

Algorithmen und Programmierung

Kapitel 8 Abstrakte Datentypen



Überblick

Abstrakte Datentypen

Signaturen und Algebren

Spezifikation von ADTs

Umsetzung von ADTs



- Motivation:
Wiederverwendbarkeit und Strukturierung von Software
- Ziel:

Beschreibung von Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache

- Abstrakter Datentyp (ADT)



- *Konkrete Datentypen:*
 - Konstruiert aus Basisdatentypen bzw. Java-Klassen
- *Abstrakte Datentypen:*
 - Spezifikation der Schnittstelle nach außen
 - Operationen und ihre Funktionalität



- ADT = Software-Modul
- **Kapselung**: darf nur über Schnittstelle benutzt werden
- **Geheimnisprinzip**: die interne Realisierung ist verborgen

... somit sind abstrakte Datentypen eine Grundlage des Prinzips der objektorientierten Programmierung!



- Eine Signatur Σ ist definiert als

$$\Sigma = (S, \Omega)$$

wobei Folgendes gilt:

- S ist eine Menge von Sorten.
- $\Omega = \{f_{S^*, S}\}$ definiert die Funktionssymbole für Funktionen $f: s_1 \times \dots \times s_n \rightarrow s$ mit Parametersorten s_1, \dots, s_n und Ergebnissorte s .
- Funktionen ohne Parameter heißen *Konstanten*



- Eine Algebra A_Σ zu einer Signatur Σ ist definiert als

$$A_\Sigma = (A_S, A_\Omega)$$

wobei Folgendes gilt:

- A_S sind die Trägermengen der Sorten in S .
- $A_\Omega = \{ A_f : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s \}$ sind Funktionen auf diesen Trägermengen.
- Eine Algebra heißt partiell definiert, wenn mindestens eine der Funktionen eine partielle Funktion ist.



Gleichungsspezifikation

- Die Angabe der *Signatur* legt die Namen der Typen sowie die Funktionssignaturen fest.
- Gleichungen dienen als Axiome zur Einschränkung möglicher Algebren als Modelle.
- Zusätzlich erfolgt evtl. ein Import anderer Spezifikationen.



```
type Nat
operators
  0: → Nat
  suc: Nat → Nat
  add: Nat × Nat → Nat
axioms  $\forall i, j : \text{Nat}$ 
  add (i, 0) = i
  add (i, suc (j)) = suc (add (i, j))
```



- Algebra der Wahrheitswerte

```
type Bool
operators
  true: → Bool
  false: → Bool
```

- Modelle für diese Spezifikation?



1. „Gewünschte“ Algebra:

$$A_{\text{Bool}} = \{ T, F \}; A_{\text{true}} := T; A_{\text{false}} := F$$

2. Wahrheitswerte über Zahlen realisiert:

$$A_{\text{Bool}} = \mathbb{N}; A_{\text{true}} := 1; A_{\text{false}} := 0$$

- Akzeptabel, aber die Trägermenge ist eigentlich zu groß

3. Algebra mit einem Element:

$$A_{\text{Bool}} = \{ 1 \}; A_{\text{true}} := 1; A_{\text{false}} := 1$$

- Diese Algebra ist sicherlich als Modell nicht erwünscht!



- Wie komme ich zu einem „kanonischen“ Standardmodell (also genau einer Algebra, die die Axiome respektiert) für eine gegebene Spezifikation?
 - Weder zu wenige noch zu viele Elemente
 - „No confusion, no junk“
- Wir gehen dazu in zwei Schritten vor:
 - In einem ersten Schritt konstruieren wir eine Algebra, die keine unnützen, also keine nicht durch Funktionsberechnung konstruierbaren Werte enthält.
 - Im zweiten Schritt setzen wir dann die Werte gleich, die mittels der Gleichungen als gleich charakterisiert werden (und nicht mehr!).



Die **Termalgebra** zu einer gegebenen Spezifikation ist definiert durch:

- Die Trägermenge, bestehend aus allen *korrekt gebildeten Termen* der Signatur.
- Der Funktionsanwendung als *Konstruktion des zugehörigen Terms*.



