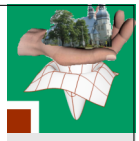


Informatik I
Abstrakte Datentypen (ADTs) & Spezifikation

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Erinnerung

- Stack & Queue = Daten + Operationen
- kommt nicht auf die interne Darstellung / Struktur der Daten an, sondern auf die Operationen, die mit den Daten durchgeführt werden können
- Wie beschreibt man die Operationen, die durchgeführt werden können?
- M.a.W.: Wie spezifiziert man einen (Daten-)Typ?

> *… the most fundamental problem in software development is **complexity**.* — *B. Stroustrup*
>
> *… our most important mental tool for coping with complexity is **abstraction**.* — *N. Wirth*

Formen der Abstraktion

- Die Brockhaus Enzyklopädie definiert den Begriff wie folgt:

Abstraktion [von lat. >Abziehen<]:

das Heraussondern bestimmter Merkmale in der Absicht, ***das Gleichbleibende und Wesentliche*** verschiedener Gegenstände zu ***erkennen***, um so zu allgemeinen Begriffen und Gesetzen zu kommen, vor allem im wissenschaftlichen Denken.

Welche Merkmale als wesentlich erachtet werden, hängt ... von der sachlichen Fragestellung ... ab.

Wie versteht man ein 100 000-Zeilen-Programm?

Die „ richtige" Abstraktionsebene

... liegt in der Mitte zwischen Extremen:

- Zu hohe Abstraktion:
  - „Das ist mir alles viel zu abstrakt (ohne erkennbare Substanz)."
  - „Das eigentliche Problem ist wegdefiniert."
  - „Bei beliebig hoher Abstraktion kommt man zu beliebig banalen Aussagen." [Trinks, Karlsruhe, Lineare Algebra I]
- Gewünscht:
  - „Das Wesentliche tritt hervor."
  - „Das Problem ist gut erkennbar."
  - „Die vielen unwesentlichen Details sind wegabstrahiert."
- Zu geringe Abstraktion:
  - „Ich erkenne jedes Detail, sehe aber den Wald vor Bäumen nicht."

(Programmier-)Sprachliche Abstraktion

- Programmiersprachen wurden zunächst in Generationen unterschiedlicher Abstraktionstufe eingeteilt.
- Dazu ein Gedankenexperiment, beginnend mit einem konkreten *Maschinenprogramm in Binärdarstellung*:

```
11110001 1010111100010110
00000000000000000000000000000110
00000000000000000000000000001000
11001011 1010111111110110 1010111111111000
11110001 1010111100000110
11001011 1010111111110010 1010111111101100
10010100 1010111111101001 1010111111101011
11101101 1010111111101100
11101011 10101111 11110001
00000000
```

??

1. Erster Abstraktionschritt: Komprimierung von Informationen:

```
00000000: F1 AF16
00000003: 00000006
00000007: 00000008
0000000B: CB AFF6 AFF8
00000010: F1 AF06
00000013: CB AFF2 AFEC
00000018: 94 AFE9 AFEB
0000001D: ED AFEC
00000020: EB AFF1
00000023: 00
```

Hexdumps wie diesen lesen gehörte zu den Fähigkeiten von Programmierern

2. Ein wichtiger Schritt: Übergang zu symbolischen Darstellung :

```
ggt:          SEG
              JUMP test
 a:           DD W 6
 b:           DD W 8
kleiner:      SUB W a, b
              JUMP test
groesser:     SUB W b, a
test:         CMP W a, b
              JLT kleiner
              JGT groesser
ende:         HALT
              END
```

Labels anstelle von Adressen und sog. "Mnemonics" erhöhen die Lesbarkeit

3. Kombinationen von Maschinenbefehlen durch universelle Kontrollstrukturen „höherer Programmiersprachen" ersetzt:

```
while a <> b:
      if a > b :
         a = a-b;
      else:
         b = b-a;
```

