Lösungen zu „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“

Im Lehrbuch „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“ von Peter Hubwieser und Gerd Aiglstorfer (erschienen im Oldenbourg Verlag München, 2004, ISBN 3-486-27572-0) schließt jedes Kapitel mit einigen Aufgaben.

Die folgenden Lösungsvorschläge wurden von Gerd Aiglstorfer erstellt. Weitere Informationen und Hilfestellungen zum Lehrbuch finden Sie unter http://www.aigl.de.

Inhalt

1 Lösungsvorschläge zu „(Basis-) Datenstrukturen“

1.1 Lösungsvorschlag Aufgabe 10.1

Das Löschen aller eingefügten Elemente ist nicht möglich. Es wird ein Element in den anfangs leeren *Space* über natürlichen Zahlen eingefügt:

```
emptySpace
joinSpace(emptySpace, 4)
```

Darauf ist keine *leaveSpace*-Regel anwendbar (die einzig vorhandene Regel benötigt mindestens zwei Elemente für die Anwendung). Sobald ein Element in einen solchen *Space* eingefügt wurde, kann kein leerer *Space* mehr erzeugt werden.

1.2 Lösungsvorschlag Aufgabe 10.2

Der abstrakte Datentyp *Sequence* aus Kapitel 10.2 wird wie folgt erweitert:

```
…
functions …
          last (seq m): m
          front (seq m): seq m
behaviour …
          last(⟨a1⟩ ○ … ○ ⟨an-1⟩ ○ ⟨an⟩) = an                  (n ≥ 2)
          front(⟨a1⟩ ○ … ○ ⟨an-1⟩ ○ ⟨an⟩) = ⟨a1⟩ ○ … ○ ⟨an-1⟩  (n ≥ 2)
endDatatype
```

Diese Variante lässt die Frage nach dem Verhalten bei Sequenzen der Länge 1 offen (dies ist auch nicht unbedingt nötig, da die beiden Regeln im Sinne der Vorgaben festlegen, wie die beiden Funktionen arbeiten sollen). Die folgende Variante spezifiziert auch Sequenzen mit einem Element:

```
…
behaviour …
          last(⟨a⟩) = a
          front(⟨a⟩) = empty
          last(⟨a⟩ ○ s) = last(s)          falls not isempty(s)
          front(⟨a⟩ ○ s) = ⟨a⟩ ○ front(s)   falls not isempty(s)
endDatatype
```

Die Beschränkung der beiden letzten Regeln auf nicht leere Sequenzen *s* ist nötig, um die Eindeutigkeit der Regeln zu gewährleisten (wenn *s* leer ist, wäre nicht klar festgelegt, welche

der beiden Regeln für die jeweilige Funktion zu verwenden ist). Außerdem spezifizieren wir das Verhalten von Funktionen bei Fehleingaben nicht (siehe dazu die Schlussbemerkungen in Kapitel 10.1).

Es existiert noch eine weitere Variante, bei der auf Einschränkungen verzichtet werden kann (falls *s* leer ist, befindet sich ein Element in der Liste) und das Verhalten vollständig angegeben ist (Anmerkung: die eleganteste Lösung):

```
…
behaviour …
          last(s ∘ ⟨a⟩) = a
          front(s ∘ ⟨a⟩) = s
endDatatype
```

1.3 Lösungsvorschlag Aufgabe 10.3

Der abstrakte Datentyp *Queue* aus Kapitel 10.3 wird wie folgt erweitert:

```
…
functions …
          concat (queue m, queue m): queue m
behaviour …
          concat(q, empty) = q
          concat(q, append(p, a)) = append(concat(q, p), a)
endDatatype
```

Dabei wird die zweite Warteschlange mit dem Ende der ersten verknüpft.

