(e,m)
(M3) delete(e, insert(f,m)) =
      insert(f, delete(e,m)) für  $f \neq e$ 
(M4) isEmpty(empty()) = true
(M5) isEmpty(insert(e,m)) = false
(M6) contains(e, empty()) = false
(M7) contains(e, insert(e,m)) = true
(M8) contains(e, insert(f,m)) = contains(e,m)
      für  $f \neq e$ 
```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 53

▪ Axiome für Vereinigung und Durchschnitt von Mengen:

```
(M09) union(m, empty()) = m
(M10) union(m1, insert(e, m2)) =
      insert(e, union(m1, m2))
(M11) inter(m, empty()) = empty
(M12) inter(m1, insert(e, m2)) =
      insert(e, inter(m1, m2)),
      falls contains(e, m1)
(M13) inter(m1, insert(e, m2)) = inter(m1, m2),
      falls nicht contains(e, m1)
```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 54

▪ Was ist von folgenden "Axiomen" zu halten?

▪ Was besagen sie?

```
(M14) insert(e, insert(f,m)) =
      insert(f, insert(e,m))
(M15) insert(e, insert(e,m)) =
      insert(e,m)
```

▪ Sind sie notwendig / nützlich / schädlich?

▪ Position der Konstruktoren? Terminierung?

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 55

### Queue

▪ Erinnerung: Queue = Datenspeicher,

- der eine Folge von Werten aufnimmt,
- in dem man „am hinteren Ende“ Werte hinzufügen
- und „am vorderen Ende“ wieder entnehmen kann.

▪ Operationen: Anlegen („neu“), Einfügen („ein“), Entnehmen („vorn“, „aus“), feststellen, ob noch Elemente vorhanden sind („leer“)

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 56

### Formalisierung von Queue

▪ Signatur (Syntax der Operationen):

```
neu :  $\rightarrow W$ 
ein :  $E \times W \rightarrow W$ 
aus :  $W \setminus \{\text{neu}\} \rightarrow W$ 
vorn :  $W \setminus \{\text{neu}\} \rightarrow E$ 
leer :  $W \rightarrow B$ 
```

▪ Axiome für Queue:

```
(W1) vorn(ein(e,k)) =
      if k == neu then e else vorn(k)
(W2) aus(ein(e,k)) =
      if k == neu then k else ein(e, aus(k))
(W3) leer(neu) = true
(W4) leer(ein(e,k)) = false
```

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 57

### Weitere Beispiele

... In Kapitel 10 einer „Bibel des Software Engineering“:

Ian Sommerville:  
*Software Engineering*  
 Addison-Wesley (5. Auflage, 1995)

Spezifiziert werden dort:

- Arrays über einem beliebigen Elementtyp
- Suchbäume über einem geordneten Elementtyp
- Cursorpositionen auf dem Bildschirm

G. Zachmann Informatik 1 - WS 05/06 Abstrakte Datentypen 58