Vorüberlegung: A_{nat} induktiv definiert:

$$0 \in A_{\text{nat}}$$

$$x, y \in A_{\text{nat}} \Rightarrow \text{suc}(x) \in A_{\text{nat}} \text{ und } \text{add}(x, y) \in A_{\text{nat}}$$

- $A_{\text{nat}} = \{ 0, \text{suc}(0), \text{suc}(\text{suc}(0)), \text{add}(0, \text{suc}(0)), \dots \}$
- $A_0 = 0$
- $A_{\text{suc}}(x) = \text{suc}(x)$
- $A_{\text{add}}(x, y) = \text{add}(x, y)$



Die **Quotiententalgebra** QTA ist wie folgt definiert:

- Die Trägermenge besteht aus den Äquivalenzklassen auf den Termen gemäß der durch die Axiome induzierten Gleichheit „=“.
- Die Operationen auf den Äquivalenzklassen nehmen jeweils (Repräsentanten der) Äquivalenzklassen und haben als Ergebnis die (Repräsentanten der) Äquivalenzklasse des Ergebnisterms über den Repräsentanten.



Vorüberlegung: Äquivalenzklassen und ihre Repräsentanten

0 **repräsentiert** { 0, add(0,0),
add(add(0,0),0), ... }

suc(0) **repräsentiert** { suc(0),
add(0,suc(0)),
add(add(0,suc(0)),0), ... }

suc(suc(0)) ...

- $A_{\text{nat}} = \{ 0, \text{suc}(0), \text{suc}(\text{suc}(0)), \text{suc}^3(0), \text{suc}^4(0), \dots \}$
- $A_0 = 0$
- $A_{\text{suc}}(x) = \text{suc}(x)$
- $A_{\text{add}}(\text{suc}^n(0), \text{suc}^m(0)) = \text{suc}^{n+m}(0)$



- QTA als Muster für Standardsemantik
- Semantik eines ADT: Klasse aller Algebren die äquivalent zur QTA sind
- Äquivalenz basiert auf bijektiven Morphismen
- **Initialität:**
 - Prinzip der maximalen Ungleichheit
 - Zwei Werte sind nur dann gleich wenn dies aufgrund der Gleichungen bewiesen werden kann
- Symmetrisches Gegenstück ist **Terminalität:**
 - Prinzip der maximalen Gleichheit
 - Zwei Werte sind nur dann ungleich wenn dies bewiesen werden kann.
 - Erfordert Spezifikation von Ungleichheit!



```
type Set[Item]
operators
  create: → Set
  is_empty: Set → Bool
  insert: Set × Item → Set
  is_in: Set × Item → Bool
axioms ∀s : Set, ∀i : Item
  is_empty (create) = true
  is_empty (insert (s, i)) = false
  is_in (create, i) = false
  is_in (insert (s, i), j) =
    if i=j then true else is_in (s, j)
```



- Mengensemantik:

```
insert(insert(s,i),j) = insert(insert(s,j),i)
insert(insert(s,i),i) = insert(s,i)
```

- Gilt auch in Beispielspezifikation:
 - Eine Menge kann man nur mit `is_empty` und `is_in` „beobachten“, und deren Gleichungen respektieren diese Regeln.
- Diese Regeln können aber nicht mit den Gleichungen bewiesen werden:
 - `insert(insert(s,i),i)` und `insert(s,i)` sind unterschiedliche Elemente in der QTA!



- Konsequenzen aus der Theorie für Spezifikation
 - Oft initiale Semantik für Konstruktorfunktionen
 - Bei terminaler Semantik muss Ungleichheit auf importierten Datentypen erhalten bleiben
- Hier Theorie nur oberflächlich diskutiert
- Ab jetzt konkrete Spezifikationen!



1. **Type:** Name des neuen Typs zusammen mit eventuellen Parametern
2. **Functions:** *Signatures* der auf dem ADT definierten Funktionen: für jede Funktion der Name, Typen und Reihenfolge der Parameter, Ausgabebetyp sowie ggf. ob die Funktion total (\rightarrow) oder partiell ($\not\rightarrow$) ist
3. **Axioms:** beschreiben *Eigenschaften* der eingeführten Funktionen als prädikatenlogische Formeln (zumeist Gleichungen)
4. **Preconditions:** für jede partielle Funktion kann eine Vorbedingung angegeben werden, die angibt, wann sie definiert ist



