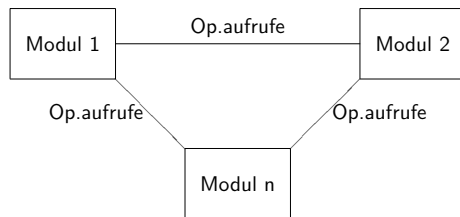


Beherrschung der Komplexität großer Systeme: MODULARISIERUNG

(I) Abstrakte Datentypen und (II) Algebraische Spezifikation



G. Görz, FAU, Inf.8

6-ADT-1

- Bekannt ist das *Was*: Syntax, Semantik der Operationsaufrufe
- Verborgen ist das *Wie*: Implementation der Operationsaufrufe
- Modul = Datenstruktur + Zugriffsfunktionen (Liskov/Zilles 1972)

G. Görz, FAU, Inf.8

6-ADT-2

Modultypen

- *Funktionsmodul*
 - beschreibt eine Funktion
 - keine dauerhaften Daten
- *Datenmodul*
 - trennt die Implementation einer Datenstruktur vom Rest des Programmes
 - umfaßt die Beschreibung der Datenstruktur und der Zugriffsfunktionen
 - Aufbau: Schnittstelle und Implementierung
 - Änderungen an der Datenstruktur möglich, ohne den Rest des Programmes zu beeinflussen
 - Beispiel: Verwaltung einer Datei

G. Görz, FAU, Inf.8

6-ADT-3

- Datentyp (Objektart)
 - Mechanismus zur Konstruktion von Datenmoduln
 - mehrere Exemplare (Instanzen) können gleichzeitig existieren
 - Beispiele: Puffer, Stapel, Baum usw.
- Typschablone ("generic module"): Parametrisierter Modul
- Syntax: programmiersprachlich präzisierbar;
Semantik: Axiomensystem!

G. Görz, FAU, Inf.8

6-ADT-4

Beispiel eines Axiomensystems: Natürliche Zahlen nach Peano

- Der Datentyp CARDINAL (nach Floyd und Hoare)
 - (1) $x + y = y + x$
 - (2) $x * y = y * x$
 - (3) $(x + y) + z = x + (y + z)$
 - (4) $(x * y) * z = x * (y * z)$
 - (5) $x * (y + z) = (x * y) + (x * z)$
 - (6) $y \leq x \Rightarrow (x - y) + y = x$
 - (7) $x + 0 = x$
 - (8) $x * 0 = 0$
 - (9) $x * 1 = x$
- Dafür gibt es mehrere Modelle (s.u.) !
- Wer nur mit (1)–(9) arbeitet, ist von der Implementation unabhängig

G. Görz, FAU, Inf.8

6-ADT-5

Algebraische Sicht der Datenabstraktion

Isomorphie von Algebren:

- eineindeutige Beziehungen zwischen Trägermengen (verträglich bezüglich der Struktur)
- ⇒ eineindeutige Beziehung zwischen Strukturen
- Eigenschaften entsprechen einander

Isomorphie ist die Äquivalenzrelation der Datenabstraktion:

- entstehendes Objekt ist unabhängig von
 - Trägermengen der Algebren
 - Namen der Operatoren

G. Görz, FAU, Inf.8

6-ADT-7

Algebraische Sicht der Datenabstraktion

ALGEBRA: "Signatur" Σ

- Trägermenge mögliche Datenobjekte (unter gegebener Repräsentation)
- Operationen: primitive Funktionen
 - Anzahl der Operatoren
 - Eigenschaften der Operatoren ("Gesetze" \mathcal{E})
 - * Stelligkeit
 - * Typen der Argumente und Werte
 - * Gleichungen zwischen Operatoren

Das Paar (Σ, \mathcal{E}) bezeichnet einen *abstrakten Typ*

G. Görz, FAU, Inf.8

6-ADT-6

wichtig: nur Eigenschaften der Operatoren

- abstrakte Datenstruktur ist implementations-unabhängig
- spezielle Repräsentation mittels vorgegebener Datenstrukturen spielt keine Rolle
- *wichtig*: nur die primitiven Funktionen

