Algorithmen und Programmierung 3, WS 2003/2004 — Aufgabe 57

Zwei Spezifikationen, zwei Implementationen, eine Abstraktionsfunktion und anderthalb Beweise
(zusammengefasst von Till Zoppke)

---

0. Bedienungsanleitung
   (a) Drucken Sie die Datei auf Papier und lesen Sie den Ausdruck.
   (b) Was ist eine gültige Darstellung? Was ist der Unterschied zwischen einer Invariante und einer Vorbedingung? Was ist ein mathematisches Modell? Was ist der Unterschied zwischen einer modellierenden und einer algebraischen Spezifikation? Was beweist man mit Hilfe der Abstraktionsfunktion?
   (c) Die Operationen
   - sindGleich : MULTI x MULTI → BOOL
   - istLeer: MULTI → BOOL

   testen zwei Multimengen auf Gleichheit bzw. eine auf Leerheit. Führen Sie für beide Operationen die Schritte (a) – (d) gemäß der Aufgabenstellung durch.
   (d) Spezifizieren Sie die Multimenge mit einem anderen Modell, als Menge von Paaren. Diese Variante hat den Vorteil, dass sich der Test auf Gleichheit einfacher spezifizieren lässt. Wie sieht dann die Abstraktionsfunktion aus?
   (e) Ändern Sie in der algebraische Spezifikation die Axiome so ab, dass Elemente tatsächlich entfernt werden (also dass aus der Kette von Operationen, die mit Leer beginnt, die Einfügen-Operation wieder ungeschehen gemacht wird). Was bedeutet das für die Implementierung in Haskell?
   (f) Fragen Sie, wenn Sie etwas nicht verstehen.

57. Eine *Multimenge* (engl. *multiset* oder *bag*) ist etwas Ähnliches wie eine Menge, außer dass Elemente auch mehrfach vorkommen dürfen. Die Reihenfolge spielt keine Rolle. Zum Beispiel ist $\{\!\{a,b\}\!\} \neq \{\!\{a,a,b\}\!\} = \{\!\{a,b,a\}\!\} \neq \{\!\{a,a,a,b\}\!\}$, für $a \neq b$.
   (a) Schreiben Sie eine Spezifikation für einen abstrakten Datentyp von Multimengen über der Grundmenge der ganzen Zahlen (`int`), die folgende Operationen unterstützt: Erzeugen einer leeren Multimenge; Einfügen und Streichen eines Elementes (dabei wird die Vielfachheit jeweils um 1 erhöht beziehungsweise erniedrigt); Feststellen der Vielfachheit eines Elementes.

Mathematisch Modellierende Spezifikation

Typen:
INT, NAT,
MULTI $= ((x_1,a_1),\dots,(x_n,a_n)), \quad \forall_{1\le i\le n} : x_i \in \mathbb{Z}, a_i \in \mathbb{N}\backslash\{0\}, \quad n \in \mathbb{N}, \quad \forall_{1\le i<j\le n} : x_i \neq x_j$

Signaturen:
leer: → MULTI
einfügen: MULTI × INT → MULTI
entfernen: MULTI × INT → MULTI
wieviel: MULTI × INT → NAT

Definitionen:
leer() = ()
einfügen$(((x_1,a_1),\dots,(x_n,a_n)),x)$

$$= \begin{cases} ((x_1,a_1),\dots,(x_n,a_n),(x,1)) & \text{falls } \forall_{1\le i\le n} : x_i \neq x \\ ((x_1,a_1),\dots,(x_i,a_i+1),\dots,(x_n,a_n)) & \text{mit } x_i = x \end{cases}$$

entfernen$(((x_1,a_1),\dots,(x_n,a_n)),x)$

$$= \begin{cases} \textit{undefiniert} & \text{falls } \forall_{1\le i\le n} : x_i \neq x \\ ((x_1,a_1),\dots,(x_i,a_i-1),\dots,(x_n,a_n)) & \text{falls } a_i > 1 \text{ mit } x_i = x \\ ((x_1,a_1),\dots,(x_{i-1},a_{i-1}),(x_{i+1},a_{i+1}),\dots,(x_n,a_n)) & \text{falls } a_i = 1 \text{ mit } x_i = x \end{cases}$$

$$\text{wieviel}(((x_1,a_1),\ldots,(x_n,a_n)),x) = \begin{cases} 0 & \text{falls } \forall_{1\leq i\leq n} : x_i \neq x \\ a_i & \text{mit } x_i = x \end{cases}$$