- *Konstruktorfunktionen:* Funktionen zum *Aufbauen* von Elementen des ADT
- *Selektorfunktionen:* die Inversen der Konstruktorfunktionen; zerlegen ein Element des Datentyps in seine „Einzelteile“
- *sonstige Funktionen*



□ ADT für natürliche Zahlen

```
type Nat
operators
  0: → Nat
  suc: Nat → Nat
  add: Nat × Nat → Nat
axioms  $\forall i, j : \text{Nat}$ 
  add (i, 0) = i
  add (i, suc (j)) = suc (add (i, j))
```



```
type List(T)
import Nat
operators
  [] : → List
  _ : _ : T × List → List
  head : List → T
  tail : List → List
  length : List → Nat
axioms  $\forall l : \text{List}, \forall x : T$ 
  head (x : l) = x
  tail (x : l) = l
  length ([]) = 0
  length (x : l) = suc (length(l))
```

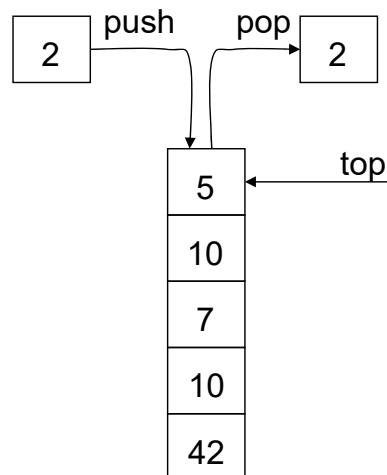


- `_ : _` ist Notation für Infix-Operatoren (wie `+` in arithmetischen Ausdrücken)
- `import` importiert bereits definierten Datentypen
- mögliche Werte für Listen über Zahlen sind nun:

```
[]  
1 : []  
5 : ( 6 : ( 7 : [] ) )  
1 : ( 1 : [] )
```



- LIFO-Prinzip: Last-In-First-Out-Speicher



```
type Stack(T)
import Bool
operators
  empty :  $\rightarrow$  Stack
  push  : Stack  $\times$  T  $\rightarrow$  Stack
  pop   : Stack  $\rightarrow$  Stack
  top   : Stack  $\rightarrow$  T
  is_empty : Stack  $\rightarrow$  Bool
axioms  $\forall s : \text{Stack}, \forall x : T$ 
  pop (push (s, x)) = s
  top (push (s, x)) = x
  is_empty (empty) = true
  is_empty (push (s, x)) = false
```



```
empty == []
push(l,x) == x:l
pop(x:l) == l
top(x:l) == x
is_empty([]) == true
is_empty(x:l) == false
```



- Auswertung einfacher arithmetischer Ausdrücke
- analysierte Sprache:

```
A ::= B =
B ::= Int | (B+B) | (B*B)
```

- Beispiel:

$$((3 + (4 * 5)) * (6 + 7)) =$$


$$\begin{aligned} \text{value}(t) &= \text{eval} \langle t, \text{empty} \rangle & (1) \\ \text{eval} \langle t, S \rangle &= \text{eval} \langle t, (S) \rangle & (2) \\ \text{eval} \langle *t, S \rangle &= \text{eval} \langle t, *S \rangle & (3) \\ \text{eval} \langle +t, S \rangle &= \text{eval} \langle t, +S \rangle & (4) \\ \text{eval} \langle)t, y*xS \rangle &= \text{eval} \langle)t, \llbracket x * y \rrbracket S \rangle & (5) \\ \text{eval} \langle)t, y+xS \rangle &= \text{eval} \langle)t, \llbracket x + y \rrbracket S \rangle & (6) \\ \text{eval} \langle)t, x(S) \rangle &= \text{eval} \langle t, xS \rangle & (7) \\ \text{eval} \langle =, x \text{ empty} \rangle &= x & (8) \\ \text{eval} \langle xt, S \rangle &= \text{eval} \langle t, \llbracket x \rrbracket S \rangle & (9) \end{aligned}$$