G. Görz, FAU, Inf.8

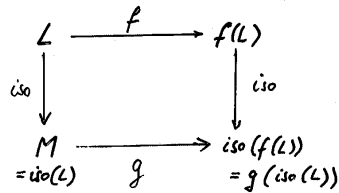
6-ADT-8

Isomorphie von Algebren

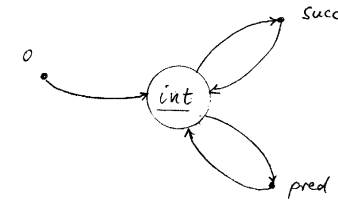
als Äquivalenzrelation der Datenabstraktion

$$(L, f) \xleftrightarrow{iso} (M, g)$$

$$iso(f(x)) = g(iso(x))$$



Signaturdiagramm:



Gesetze

$$INV := \begin{cases} succ(pred(x)) = x \\ pred(succ(x)) = x \end{cases}$$

Daraus Eindeutigkeit der Operationen

ableitbar:

$$INJ := \begin{cases} succ(x) = succ(y) \Rightarrow x = y \\ pred(x) = pred(y) \Rightarrow x = y \end{cases}$$

Die ganzen Zahlen als abstrakter Typ

Aus der Null kann man mithilfe der Nachfolger- und Vorgängeroperation

- jede ganze Zahl in endlich vielen Schritten gewinnen;
- Addition, Subtraktion und Multiplikation rekursiv ausdrücken

Signatur (int; 0, succ, pred) mit

$$\begin{aligned}
 0: & \rightarrow \underline{\text{int}} \\
 succ: & \underline{\text{int}} \rightarrow \underline{\text{int}} \\
 pred: & \underline{\text{int}} \rightarrow \underline{\text{int}}
 \end{aligned}$$

Die ganzen Zahlen als abstrakter Typ (2)

Gibt es neben den "normalen" ganzen Zahlen noch weitere Modelle dieses Typs?

Exkurs: Die Äquivalenzrelation "Kongruenz modulo n"

$$a \equiv b \pmod{n}$$

b ist Rest bei Division von a durch n, d.h. $a = q \cdot n + b$, $0 \leq b < n$

Damit gilt: $n \equiv 0 \pmod{n}$

Bemerkung: Kongruenz mod n gilt auch für $n < 0$:

$$-3 \equiv 2 \pmod{5}, \text{ denn } -3 = -1 \cdot 5 + 2$$

$$a \equiv b \pmod{n}, \text{ falls } a - b \equiv 0 \pmod{n},$$

d.h. $a - b$ ist durch n ohne Rest teilbar,

z.B. $10 \equiv 5 \pmod{5}$, denn $10 - 5 \equiv 0 \pmod{5}$, $11 \equiv 1 \pmod{5}$

Die ganzen Zahlen als abstrakter Typ (3)

Andere Schreibweise: MOD- $n(a,b)$ als zweistellige Relation über den ganzen Zahlen,
z.B. MOD-5(10,5) für $10 \equiv 5 \pmod{5}$

MOD- n ist eine Äquivalenzrelation

- Reflexivität:
 $\text{MOD-}n(a, a) \leftrightarrow a-a$ durch n ohne Rest teilbar
 $\leftrightarrow 0$ durch n ohne Rest teilbar
- Symmetrie:
 $\text{MOD-}n(a, b) \leftrightarrow a-b$ durch n ohne Rest teilbar
 $\leftrightarrow b-a$ durch n ohne Rest teilbar
 $\leftrightarrow \text{MOD-}n(b, a)$
 $(a - b = q \cdot n \leftrightarrow b - a = -q \cdot n)$

Die ganzen Zahlen als abstrakter Typ (4)

Zurück zur obigen Frage: Gibt es neben den ganzen Zahlen noch weitere Modelle des abstrakten Typs `int`?