Die Notation ist kompakter und dank Normierung besser zugänglich

4. Kapselung in Funktion stellt klar den Zusammenhang zu den Mathematischen Gesetzen heraus:

```
functon ggt( a, b: Integer) return integer is
begin
     if a=b then
        reuturn a;
     elsif a>b then
        return ggt (a-b, b);
     else
        return ggt(a, b-a);
     end if;
end ggt;
```

ggt(a, a) = a

ggt(a, b) = ggt(a-b,b), falls a > b

ggt(a, b) = ggt (a, b-a), falls a < b

5. Mathematischen Gesetze der ggt-Funktion:

$$ggT(a,b) = x \;:\Leftrightarrow:$$
$$(a \mod x = 0 \;\wedge\; b \mod x = 0) \;\wedge$$
$$\forall\, y : (a \mod y = 0 \;\wedge\; b \mod y = 0) \rightarrow y \leq x$$

Gesetze:

i. **ggt(a,a) = a**

ii. **ggt(a,b) = ggt(b,a)**

iii. **ggt(a,b) = ggt(a-b,b), falls a > b**

iv. **ggt(a,b) = ggt(a,b-a), falls a < b**

(iv. folgt aus ii. und iii.)

Vorteile (programm-)sprachlicher Abstraktion

1. Abstraktere Programmnotationen sind der menschlichen Denkweise besser angepasst als die stark mit Details befrachteten konkreten Maschinenprogramme.
2. Es ist unwichtig, wie Daten, Ausdrücke und Kontrollstrukturen auf einer Maschine konkret realisiert werden. Die Umsetzung der Programme in eine maschinennahe Form kann von einem Programm („Compiler") erledigt werden.
3. Durch Einführung abstrakter Programmiersprachen wurden Programme portabel: Um alle Programme einer Sprache auf einem Rechner ausführbar zu machen, ist nur die Erstellung eines Compilers erforderlich.

Wovon kann man abstrahieren?

- Ein Algorithmus legt fest,
  - Welche Operationen
  - In welcher Reihenfolge
  - Auf welchen Daten
- Drei Ansatzpunkte für Abstraktion. Entsprechende Abstraktionsmechanismen heißen
  - Prozedurale Abstraktion
  - Kontrollabstraktion
  - Datenabstraktion

  Wir betrachen sie der Reihe nach.

Prozedurale Abstraktion

Kennen wir im Prinzip

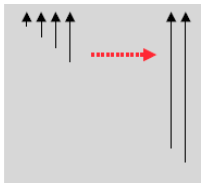
- Durch prozedurale Abstraktion entstehen aus nützlichen Programmstücken benannte, einfach verwendbare Operationen in Form von Prozeduren oder Funktionen.
- Die Parameter erhöhen die Anwendbarkeit dieser Konstrukte wesentlich
  - Bezüge auf globale Variablen erschweren die Verwendung von Unterprogrammen in einem anderen Kontext erheblich, *sie sollten daher in der Regel vermieden werden!*
- Die Spezifikation einer prozeduralen Abstraktion besteht aus einer Vorbedingung (= Anforderungen an die aktuellen Parameter) und der Nachbedingung (= Beschreibung des Ergebnisses des Aufrufs)

Kontrollabstraktion

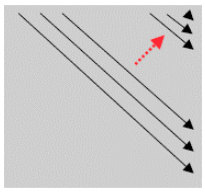
- Konkrete Kontrollstrukturen (Fallunterscheidungen und Schleifen) sind früh durch Abstraktion typischer Befehlsfolgen von Maschinensprachen entstanden.
- Sie sind nicht abstrakt („auf das Was bezogen"), sondern implementierungsnah („auf das Wie bezogen"): Man erwartet, daß Kontrollstrukturen effizient ausgeführt werden.
- David Parnas beklagt *„... unnecessary and arbitrary decisions that are forced on a programmer by a deterministic programming notation."*

Beispiel für Überspez. durch Kontrollstrukturen

```
for j in range (0,n):
    for i in range (j, 0, -1):
        s(i, j)
```



```
for d in range (0, n):
    j = d
    for i in range (0, n-d+1):
        s(i, j)
        j += 1
```