(b) Geben Sie eine konkrete Darstellung (etwa als Java-Klasse `Multimenge`) an. Beschreiben Sie die Abstraktionsfunktion, sowie die Invarianten, die die gültigen Darstellungen charakterisieren. Geben Sie auch die Vorbedingungen für alle Operatiomen an. (Sie dürfen dabei vernünftige Einschränkungen für die verfügbaren Operationen machen.)

Datentyp:

```
// maximale Anzahl verschiedener Elemente
static final int maxLänge = 100;

// aktuelle Anzahl verschiedener Elemente
int länge;

// Array zum Speichern der Elemente
int[] x;

// Array zum Speichern der Vielfachheiten
int[] a;
```

Invarianten für eine gültige Darstellung:

i. `0 <= länge <= maxLänge`
ii. `x.length == a.length == maxLänge`
iii. $\forall$`int i; 0<=i<länge` : `a[i] > 0`
iv. $\forall$`int i,j; 0<=i<j<=länge` : `x[i] != x[j]`

Methoden und Vorbedingungen:

- `Multimenge(); // Konstruktor für leer()`
  Keine Vorbedingungen.
- `void einfügen(int xx);`
  Vorbedingung: `länge < maxLänge ||` $\exists$`int i; 0<=i<maxLänge` : `x[i] == xx`
- `void entfernen(int xx);`
  Keine Vorbedingungen (falls Element nicht enthalten: Fehler gemäß Spezifikation).
- `int wieviel(int xx);`
  Keine Vorbedingungen.

Abstraktionsfunktion
Die Abstraktionsfunktion bildet eine Instanz der Java-Klasse `Multimenge` (dargestellt durch ein Tupel der Variablen) auf ein Element des Typs MULTI ab.

abst: `Multimenge` $\rightarrow$ MULTI,
`(länge,x,a)`$\mapsto$ ((`x[0]`, `a[0]`), …,(`x[länge-1]`, `a[länge-1]`))

Eine notwendige Eigenschaft für die Abstraktionsfunktion ist die Surjektivität. Eine Funktion f ist surjektiv, wenn es zu jedem Element y aus dem Bildraum ein Element x aus dem Definitionsraum gibt, so dass $f(x) = y$. Für die Abstraktionsfunktion bedeutet das, dass es zu jedem Element des mathematischen Modells (mindestens) eine entsprechende Instanz der implementierenden Klasse gibt.
Unsere vorliegende Abstraktionsfunktion ist surjektiv (mit der Einschränkung, dass die Implementation auf maximal 100 verschiedene Zahlen beschränkt ist). Sie ist jedoch nicht injektiv. Es gibt also mindestens ein Element des Modells, dem mehr als eine Instanz der Java-Klasse entspricht. Man vergleiche z.B. ein frisch erzeugtes Objekt mit einem, in das zuvor Elemente eingefügt, und dann wieder entfernt wurden. Im frisch erzeugten Objekt ist `länge` = 0 und auch alle Einträge in `x[]` und `a[]` sind = 0. Haben wir zuvor Elemente eingefügt und wieder gelöscht, gilt ebenfalls `länge` = 0, aber die Einträge im Array `x[]` werden nicht auf 0

zurückgesetzt. Beiden Objekten entspricht in der Spezifikation das leere Tupel, somit ist die Abstraktionsfunktion nicht injektiv. Ein entsprechendes Beispiel lässt sich für jedes Objekt mit `länge` $<$ `maxLänge` konstruieren. Würde man beim Löschen alle alten Einträge mit 0 überschreiben, wäre die Funktion injektiiv.