Auswertungsbeispiel (1)

bearbeiteter Term t	Stack	Regel
$' ((3 + (4 * 5)) * (6 + 7)) = '$]	(2)
$' (3 + (4 * 5)) * (6 + 7) = '$	(]	(2)
$' 3 + (4 * 5)) * (6 + 7) = '$	((]	(9)
$' + (4 * 5)) * (6 + 7) = '$	3 ((]	(4)
$' (4 * 5)) * (6 + 7) = '$	+3 ((]	(2)
$' 4 * 5)) * (6 + 7) = '$	(+3 ((]	(9)
$' * 5)) * (6 + 7) = '$	4 (+3 ((]	(3)
$' 5)) * (6 + 7) = '$	*4 (+3 ((]	(9)
$')) * (6 + 7) = '$	5*4 (+3 ((]	(5)
$')) * (6 + 7) = '$	20 (+3 ((]	(7)
$') * (6 + 7) = '$	20+3 ((]	(6)



Auswertungsbeispiel (2)

bearbeiteter Term t	Stack	Regel
$') * (6 + 7) = '$	20+3 ((]	(6)
$') * (6 + 7) = '$	23 ((]	(7)
$' * (6 + 7) = '$	23 (]	(3)
$' (6 + 7) = '$	*23 (]	(2)
$' 6 + 7) = '$	(*23 (]	(9)
$' + 7) = '$	6 (*23 (]	(4)
$' 7) = '$	+6 (*23 (]	(9)
$') = '$	7+6 (*23 (]	(6)
$') = '$	13 (*23 (]	(7)
$') = '$	13 *23 (]	(5)
$') = '$	299 (]	(7)
$' = '$	299]	(8)



$A ::= B =$

$B ::= \text{Int} \mid B+B \mid B*B \mid (B)$

- Arithmetische Ausdrücke ohne vollständige Klammerung
- Interpreter muss die unterschiedlichen Prioritäten der Operatoren berücksichtigen
- Lösungsansatz – zwei Stacks:
 - einen für die Operatoren,
 - den zweiten für die Operanden



$$\text{value}(t) = \text{eval} \langle t, \text{empty}, \text{empty} \rangle \quad (10)$$

$$\text{eval} \langle t, S_1, S_2 \rangle = \text{eval} \langle t, (S_1, S_2) \rangle \quad (11)$$

$$\text{eval} \langle *t, S_1, S_2 \rangle = \text{eval} \langle t, *S_1, S_2 \rangle \quad (12)$$

$$\text{eval} \langle +t, *S_1, yxS_2 \rangle = \text{eval} \langle +t, S_1, \llbracket x * y \rrbracket S_2 \rangle \quad (13)$$

$$\text{eval} \langle +t, S_1, S_2 \rangle = \text{eval} \langle t, +S_1, S_2 \rangle \quad (14)$$

$$\text{eval} \langle)t, +S_1, yxS_2 \rangle = \text{eval} \langle)t, S_1, \llbracket x + y \rrbracket S_2 \rangle \quad (15)$$

$$\text{eval} \langle)t, *S_1, yxS_2 \rangle = \text{eval} \langle)t, S_1, \llbracket x * y \rrbracket S_2 \rangle \quad (16)$$

$$\text{eval} \langle)t, (S_1, S_2) \rangle = \text{eval} \langle t, S_1, S_2 \rangle \quad (17)$$

$$\text{eval} \langle =, +S_1, yxS_2 \rangle = \text{eval} \langle =, S_1, \llbracket x + y \rrbracket S_2 \rangle \quad (18)$$

$$\text{eval} \langle =, *S_1, yxS_2 \rangle = \text{eval} \langle =, S_1, \llbracket x * y \rrbracket S_2 \rangle \quad (19)$$

$$\text{eval} \langle =, \text{empty}, x \text{ empty} \rangle = x \quad (20)$$

$$\text{eval} \langle xt, S_1, S_2 \rangle = \text{eval} \langle t, S_1, \llbracket x \rrbracket S_2 \rangle \quad (21)$$



Komplexere Terme: Abarbeitung (1)

bearbeiteter Term t	Stack S_1	Stack S_2	Regel
' (4*5+3) * (6+7) =']]	(11)
' 4*5+3) * (6+7) ='	(]]	(21)
' *5+3) * (6+7) ='	(]	4]	(12)
' 5+3) * (6+7) ='	* (]	4]	(21)
' +3) * (6+7) ='	* (]	5 4]	(13)
' +3) * (6+7) ='	(]	20]	(14)
' 3) * (6+7) ='	+ (]	20]	(21)
') * (6+7) ='	+ (]	3 20]	(15)
') * (6+7) ='	(]	23]	(17)