Beide durchlaufen Dreiecksmatrix, sind aber beide dafür völlig „überspezifiziert"

- Solche Überspezifikation ist schädlich, wenn man auf der Basis eines Grundalgorithmus Varianten betrachtet, z.B. Varianten, die Parallelausführung nutzen. Da Freiheitsgrade in der Ausführungsreihenfolge nicht erkennbar sind, weiß man auch nicht, wo Parallelausführung möglich ist.
- Überspezifikation der Ausführungsreihenfolge erschwert Programmbeweise, weil Unwesentliches zusätzlich mitgeführt und bewiesen werden muß.
- Es ist möglich (und nützlich), wesentliche Anforderungen an die Ausführungsreihenfolge explizit auszudrücken, z.B.
  - Durchlaufe den Baum nach Breite (oder Tiefe).
  - Durchlaufe einen Graphen in (inverser) toplogischer Ordnung.
  - Durchlaufe Dreiecksmatrix von Hauptdiagonale zur Ecke hin.

- In einfachen Fällen läßt sich Überspezifikation mit Hilfe der von Hoare eingeführten Mengenaufzählschleife

  ```
  for x ∈ M loop ...
  ```

  vermeiden: Die Aufzählungsreihenfolge bleibt unspezifiziert
- Moderne Sprachen (Java, C++, Python) stellen Iteratoren (*iterators*) oder Aufzählungskonstrukte (**`range`** und **`enumerator`**) bereit, mit denen man Container-Datenstrukturen und Schleifen irgendwie durchlaufen kann (→ erleichtert Wechsel der Datenstruktur)
  - Wesentliche Operation eines solchen Iterators : next-Operation, über die man sich jeweils das nächste Element geben lassen kann.

Datenabstraktion

- Im Arbeitsspeicher sind Informationen konkret als Bitfolgen (zu Bytes und Speicherwörtern zusammengefaßt) abgelegt.
- Auf der nächsten Abstraktionsebene werden die gleiche Bitmuster verschieden interpretiert, z.B. Speicherwörter als Zeiger oder als Zahlen (Integer oder Float), Bytes als Characters oder als Booleans.
- An Typen, die aus dem gleichen Grundtyp abgeleitet sind, erkennt man, daß es nicht nur auf die Wertemengen, sondern genauso auf die anwendbaren Operationen ankommt
  - Bsp.: Produkt von Zahlen
  - Falls Zahlen = Längen → Produkt = Flächenmaß,
  - Falls Zahlen = Jahreszahlen → Produkt sinnlos

- Containerklassen (Behälterklassen)
  - häufig benötigt man "Behälter", um andere Objekte (Elemente) zu organisieren
  - verschiedene Methoden, Elemente hinzuzufügen und auf Elemente zuzugreifen
    - Unterschiede in der Komplexität der Operationen
    - je nach Anforderungen sind andere Containerklassen optimal
  - Liste, Stack, Queue sind Beispiele solcher Containerklassen

Zwei Begriffe, leicht verwechselt

- Von Datenabstraktion spricht man, wenn Datentypen mit den auf sie anwendbaren Operationen so gekapselt sind, daß außen nur die in der Beschreibung ihrer Schnittstelle aufgeführten Bestandteile und Operationen verfügbar sind - nicht deren Implementierung
  - Beispiele: Moduln in Haskell, Pakete in Ada, Klassen in objekt-orientierten Sprachen (Java, C++, Python)
- Als Abstrakte Datentypen (ADTs) bezeichnet man die formale (oder manchmal auch informelle) Spezifikation von Datenabstraktionen. ADTs (und ihre Implementierung durch Datenabstraktionen) haben sich als praktisch außerordentlich nützliches Konzept bewährt.
- Historie: Konzept der Klassen wurde durch ADTs inspiriert

- ADTs werden programmiertechnisch u.a. durch Klassen unterstützt
- ADTs als formale Spezifikationen von Datenabstraktionen sind nützlich, weil:
  1. auch ohne Bezug auf eine Referenzimplementierung in einer Programmiersprache die Bedeutung der Abstraktion klar sein muß;
  2. die Korrektheit einer Implementierung nur relativ zu einer Spezifikation nachgewiesen werden kann;
  3. sich aus der Spezifikation Eigenschaften herleiten und auch beweisen lassen.
- Informelle Spezifikationen lassen mehr Raum für Interpretation und erlauben keine strengen Beweise.

