Vollständigkeit

- Definition Konstruktormenge := mininale Menge von Operationen, mit denen man alle Elemente (=Instanzen) des ADT konstruieren kann.
- Für Stack ist das die Menge `{create, push}`

  `create`: Leeren Stack anlegen — `push`: 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(...)) = ...` gibt.
- Für Stack bräuchte man sechs Axiome, weil

  `z` ∈ `{create,push}`

  `o` ∈ `{top,pop,isempty}`

- 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 \ {create()}` → `K`

  `top : K \ {create()}` → `E`

- 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` → `K` ∪ `F`

  `top : K` → `E` ∪ `F`

  sowie folgende Axiome hinzufügt:

  `(K5) top(create()) = stack underflow`

  `(K6) pop(create()) = stack underflow`

(zieht evtl. aber einen „Rattenschwanz" von Änderungen nach sich!)

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-Konstruktoren.
  - 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()`**)

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*
```


Operationen im modellbasierten Ansatz

- die Operationen werden erklärt durch ($\sigma \in$ `E*`) :

```
push(e,σ)     → e•σ      #→ "oben" im Stack = links i.d. Folge
pop(e•σ)      → σ
top(e•σ)      → e
create()      → ε
isEmpty(e•σ)  → false
isEmpty(ε)    → true
```

- Die Operationen pop und top vervollständigt man ggfs durch:

```
pop(ε)  → stack underflow
top(ε)  → stack underflow
```


- 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())) )) = ??
```


Formalisierung von „Liste"

- Sorten:
  - `L` Menge der Listen
  - `E` Menge der Elemente
  - `B` Menge der Wahrheitswerte
  - `N` Menge der natürlichen Zahlen
- **Signatur:**

```
new    : → L             } Konstruktoren
cons   : E × L → L       }
head   : L\{new()} → E   } Partiell
tail   : L\{new()} → L   }
isNew  : L → B
length : L → N
```

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 ∈ {new,cons}`

  `O ∈ {head,tail,isNew,length}`

- Beispiel:

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

  `[5,7,9]` ist abkürzende Schreibweise für `cons(5,[7,9]) = 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                  , l = new()
                       { head(l)            , sonst

(L2) tail(cons(l,e)) = { new()              , l = new()
                       { cons(tail(l),e)    , sonst
```

- Beispiel:

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


Beispiel: „Liste" als Modell für „ Stack "

- Rückführung: `L ≈ K,  E ≈ 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
```


Zusicherungen-basierte Spezifikation von ADTs

- Weiteres Stack-Modell

$$\text{Stack} = (S, c), \quad c \in \mathbb{N},\ S \subseteq E \times \mathbb{N}$$

- Stack muß folgenden Invarianten genügen:

$$(I1) \quad \forall (e_1, n_1) \in S : (n_1 = n_2) \Rightarrow (e_1 = e_2)$$
$$(I2) \quad \forall (e, n) \in S : c > n$$

- 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.

push(e,k) pre : $true$

post : $S' = \{(e,c)\} \cup S \;\wedge\; \underbrace{c' > c}_{\text{wg}(I2)}$

top(k) pre : $|S| > 0$

post : $S' = S \wedge$

$\exists (t,m) \in S : \text{top(k)} = t \wedge$

$\forall (e,n) \in S : n \leq m$

pop(k) pre : $|S| > 0$

post : $\exists (t,m) \in S \; \forall (e,n) \in S :$

$n \leq m \;\wedge\; S' = S - \{(t,m)\}$

create() pre : $true$ post : $S' = \varnothing$

isEmpty(k) pre : $true$

post : $S' = S \wedge$

$\text{isEmpty(k)} = (|S| = 0)$

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)

Beispiel

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

```
def search( A, k )
```

pre : $\text{typ}(A) = \text{array}$

$\text{typ}(k) = \text{int}$

$\exists i \in [0, \text{len}(A) - 1] : A[i] = k$

post : $\text{typ(search)} = \text{int}$

$A[\text{search}(A,k)] = k$

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 : $\exists i \in [0, \text{len}(A) - 1] : A[i] = k$

post : $A[\text{search}(A,k)] = k$

err : $\text{search}(A,k) = -1$

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

```
def search( A, k )
```

pre : $\text{typ}(A) = \text{array}$
$\text{typ}(k) = \text{int}$

post : $\text{typ}(\text{search}) = (\text{int}, \text{bool})$
$(i, e) = \text{search}(A, k)$
$\exists j \in [0, \text{len}(A) - 1] : A[j] = k \Rightarrow i = j \wedge e = \text{false}$
$\nexists j \in [0, \text{len}(A) - 1] : A[j] = k \Rightarrow e = \text{true}$

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

Formalisierung von „ Menge"

- Relevante Mengen („Sorten"):
  - `M` Sorte der Mengen
  - `E` Sorte der Elemente
  - `B` Sorte der Wahrheitswerte
- Signatur (Syntax der Operationen):

```
empty    : → M              } Konstruktoren
insert   : E × M → M        }
delete   : E × M → M
isEmpty  : M → B
contains : E × M → B
union    : M × M → M
inter    : M × M → M
```


- 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 ≠ 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 ≠ e
```


- 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)
```

- 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?

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“)

Formalisierung von Queue

- Signatur (Syntax der Operationen):

  neu : $\rightarrow W$
  ein : $E \times W \rightarrow W$
  aus : $W \backslash \{\text{neu}\} \rightarrow W$
  vorn : $W \backslash \{\text{neu}\} \rightarrow E$
  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
```


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