(c) Implementieren Sie die Operationen. Sie können von der in der Vorlesung besprochenen Implementierung für Mengen mit bis zu 100 Elementen[1] ausgehen.

```
/**
 * Eine Menge, in der Elemente mehrfach enthalten sein dürfen.
 */
class Multimenge {

  // maximale Anzahl verschiedener Elemente
  static final int maxLänge = 100;

  // aktuelle Anzahl verschiedener Elemente
  int länge;

  // Array zum Speichern der Elemente
  int[] x;

  // Array zum Speichern der Vielfachheiten
  int[] a;

  /**
   * Erzeugt eine leere Multimenge.
   */
  Multimenge() {
    länge = 0;
    x = new int[maxLänge];
    a = new int[maxLänge];
  }

  /**
   * Fügt ein Element in die Multimenge ein.
   * Falls die Multimenge voll ist,
   * wird eine RuntimeException ausgelöst.
   */
  void einfügen(int xx) {
    // Prüfe, ob das Element bereits enthalten ist.
    for (int i=0; i<länge; ++i) {
      if (x[i] == xx) {
        // Fall 1: Das Element ist bereits enthalten.
        // wir erhöhen seine Vielfachheit.
        a[i]++;
        return;
      }
    }
    // Das Element ist nicht enthalten.
    if (länge == maxLänge) {
      // Kein Platz frei. Fehler.
      throw new RuntimeException("Kann nicht einfügen. Voll.");
    }
    // noch ein Platz frei. Füge das neue Element
    // an das Ende des Arrays an und passe die länge an.
    x[länge] = xx;
    a[länge++] = 1;
```

[1] http://www.inf.fu-berlin.de/~rote/Lere/2003-04-WS/Algorithmen+Programmierung3/Menge.java

```
  }

  /**
   * entfernt ein Element aus der Multimenge.
   * Wenn das Element nicht enthalten ist,
   * wird eine RuntimeException ausgelöst.
   */
  void entfernen(int xx) {
    // Prüfe, ob das Element enthalten ist.
    for (int i=0; i<länge; ++i) {
      if (x[i] == xx) {
        // das Element ist enthalten. Verringere die Vielfachheit
        if (--a[i] == 0) {
          // letztes x wurde entfernt. Verschiebe nachfolgende x nach links.
          for (int j=i; j<länge-1; ++j) {
            a[j] = a[j+1];
            x[j] = x[j+1];
          }
          // Passe die Länge an.
          länge--;
        }
        return;
      }
    }
    // Element ist nicht enthalten. Fehler.
    throw new RuntimeException("Element nicht enthalten.");
  }

  /**
   * Bestimmt, wie oft das Element enthalten ist.
   */
  int wieviel (int xx) {
    // suche das Element.
    for (int i=0; i<länge; ++i) {
      if (x[i] == xx) {
        // Element gefunden. Gib seine Vielfachheit zurück.
        return a[i];
      }
    }
    // Element ist nicht enthalten. Rückgabe 0.
    return 0;
  }
}
```

(d) Beweisen Sie die Korrektheit Ihrer Implementierung.

Korrektheit der Invarianten für eine gültige Darstellung

- `Multimenge()`
  i. Erfüllt, da `länge=0`.
  ii. Die Arrays werden entsprechend initialisiert.
  iii. Kein `i`, für das es gelten müsste.
  iv. Keine `i`,`j` für das es gelten müsste.
- `void einfügen(int xx)`
  i. – `länge>=0` – klar, da `länge` nicht verringert wird.
     – `länge<=maxLänge` – erfüllt, da `länge` nur um eins erhöht wird, wenn `länge<maxLänge`
  ii. Längen der Arrays bleiben unverändert.
  iii. `a[i]` wird erhöht und `länge` bleibt konstant (also kein neues `i` für das `a[i]>0` gelten muss), oder `länge` wird um 1 erhöht und `a[länge-1]=1 > 0`.

iv. Falls `x[i]==xx` für ein `i` oder `länge == maxLänge`, dann werden `x` und `länge` nicht verändert. Ansonsten ist `xx` ein neues Element und von allen Einträgen in `x` verschieden. Dann wird `länge` um 1 erhöht und `xx` als neues Element in `x` aufgenommen.

- `void entfernen(int xx)`
  i. – `länge>=0` – die for-Schleife, in der `länge` dekrementiert wird, wird nur betreten, falls `länge>0`.
     – `länge>=maxLänge` – klar, da `länge` nicht vergrößert wird.
  ii. bleibt unverändert.
  iii. Mit `if (--a[i]==0)` wird diese Invariante explizit geprüft. Falls sie nicht erfüllt sein sollte, wird der obere Teil von `x` nach links verschoben und `länge` entsprechend angepasst.
  iv. Ist erfüllt, da kein neuer Wert in das Array `x` eingefügt wird.
- `int wieviel(int xx)`

  Ist klar, da dies eine sondierende Funktion ist und keine Änderungen am Objekt vorgenommen werden.

