Programmierung und Algorithmen

Kapitel 8
Abstrakte Datentypen

(Dieses Kapitel wird ab der Prüfungsordnung 2021 nicht mehr behandelt, ist daher nicht prüfungsrelevant und wird nur für interessierte Studierende bereitgestellt.)

Überblick

Einführung

- Motivation:
  Wiederverwendbarkeit und Strukturierung von Software
- Ziel:

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

- Abstrakter Datentyp (ADT)

Formen von Datentypen

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

Prinzipien von Abstrakten Datentypen

- 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!

Signatur

- Eine Signatur $\sum$ ist definiert als

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

wobei Folgendes gilt:

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

- Funktionen ohne Parameter heißen *Konstanten*

Algebra zu einer Signatur

- 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.
  - $A_\Omega = \{ A_f : A_{s1} \times \cdots \times A_{sn} \rightarrow A_s \}$ sind Funktionen auf diesen Trägermengen.

- Eine Algebra heißt partiell definiert, wenn mindestens eine der Funktionen eine partielle Funktion ist.

Algebraische Spezifikation

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.

Algebraische Spezifikation: Beispiel

```
type Nat
operators
    0: → Nat
    suc: Nat → Nat
    add: Nat × Nat → Nat
axioms ∀i,j : Nat
    add (i, 0) = i
    add (i, suc (j)) = suc (add (i, j))
```


Spezifikationen und Modelle

- Algebra der Wahrheitswerte

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

- Modelle für diese Spezifikation?

Modelle für `Bool`

1. „Gewünschte“ Algebra:

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

2. Wahrheitswerte über Zahlen realisiert:

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

   - Akzeptabel, aber die Trägermenge ist eigentlich zu groß

3. Algebra mit einem Element:

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

   - Diese Algebra ist sicherlich als Modell nicht erwünscht!

Standardmodelle

- 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!).

Termalgebra

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*.

Beispiel Termalgebra für `Nat`

Vorüberlegung: A_{nat} induktiv definiert:

$0 \in A_{nat}$

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

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

Quotiententermalgebra QTA

Die Quotiententermalgebra 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.

Beispiel QTA für `Nat`

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)) ...
```

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

Initiale Semantik

- 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!

Probleme mit initialer Semantik

```
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)
```

Probleme mit initialer Semantik II

- 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!

Spezifikation und Beispiele

- 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!

Spezifikation von Abstrakten Datentypen

1. **Type**: Name des neuen Typs zusammen mit eventuellen Parametern
2. **Functions**: *Signaturen* der auf dem ADT definierten Funktionen: für jede Funktion der Name, Typen und Reihenfolge der Parameter, Ausgabetyp sowie ggf. ob die Funktion total ($\rightarrow$) oder partiell ($\nrightarrow$) 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

Klassifikation der Funktionen

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

Beispiel für Abstrakten Datentyp

- ADT für natürliche Zahlen

```
type Nat
operators
    0: → Nat
    suc: Nat → Nat
    add: Nat × Nat → Nat
axioms ∀i,j : Nat
    add (i, 0) = i
    add (i, suc (j)) = suc (add (i, j))
```

Beispiel: Liste

```
type List(T)
import Nat
operators
      [] : → List
      _ : _ : T × List → List
            head : List → T
      tail : List → List
      length : List → Nat
axioms ∀l : List, ∀x : T
            head (x : l) = x
      tail (x : l) = l
      length ([]) = 0
      length (x : l) = suc(length(l))
```

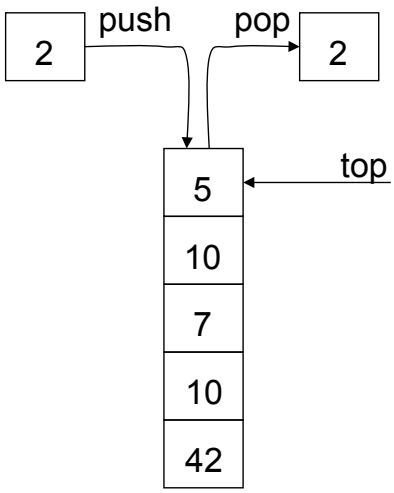
Liste: Erläuterungen

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

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

Kellerspeicher (Stack): Prinzip

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



Stack: Definition des ADT