1.4 Lösungsvorschlag Aufgabe 10.4

Um Sequenzen zur Realisierung von Warteschlangen zu verwenden, muss das Verhalten der Funktionen für Warteschlangen durch Funktionen auf Sequenzen simuliert werden. Wir benutzen die Schnittstelle des abstrakten Datentyps *Sequence* (der Zusatz *QS* zu Beginn der Funktionsnamen steht für *QueueSequence* als Hinweis auf die Funktionen von Warteschlangen auf Basis von Sequenzen) und realisieren Warteschlangen auf natürlichen Zahlen:

```
function QSisempty (seq nat s): bool
begin
return isempty(s)
endfct

function QSappend (seq nat s, nat a): seq nat
begin
return s ∘ ⟨a⟩
endfct
```

```
function QSfirst (seq nat s): nat
begin
return first(s)
endfct

function QSrest (seq nat s): seq nat
begin
return rest(s)
endfct
```

1.5 Lösungsvorschlag Aufgabe 10.5

Der abstrakte Datentyp *Stack* aus Kapitel 10.4 wird wie folgt erweitert:

```
…
functions …
          concat (stack m, stack m): stack m
behaviour …
          concat(empty, t) = t
          concat(push(s, a), t) = push(concat(s, t), a)
endDatatype
```

Diese Verhaltensbeschreibung bewirkt, dass der Keller im zweiten Eingabeparameter an den anderen Keller unten angehängt wird.

1.6 Lösungsvorschlag Aufgabe 10.6

Der abstrakte Datentyp *Binarytree* aus Kapitel 10.5 wird wie folgt erweitert:

```
…
use bool, nat
functions …
          height (bintree m): nat
          isbalance (bintree m): bool
behaviour …
          height(maketree(empty, r, empty)) = 0
          height(maketree(empty, r, rt)) = 1 + height(rt)
          height(maketree(lt, r, empty)) = 1 + height(lt)
          height(maketree(lt, r, rt)) =
            1 + max(height(lt), height(rt))
          isbalance(maketree(lt, r, rt)) =
            equal(height(lt), height(rt))
endDatatype
```

Dabei gilt: *lt* und *rt* sind jeweils nicht leere Binärbäume (enthalten also mindestens die Wurzel). Es wird die Funktion *height* zur Bestimmung der Höhe eines Baumes benötigt. Dabei habe ein leerer Baum keine Höhe (für den Fehlerfall *height(empty)* sei wieder auf die Schlussbemerkung in Kapitel 10.1 verwiesen) und sie sei 0, wenn nur die Wurzel existiert. Die Höhe ergibt sich dann aus der Anzahl der Kanten auf einem Pfad von der Wurzel über Knoten zu einem Blatt auf der letzten Ebene eines Baumes (eine Definition des Begriffs „Pfad“ befindet sich in Kapitel 14.1).

Für *isbalance* wird in der Spezifikation vorausgesetzt, dass auch tatsächlich zwei Teilbäume vorhanden sind. Die Ausgaben bzw. das Verhalten bei nicht vorhandenen Teilbäumen obliegt einer entsprechenden Definition oder Fehlerbehandlung, die wir hier nicht näher betrachten wollen.

Die beiden verwendeten Hilfsfunktionen geben wir folgendermaßen an:

```
function max (nat a, b): nat
return if a ≥ b then a else b

function equal (nat a, b): bool
return if a = b then true else false
```

Bemerkung: Diese Spezifikation ist schon sehr implementierungsnah. Prinzipiell sollte dies jedoch möglichst vermieden werden (es hängt vom Problem und der benötigten Genauigkeit ab), um den Einsatz einer Spezifikation in mehreren Szenarien zu erlauben (Stichwort *Wiederverwendbarkeit*). Es ist wichtig, dass Spezifikationen in sich schlüssig sind. Darum empfiehlt es sich, mit überschaubarem Umfang zu beginnen, die Konsistenz immer wieder zu überprüfen und eine Erweiterung nur bei Bedarf vorzunehmen.

1.7 Lösungsvorschlag Aufgabe 10.7

Die Funktion durchläuft die einfach verkettete Liste einmal von links nach rechts und zählt die Anzahl der Elemente *a*:

```
function search (nat a, var pelist first): nat
var nat result := 0;
var pelist elem := first;
begin
while elem ≠ null do
      if elem↓.item = a then result := result + 1 endif
      elem := elem↓.next
endwhile
return result
endfct
```

1.8 Lösungsvorschlag Aufgabe 10.8

search für zweifach verkettete Listen arbeitet wie die Suche in Aufgabe 10.7. Um von rechts nach links durch die Liste zu navigieren, müssen lediglich die Zeiger etwas anders gesetzt werden:

```
function search (nat a, var pzlist first, last): nat
var nat result := 0;
var pzlist elem := last;
begin
while elem ≠ null do
      if elem↓.item = a then result := result + 1 endif
      elem := elem↓.pre
endwhile
return result
endfct
```