Ja: Sei N positive ganze Zahl.
Die *Kongruenzklassen modulo N* bilden ein (aus N Elementen bestehendes) Modell dieses Typs

Beispiel: Sei $N=3$

Bezeichne $[i]$ die Klasse der durch 3 mit dem Rest i teilbaren ganzen Zahlen und sei

$\text{succ}([0]) = [1], \text{succ}([1]) = [2], \text{succ}([2]) = [0]$
 $\text{pred}([0]) = [2], \text{pred}([1]) = [0], \text{pred}([2]) = [1]$

Dann bildet $\{[0], [1], [2]\}$ ein Modell

(II) Axiomatische Spezifikation

- Transitivität:
 $\text{MOD-}n(a, b)$ und $\text{MOD-}n(b, c)$
 $\leftrightarrow a - b$ und $b - c$ durch n ohne Rest teilbar
 $\leftrightarrow a - b = q_1 n \wedge b - c = q_2 n$
d.h. $(a - b) + (b - c) = q_1 n + q_2 n$
d.h. $a - c = (q_1 + q_2) n$
 $\leftrightarrow a - c$ durch n ohne Rest teilbar
 $\leftrightarrow \text{MOD-}n(a, c)$

Kongruenz \pmod{n} zerlegt die ganzen Zahlen in Äquivalenzklassen:

z.B. MOD-5:
 $\{0, -5, 5, -10, 10, \dots\}$
 $\{1, -4, 6, -9, 11, \dots\}$
 $\{2, -3, 7, -8, 12, \dots\}$
 $\{3, -2, 8, -7, 13, \dots\}$
 $\{4, -1, 9, -6, 14, \dots\}$

- Stelle Beziehungen zwischen den Operationen her
- ZIEL: Widerspruchsfreiheit, Vollständigkeit
- Parnas (1974): Unterscheidung von *V- und O-Funktionen*
- V-Funktionen (value returning) sagen etwas über die Objekte aus
- O-Funktionen (operate) verändern bzw. konstruieren Objekte (sind nicht beobachtbar!)
- Eine Spezifikation ist *hinreichend vollständig*, wenn man von jeder O-Funktion weiss, wie sie auf den nachfolgenden Aufruf einer V-Funktion wirkt.

- Vorgehensweise nach Parnas:
 - Identifiziere O- und V-Funktionen.
 - Definiere ein „Axiom“ für jedes Paar einer O- und einer V-Funktion.
 - Implementiere die Funktionen, und beweise, dass die Implementation die Axiome erfüllt.
- Anwendung auf Datenmoduln: Verändertes Objekt ist verborgen, also nicht Parameter.

Axiomatische Spezifikation (2)

- Modifizierte Vorgehensweise
 - Identifiziere V- und O-Funktionen
 - Identifiziere unter den O-Funktionen die Konstruktoren: Jedes Element läßt sich unter ausschließlicher Verwendung der Konstruktoren gewinnen (funktioniert fast immer).
 - Definiere “Axiome” für jedes Paar
 - * Konstruktor / V-Funktion
 - * Konstruktor / sonstige O-Funktion
 - Implementiere die Funktionen, und beweise, dass die Implementation die Axiome erfüllt.

- Beispiel: Natürliche Zahlen
 - Konstruktoren: *zero, succ*
 - Sonstige O-Funktionen: *plus, times*
 - V-Funktionen: *less, odd*

Beispiel: Warteschlange

- Syntax

<i>init</i> :		→	<i>queue</i>
<i>append</i> :	<i>queue</i> × <i>elem</i>	→	<i>queue</i>
<i>remove</i> :	<i>queue</i>	→	<i>queue</i>
<i>is_empty</i> :	<i>queue</i>	→	<i>bool</i>
<i>next</i> :	<i>queue</i>	→	<i>elem</i>

 - Konstruktoren: *init, append*
 - V-Funktionen: *is_empty, next*
 - Sonstige O-Funktionen: *remove*
- Semantik
 - V-Funktion nach Konstruktor:

<i>is_empty</i> (<i>init</i>)	=	...
<i>is_empty</i> (<i>append</i> (<i>q, e</i>))	=	...
<i>next</i> (<i>init</i>)	=	...
<i>next</i> (<i>append</i> (<i>q, e</i>))	=	...