Arten der Spezifikation von ADTs

- Axiomatisch
  - Beispiel: Natürliche Zahlen mittels Peano-Axiome
- Modell-basiert:
  - Analogie: Implementierung von Stack mittels Liste
- Zusicherungen-basiert:
  - Durch Prädikate, die vor und nach Operationen gelten

Zum Beispiel Stack

- Frage: Wie lassen sich Stacks ohne Bezug auf eine konkrete Implementierung spezifizieren?
- Edsger Dijkstra soll laut Meyer gesagt haben: „Abstract data types are a remarkable theory, whose purpose is to describe stacks".

Wdh.: Stack informell

- Charakteristisch für eine Datenabstraktion ist „was man damit machen" kann, d.h die anwendbaren Operationen:
- `push(e,k):` fügt Element e am oberen Ende des Stacks k an; Ergebnis ist der modifizierte Stack.
- `pop(k):` löscht das zuletzt angefügte Element; Ergebnis ist der modifizierte Stack.
- `top(k):` ergibt Kopie des obersten Stackelements.
- `create():` ergibt einen neuen, leeren Stack.
- `isEmpty(k):` wahr, wenn Stack k leer ist.

Formalisierung von Stack

- Relevante Mengen („Sorten”):
  - `B` Menge der Wahrheitswerte
  - `E` Menge der Stackelemente
  - `K` Menge der Stack, die Elemente aus E enthalten
- Signatur (Syntax der Operationen):

```
push      : E × K → K
pop       : K → K
top       : K → E
create    :   → K
isEmpty   : K → B
```


- Welche der folgenden Ausdrücke sind syntaktisch zulässig?
  - `isEmpty(push(e,create()))`
  - `top(isEmpty(create()))`
  - `top(push(b,pop(push(e,create()))))`
  - `push(e,pop(create()))`
  - `pop(push(create(),e))`

```
push    : E × K → K
pop     : K → K
top     : K → E
create  :   → K
isEmpty : K → B
```

- Analog in "objekt-orientierter Schreibweise"

Freie (Term-)Algebra

- Welche Stacks kann man mit diesen Operatoren beschreiben / erzeugen?
  - Alle diejenigen, zu denen es einen syntaktisch korrekten Term gibt, dessen Wert ein Stack ist
  - Beispiele:
    - `create()`
    - `push(e,create())`
    - `pop(push(e,create()))`
    - `push(e1,push(e2,create()))`
    - etc. ...
- Menge aller dieser Terme heißt freie Algebra (über der Sorte Stack)

Algebraischer Ansatz

- Zur Definition der Semantik (Bedeutung) der Operationen
- Dabei werden die Beziehungen zwischen den Operationen durch Axiome charakterisiert. Axiome sind hier ∀-quantifizierte Gleichungen zwischen Operationstermen.
- Ein Beispiel:

$$\forall e \in E \; \forall k \in K : \texttt{pop(push}(e, k)) = k$$

- Die Quantifizierung ist meist implizit

  (wird nicht hingeschrieben, aber angenommen).

- Ein kompletter Satz von Axiomen für Stack:

```
(K1) top(push(e,k)) = e
(K2) pop(push(e,k)) = k
(K3) isEmpty(create()) = true
(K4) isEmpty(push(e,k)) = false
```

- Beispiel: berechne

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

- Woher weiß man, daß man „genug" Axiome hat?

  (es könnten ja zu viele oder zu wenige sein)