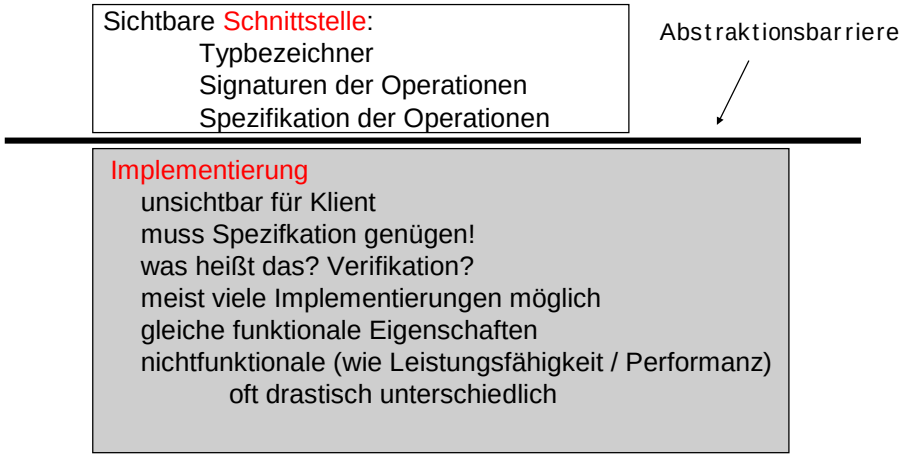
Abstrakte Datentypen (ADT)

2.3 Spezifikation von Abstrakten Datentypen



Spezifikation von ADT

- Grundsätze
  - Keine Annahmen über Implementierung des Typs
  - Methodensignaturen (Prozedurköpfe) festlegen
  - Semantik der Methoden festlegen
- Zwei Verfahren
  - Modellierende Spezifikation
    - Modell zugrunde legen, Voraussetzungen, zu erzielende Effekte, Invarianten darin spezifizieren
    - Ähnlich wie bisher: Modell war Zustandsraum, allerdings implementierungsabhängig!
    - Besser: Natürliche Zahlen, Mengen, Listen, ...
    - Wichtig: Spezifizierende und Implementierer müssen gleiches Verständnis des Modells haben ("gleiche Semantik")
  - Algebraische Spezifikation

Algebraische Spezifikation

2.3.1 Algebraische Spezifikation

ADT wird festgelegt durch:

Wertemengen (Typen, Sorten) Syntax
Signaturen der Operationen

Definierende Gleichungen Semantik

legen Beziehungen zwischen Operationen fest

Stack (algebraisch)

types
Stack, T, Bool , Int // repräsentiert durch Typen

operators
createStack : -> Stack
push: T X Stack -> Stack
pop : Stack -> Stack
top : Stack -> T
size : Stack -> Int

axioms ....

Stack (algebraisch)

axioms

for all x ∈ T, s ∈ Stack :

pop (push (x , s)) = s

top (push (x , s)) = x

size (createStack) = 0

size (push (x , s)) = 1 + size (s)

Einzige Möglichkeit zur Charakterisierung von Stack-Objekten sind Axiome

Stack (algebraisch)

Kanonische Darstellungen

Viele Darstellungen repräsentieren den gleichen Wert

push(y (pop (push (x , createStack)))) =
push (y, createStack)

Axiome als Ersetzungsregeln (rewrite rules)
Systematische Ersetzung -> kanonische Darstellung

Stack (algebraisch): Haskell

- Signatur

```
-- EmptyStack ::                  Stack t
-- Push       :: t -> Stack t -> Stack t
   pop        ::      Stack t -> Stack t
   top        ::      Stack t -> t
   size       ::      Stack t -> Int
```


Stack (algebraisch): Haskell

```
data Stack t = EmptyStack | Push t (Stack t)

pop EmptyStack = error “stack underflow“
pop(Push x s)  = s

top EmptyStack = error “no top“
top(Push x s)  = x

size EmptyStack = 0
size(Push x s)  = 1 + size s
```

Stack (algebraisch): Haskell

```
Main> Push 4(pop(Push 3(Push 1 EmptyStack)))
      Push 4 (Push 1 EmptyStack)
```

Kanonische Form

```
Main> Main> (top s, size s) where s =Push 4(pop(pop(Push
                                3(Push 1(Push 0 EmptyStack)))))
(4,2)
```

Haskell :

- Ersetzungsregeln automatisch anwenden (Reduktion auf Normalform !)
- Implementierung berücksichtigt Fehlersituationen
  Sollte Spezifikation auch!
  top, pop sind partielle Funktionen

Voraussetzungen und Effekte gehören auch hier zu den Signaturen

Algebraische Spezifikation: Mengen

```
-- Types
-- Set t,  Bool

-- Signature

-- createS :: Set t
   isEmpty :: Set t -> Bool
-- Ins     :: Ord t => t -> Set t -> Set t  -- insert
   isIn    :: Ord t => t -> Set t -> Bool
   delete  :: Ord t => t -> Set t -> Set t

   data Set t = ESet | Ins t (Set t)
              deriving (Show)
```

Algebraische Spezifikation: Mengen

```
-- axioms
   createS = Eset

   isEmpty ESet      = True
   isEmpty (Ins x s) = False

   isIn x ESet = False
   isIn  j (Ins i s)
        | i==j       = True
        | otherwise  = isIn j s
```


Algebraische Spezifikation von Mengen

- Gleichheit und Ungleichheit

contained s1 s2
  = ∀ x (isIn x s1 => isIn x s2)

eqS s1 s2 = contained s1 s2 && contained s2 s1

```
Main> eqS (Ins 3 (Ins 2 (Ins 3 ESet)))
          (Ins 2 (Ins 3 ESet))
True
```