```
type Stack(T)
import Bool
operators
    empty : → Stack
    push : Stack × T → Stack
    pop : Stack → Stack
    top : Stack → T
    is_empty : Stack → Bool
axioms ∀s : Stack, ∀x : T
    pop (push (s, x)) = s
    top (push (s, x)) = x
    is_empty (empty) = true
    is_empty (push (s, x)) = false
```

Keller implementiert über Liste

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

Beispiel für Keller-Nutzung

- Auswertung einfacher arithmetischer Ausdrücke
- analysierte Sprache:

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

- Beispiel:

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

Bearbeitungsregeln

value(*t*)	=	*eval*	⟨ *t* , *empty* ⟩	(1)
eval ⟨ (*t* , *S* ⟩	=	*eval*	⟨ *t* , (*S* ⟩	(2)
eval ⟨ ∗ *t* , *S* ⟩	=	*eval*	⟨ *t* , ∗ *S* ⟩	(3)
eval ⟨ +*t* , *S* ⟩	=	*eval*	⟨ *t* , +*S* ⟩	(4)
eval ⟨)*t* , *y*∗*xS* ⟩	=	*eval*	⟨)*t* , ⟦ *x* ∗ *y* ⟧ *S* ⟩	(5)
eval ⟨)*t* , *y*+*xS* ⟩	=	*eval*	⟨)*t* , ⟦ *x* + *y* ⟧ *S* ⟩	(6)
eval ⟨)*t* , *x*(*S* ⟩	=	*eval*	⟨ *t* , *xS* ⟩	(7)
eval ⟨ =, *x empty* ⟩	=	*x*		(8)
eval ⟨ *xt* , *S* ⟩	=	*eval*	⟨ *t* , ⟦ *x* ⟧ *S* ⟩	(9)

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)

Komplexere Terme

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

Komplexere Terme: Regeln

$$
\begin{array}{lcll}
value(t) & = & eval\ \langle\ t\ ,\ empty,\ empty\ \rangle & (10)\\
eval\ \langle\ (t\ ,\ S_1,\ S_2\ \rangle & = & eval\ \langle\ t\ ,\ (S_1,\ S_2\ \rangle & (11)\\
eval\ \langle\ *t\ ,\ S_1,\ S_2\ \rangle & = & eval\ \langle\ t\ ,\ *S_1,\ S_2\ \rangle & (12)\\
eval\ \langle\ +t\ ,\ *S_1,\ yxS_2\ \rangle & = & eval\ \langle\ +t\ ,\ S_1,\ [\![\ x * y\ ]\!]\ S_2\ \rangle & (13)\\
eval\ \langle\ +t\ ,\ S_1,\ S_2\ \rangle & = & eval\ \langle\ t\ ,\ +S_1,\ S_2\ \rangle & (14)\\
eval\ \langle\ )t\ ,\ +S_1,\ yxS_2\ \rangle & = & eval\ \langle\ )t\ ,\ S_1,\ [\![\ x + y\ ]\!]\ S_2 & (15)\\
eval\ \langle\ )t\ ,\ *S_1,\ yxS_2\ \rangle & = & eval\ \langle\ )t\ ,\ S_1,\ [\![\ x * y\ ]\!]\ S_2\ \rangle & (16)\\
eval\ \langle\ )t\ ,\ (S_1,\ S_2\ \rangle & = & eval\ \langle\ t\ ,\ S_1,\ S_2\ \rangle & (17)\\
eval\ \langle\ =,\ +S_1,\ yxS_2\ \rangle & = & eval\ \langle\ =,\ S_1,\ [\![\ x + y\ ]\!]\ S_2\ \rangle & (18)\\
eval\ \langle\ =,\ *S_1,\ yxS_2\ \rangle & = & eval\ \langle\ =,\ S_1,\ [\![\ x * y\ ]\!]\ S_2\ \rangle & (19)\\
eval\ \langle\ =,\ empty,\ x\ empty\ \rangle & = & x & (20)\\
eval\ \langle\ xt\ ,\ S_1,\ S_2\ \rangle & = & eval\ \langle\ t\ ,\ S_1,\ [\![\ x\ ]\!]\ S_2\ \rangle & (21)
\end{array}
$$

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)

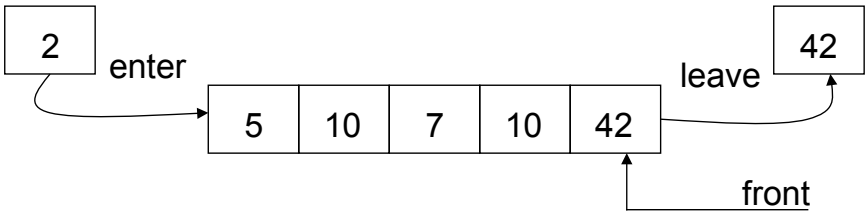
bearbeiteter Term t	Stack S_1	Stack S_2	Regel
´) ∗ (6+7) =´	( ]	23]	(17)
´ ∗ (6+7) =´	]	23]	(12)
´ (6+7) =´	∗]	23]	(11)
´ 6+7) =´	(∗]	23]	(21)
´ +7) =´	(∗]	6 23]	(14)
´ 7) =´	+ (∗]	6 23]	(21)
´) =´	+ (∗]	7 6 23]	(15)
´) =´	(∗]	13 23]	(17)
´ =´	∗]	13 23]	(19)
´ =´	]	299]	(20)

Nebenbemerkung zur Termauswertung

- Eine „einfache“ Erweiterung dieser Regelmenge um entsprechende Regeln für „-“ und „/“ führt nicht zum gewünschten Ergebnis.
- Der Grund hierfür ist, dass die Operationen „+“ und „*“ sowohl links- als auch rechtsassoziativ sind, die Operationen „-“ und „/“ jedoch (nur) linksassoziativ sind
  - Beispiel: 10 + 3 + 2 = (10 + 3) + 2 = 10 + (3 + 2)
    Aber: 10 - 3 - 2 = (10 - 3) - 2 ≠ 10 - (3 – 2)
  - Die Regelmenge führt zu einer rechtsassoziativen Auswertung, ist also nur für „+“ und „*“ korrekt.
- Sollen auch linksassoziative Operationen unterstützt werden, so wird ein anderer Algorithmus benötigt – siehe hierzu auch:
  `https://www.cis.upenn.edu/~matuszek/cit594-2002/Assignments/5-expressions.html`

Warteschlange (Queue)

FIFO-Prinzip: „First in, first out.“



Warteschlange (Queue)

```
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
```

Entwurf von ADT

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

→ leider keine einfache Antworten zu erwarten

Systematische Vorgehensweise

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

Festlegung weiterer Operationen

- 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

Umsetzung von ADTs

- 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.

Zusammenfassung

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