– Sonstige O-Funktion nach Konstruktor:

$$\begin{aligned} \text{remove}(\text{init}) &= \dots \\ \text{remove}(\text{append}(q, e)) &= \dots \end{aligned}$$

• FIFO (Gewöhnliche Warteschlange):

$$\begin{aligned} \text{is_empty}(\text{init}) &= \text{true} \\ \text{is_empty}(\text{append}(q, e)) &= \text{false} \\ \text{next}(\text{init}) &= \text{undef} \\ \text{next}(\text{append}(q, e)) &= \begin{cases} e & \text{is_empty}(q) \\ \text{next}(q) & \neg \text{is_empty}(q) \end{cases} \\ \text{remove}(\text{init}) &= \text{undef} \\ \text{remove}(\text{append}(q, e)) &= \begin{cases} \text{init} \\ \text{append}(\text{remove}(q), e) \end{cases} \end{aligned}$$

Axiomatische Festlegung der Modulsemantik für Warteschlangen: LIFO/FIFO

• LIFO (Stapel):

$$\begin{aligned} \text{is_empty}(\text{init}) &= \text{true} \\ \text{is_empty}(\text{append}(q, e)) &= \text{false} \\ \text{next}(\text{init}) &= \text{undef} \\ \text{next}(\text{append}(q, e)) &= e \\ \text{remove}(\text{init}) &= \text{undef} \\ \text{remove}(\text{append}(q, e)) &= q \end{aligned}$$

• Alternative Formulierung zu FIFO:

$$\begin{aligned} \text{is_empty}(\text{init}) &= \text{true} \\ \text{is_empty}(\text{append}(q, e)) &= \text{false} \\ \text{next}(\text{init}) &= \text{undef} \\ \text{remove}(\text{init}) &= \text{undef} \\ \text{next}(\text{append}(\text{init}, e)) &= e \\ \text{next}(\text{append}(\text{append}(q, e'), e)) &= \text{next}(\text{append}(q, e')) \\ \text{remove}(\text{append}(\text{init}, e)) &= \text{init} \\ \text{remove}(\text{append}(\text{append}(q, e'), e)) &= \text{append}(\text{remove}(\text{append}(q, e')), e) \end{aligned}$$

• Diese Beziehungen muss die Implementierung erfüllen. (Übung)

Warteschlange: Korrektheitsbeweis

- Implementation:

```
init      empty-queue
append(q,e) (add-element e q)
is_empty(q) (is-empty-queue? q)
next(q)    (first-of q)
remove(q)  (remove-from q)
```

Zu beweisen:

- $next(append(init, e)) = e$
(first-of (add-element e empty-queue)) = e
- $next(append(append(q, e'), e)) = next(append(q, e'))$
(first-of (add-element e (add-element e' q)))
= (first-of (add-element e' q))

Warteschlange: Korrektheitsbeweis (Forts.)

- Behauptung 1:

```
(first-of (add-element e empty-queue)) = e
(first-of (add-element e empty-queue))
= (first-of (cons e empty-queue))
= (car (cons e empty-queue))
= e
```

- Behauptung 2:

```
(first-of (add-element e
              (add-element e' q)))
= (first-of (add-element e' q))
```

Abk.: $q' = (add-element e' q)$

- Implementation von add-element und first-of:

```
(define (add-element element queue)
  (cond ((is-empty-queue? queue)
        (cons element empty-queue))
        (else
         (cons (first-of queue)
               (add-element element
                             (remove-from queue))))))
```

```
(define (first-of queue)
  (cond ((is-empty-queue? queue)
        (queue-error 1))
        (else (car queue))))
```

```
(first-of (add-element e q'))
= (first-of
   (cons (first-of q')
         (add-element e (remove-from q'))))
= (car
   (cons (first-of q')
         (add-element e (remove-from q'))))
= (first-of q')
```

Warteschlange: Korrektheitsbeweis (Forts.)