Algebraische Spezifikation von Mengen

```
contained :: Ord t => Set t -> Set t -> Bool
contained ESet s   = True
contained (Ins x s1) s2
          = isIn x s2 && (contained s1 s2)

eqSet :: Ord t => Set t -> Set t -> Bool
eqS s1 s2
  = (contained s1 s2) && (contained s2 s1)
```

Bisher keinen Gebrauch von Darstellung gemacht:

```
Ins 3 (Ins 2 (Ins 3 ESet))) == (Ins 2 (Ins 3 ESet))
```


Algebraische Spezifikation von Mengen: Probleme

- Gleichheit und Ungleichheit?
  - delete x (Ins x set) = set
    als definierende Gleichung nicht ausreichend
  - delete – Definition

```
delete x ESet = ESet
   delete x (Ins y s)
    | x == y  = delete x s
    | otherwise    = Ins y (delete x s)
```

    Greift auf Repräsentation der Menge zurück
  - Allgemein: definierende Gleichungen für = und != technisch schwierig und manchmal verwirrend

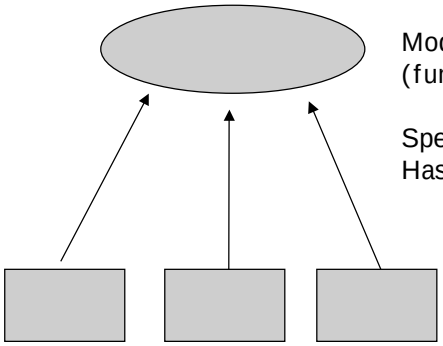
Algebraische Spezifikation

- Probleme im Zusammenhang mit Algebraischer Spezifikation
  - Welche Gleichungen definieren?
  - Wie findet man überhaupt welche?
  - Vollständigkeit?
    - Genügend Gleichungen, um Operationen zu spezifizieren?
  - Widerspruchsfreiheit der definierenden Gleichungen? Redundanz?

-> Praktische Relevanz gering trotz "vieler Bücher"

2.3.2 Modellierende Spezifikation

- Modell und Implementierung(en)



Modell: z.B. math. Mengen, Folgen, auch (funktionale) Programmiersprache

Spezifikation: Prädikatenlogik, Haskell-(Pseudo)code

- Viele mögliche Implementierungen.
- Korrekte Implementierung muss Spezifikation auf Modellebene erfüllen

Modellierende Spezifikation: Stack

- Spezifikation von Stack mit verschiedenen Modellen

Typen:

Stack, T, int

Operatoren:

createStack : -> Stack

push: T X Stack -> Stack

pop : Stack -> Stack

top : Stack -> T

size : Stack -> Int

Modellierende Spezifikation mit Folgen (Stack)

Modell

$s = e \mid s = (x_i)_{i=1..n}$, type x_i = T, n <= N -- längenbeschränkte -- Folgen

Semantik

createStack = e

push x e = (x_1)

push x $(x_i)_{i=1..k}$

// requires: k < N

// effect: push x $(x_i)_{i=1..k}$ = $(x_i)_{i=1..k+1}$ and x_{k+1} = x

pop s

// requires:: s != e // andere Schreibweise als push

// effect: returns e , if k=1

returns $(x_i)_{i=1..k-1}$, if k > 1

where s = $(x_i)_{i=1..k}$

top...

size ...

Hier wird Gebrauch von der "Modellrepräsentation" gemacht

Modellierende Spezifikation mit Mengen (Stack)

Modell

s = {(int,x) | type x = T} -- Menge von Paaren
-- (Einfügefolge, Wert)

... ganz ähnlich wie Folgen

- Etwas exotisches Modell für Folgen
- Aber z.B. Prioritätsschlangen?

Modellierende Spezifikation mit Haskell' (Stack)

```
interface Stack[T] {// model:  [T]
int N = 20;         // inv:    length < N = 20
void init();        // requires:    True
                    // effect:  returns []
void push(T x) throws Overflow;
                     // requires:    length s < max
                     //            -- else Overflow
                     // effects:   s == [x] ++ S
                     //          if self == S
void pop () ...
T top() throws Underflow;
                     // requires:    not([])
                     //           -- else Underflow
                     // effects:
                     // returns x,  if  self == x:s
int size...
}
```

Modellierende Spezifikation

- Finde geeignetes Modell für zu spezifizierenden Datentyp

  Beispiel: Supermarktkassen

  Modell: SK= {(KassenNr, besetzt, bestand(bar, EC))}

  Inv: 0 < KassenNr <= N, bar>=0, EC>=0
- Definiere Operationen

  besetzen : Kassennr -> Kasse

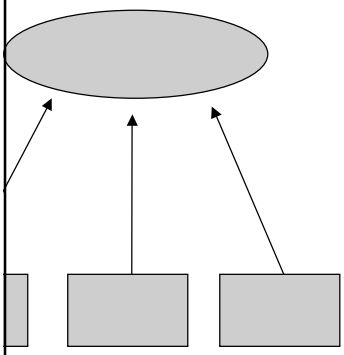
  zahlenBar: .....
- Definiere Semantik

  bezogen auf die abstrakte Modellrepräsentation

  besetzen i = (i, True, (b1,b2))

  //requires: 0 < i < N , (i, False, (b1,b2)) ∈ SK

Spezifikation und Implementierung



Wie findet man Implementierung?

- Kreativ mit informatischem Handwerkszeug umgehen
- Programm-Bibliotheken nutzen

Korrekte Implementierung? (s.u)

- Beziehung zwischen Modell-spezifikation und Imple-mentierung herstellen
- Korrektheit zeigen

Beste Implementierung?

- Nichtfunktionale Eigen-schaften untersuchen -> Datenstrukturen ! (später)