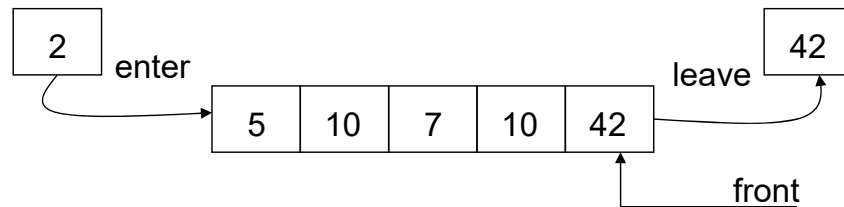


Komplexere Terme: Abarbeitung (2)

bearbeiteter Term t	Stack S_1	Stack S_2	Regel
') * (6+7) ='	(]	23]	(17)
' * (6+7) =']	23]	(12)
' (6+7) ='	*]	23]	(11)
' 6+7) ='	(*]	23]	(21)
' +7) ='	(*]	6 23]	(14)
' 7) ='	+ (*]	6 23]	(21)
') ='	+ (*]	7 6 23]	(15)
') ='	(*]	13 23]	(17)
' ='	*]	13 23]	(19)
' =']	299]	(20)



FIFO-Prinzip: „First in, first out.“



```
type Queue (T)
import Bool
operators
  empty : → Queue
  enter : Queue × T → Queue
  leave : Queue → Queue
  front : Queue → T
  is_empty : Queue → Bool
axioms ∀q : Queue, ∀x, y : T
  leave (enter (empty, x)) = empty
  leave (enter (enter(q, x), y)) =
    enter (leave (enter(q, x)), y)
  front (enter (empty, x)) = x
  front (enter (enter(q, x), y)) =
    front (enter (q, x))
  is_empty (empty) = true
  is_empty (enter(q, x)) = false
```



- Wie komme ich zu den Gleichungen?
- Wann habe ich genug Gleichungen?
- Wann fehlt mir ein Axiom?

→ leider keine einfache Antworten zu erwarten



1. Festlegung der Konstruktoren
 - Bei Stack: `empty` und `push`
2. Eventuell: Axiome zur Gleichsetzung von Konstruktortermen
 - Siehe Beispiel „Set“: mehrfaches Einfügen und Reihenfolge der Einfügeoperationen ist unerheblich
3. Definition geeigneter Selektoren
 - Selektoren erlauben den lesenden Zugriff auf konstruierte Werte und liefern meist importierte Datentypen zurück
 - Die Auswertung von Selektoren kann auf ausschließlich mit Konstruktoren konstruierten Termen erfolgen
 - Bei Stack: `is_empty`, `top`
4. Weitere Operationen („Manipulatoren“) festlegen
 - Bei Stack: `pop`
5. Fehlersituationen abfangen



- ❑ Wenn möglich: direkt auf Konstruktoren und Selektoren zurückführen
- ❑ Regeln von links nach rechts als Ersetzungsregeln aufbauen
 - ❑ Rechte Seite 'einfacher' als linke Seite
 - ❑ Bei Rekursion:
 - ❑ Argumente 'einfacher' (echt monoton schrumpfend)
 - ❑ Abbruch garantiert? (Vorsicht! — wird oft vergessen)
 - ❑ Oft: Vorgehen analog zu applikativen Algorithmen
- ❑ Vollständige Fallunterscheidung:
 - ❑ Jeder Konstruktor muss bei den Parametern berücksichtigt werden



- ❑ ADT in Programmiersprachen
 - ❑ Konzept der Kapselung
 - ❑ Verbergen der internen Repräsentation
 - ❑ Gleiche Verwendung trotz unterschiedlicher Implementierung
 - ❑ Vorteile: Stabilität gegenüber Änderungen, Auswahl einer geeigneten Implementierungsvariante
- ❑ ADT in Java (objektorientiert!)
 - ❑ Typen → Klassen
 - ❑ Funktionen → Methoden
 - ❑ Klassen als konkrete Implementierung

Das nächste Kapitel führt daher in die objektorientierte Programmierung mit Java ein.



- ❑ Abstrakte Datentypen
- ❑ Kapselung & Geheimnisprinzip
- ❑ Implementierungsunabhängige Definition
- ❑ Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kapitel 11-12