- Implementation von `remove-from`:

```
(define (remove-from queue)
  (cond ((is-empty-queue? queue)
        (queue-error 2))
        (else (cdr queue)) ))
```

- Zu zeigen $remove(append(init, e)) = init$
 $remove(appd(appd(q, e'), e)) = appd(remove(appd(q, e')), e)$

```
(remove-from (add-element e empty-queue))
= empty-queue
(remove-from (add-element e (add-element e' q)))
= (add-element e (remove-from (add-element e' q)))
```

Prioritätengesteuerte Warteschlange

(kleinere Zahl hat Vorrang)

- Übernehmbar:

```
is_empty(init)      = true
is_empty(append(q, e)) = false
next(init)          = undef
remove(init)        = undef
next(append(init, e)) = e
remove(append(init, e)) = init
```

- Fall 1:

```
(remove-from (add-element e empty-queue))
= (remove-from (cons e empty-queue))
= (cdr (cons e empty-queue))
= empty-queue
```

- Fall 2 mit $q' = (add-element e' q)$

```
(remove-from (add-element e q'))
= (remove-from
   (cons (first-of q)
         (add-element e (remove-from q')) ))
= (cdr
   (cons (first-of q)
         (add-element e (remove-from q')) ))
= (add-element e (remove-from q'))
```

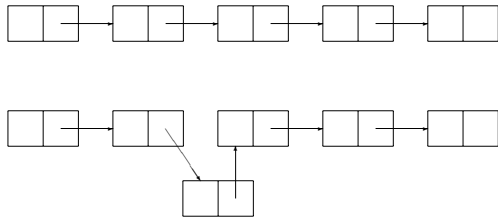
- e hat Priorität vor e' ($e < e'$):

```
next(append(append(q, e'), e)) = next(append(q, e))
remove(append(append(q, e'), e)) = append(remove(
                                         append(q, e)), e')
```

- e' hat Priorität vor e ($e' < e$):

```
next(append(append(q, e'), e)) = next(append(q, e'))
remove(append(append(q, e'), e)) = append(remove(
                                         append(q, e')), e)
```

- Eine Implementation (nicht optimal):

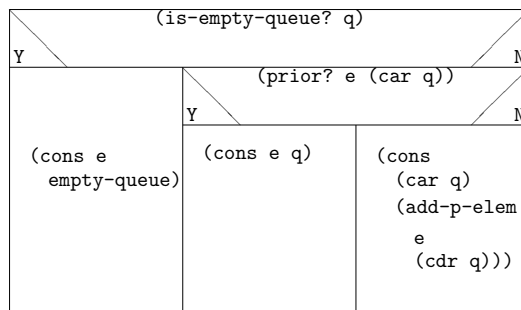


- Zum Korrektheitsbeweis:

- Zeige zunächst, dass die beiden Konstruktoren nur Warteschlangen erzeugen, in denen Elemente nach fallender Priorität geordnet sind.
- Beweise, dass aus dieser Aussage die "Axiome" folgen.

Eine Implementation der Prioritätenwarteschlange

- Übernahme aus FIFO: empty-queue, is-empty-queue?, first-of, remove-from
- Neu zu schreiben: add-p-elem



Implementierungsoptionen: Aufwandsbetrachtung

- Ungeordnete Liste:
 - beim Einfügen: konstante Zeit
 - beim Herausnehmen: ca. n Schritte
- Geordnete Liste:
 - beim Einfügen: ca. $n/2$ Schritte
 - beim Herausstreichen: konstante Zeit (wenn das vorrangige Element am Ende steht)
- Effizientere Idee: Speichere einmal durchgeführte Vergleiche!

$$(\forall 2 \leq k \leq n)(d_{k/2} < d_k)$$

- Das vorrangige Element steht auf Platz 1
- Einfügen: Füge auf Verdacht am Ende an, vergleiche mit dem halbierten Index, und vertausche gegebenenfalls: $\text{Id } n$ Schritte
- Herausnehmen: Nimm auf Verdacht das letzte Element auf die erste Position, vergleiche d_k mit d_{2k} und d_{2k+1} , und vertausche ggf. mit dem kleineren von beiden: $2 \text{ Id } n$ Schritte.

- Beispiel:

$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline 3 & 5 & 7 & 6 & 10 & 8 & 11 & \end{array}$$

- Implementierung durch einen binären Baum: Logarithmischer Aufwand für Einfügen und Entfernen!