Übereinstimmung mit der Spezifikation

- zz: leer() = abst(`new Multimenge()`)

  Beweis:
  L.S. leer() = () R.S. abst(`new Multimenge()`) = abst(`(0,[],[])`) = ()

  $\square$

- zz: $\forall$ `Multimenge mm, int x` : einfügen(abst(mm),x) = abst(`mm.einfügen(x)`)
  (wobei `mm` den Vorbedingungen für `einfügen` entspricht.)

  Beweis:
  L.S. einfügen(abst(`(l,x[],a[])`),y) = einfügen(((`x[0]`,`a[0]`),...,(`x[l-1]`,`a[l-1]`)),y)

$$= \begin{cases} ((\texttt{x[0]},\texttt{a[0]}),\ldots,(\texttt{x[l-1]},\texttt{a[l-1]}),(y,1)) & \text{falls } \forall_{0\le \texttt{i}\le \texttt{l-1}} : \texttt{x[i]} \neq y \\ ((\texttt{x[0]},\texttt{a[0]}),\ldots,(\texttt{x[i]},\texttt{a[i]+1}),\ldots,(\texttt{x[l-1]},\texttt{a[l-1]})) & \text{mit } \texttt{x[i]} = y \end{cases}$$

  R.S. abst(`(l,x[],a[]).einfügen(y)`)

$$= \begin{cases} \text{abst}(\texttt{RuntimeException}) & \text{falls } \forall_{0\le \texttt{i}\le \texttt{l-1}} : \texttt{x[i]} \neq \texttt{y} \land \texttt{l} \geq \texttt{maxLänge} \\ & \text{(}\texttt{mm}\text{ erfüllt die Vorbedingungen} \\ & \text{nicht, also unmöglicher Fall.)} \\ \text{abst}((\texttt{l+1},\texttt{x'[]},\texttt{a'[]})), & \text{falls } \forall_{0\le \texttt{i}\le \texttt{l-1}} : \texttt{x[i]} \neq \texttt{y} \land \texttt{l} < \texttt{maxLänge} \\ & \text{mit } \texttt{x'[]=x[]} \text{ und } \texttt{a'[]=a[]}, \\ & \text{wobei } \texttt{x'[l]=y} \text{ und } \texttt{a'[l]=1} \\ \text{abst}((\texttt{l},\texttt{x[]},\texttt{a'[]})), & \text{mit } \texttt{x[i]} = \texttt{y} \\ & \text{und } \texttt{a'[]=a[]}, \text{ wobei } \texttt{a'[i]=a[i]+1} \end{cases}$$

$$= \begin{cases} ((\texttt{x[0]},\texttt{a[0]}),\ldots,(\texttt{x[l-1]},\texttt{a[l-1]}),(\texttt{y},1)) & \text{falls } \forall_{0\le \texttt{i}\le \texttt{l-1}} : \texttt{x[i]} \neq \texttt{y} \\ ((\texttt{x[0]},\texttt{a[0]}),\ldots,(\texttt{x[i]},\texttt{a[i]+1}),\ldots,(\texttt{x[l-1]},\texttt{a[l-1]})) & \text{mit } \texttt{x[i]} = \texttt{y} \end{cases}$$

  $\square$

  Anmerkung: Das y der linken Seite ist *kursiv*, und das `y` der rechten Seite `monospaced`. Eigentlich bräuchte man noch eine Abstraktionsfunktion vom Computer-Int auf die ganzen Zahlen, die das umwandelt.

- zz: $\forall$ `Multimenge mm, int x` : entfernen(abst(mm),x) = abst(`mm.entfernen(x)`)
  Beweis: eingespart
- zz: $\forall$ `Multimenge mm, int x` : wieviel(abst(mm),x) = abst(`mm.wieviel(x)`)
  Beweis: eingespart

58. Das gleiche über eine Algebraische Spezifikation.

(a) Algebraische Spezifikation

Typen: MULTI, INT, NAT

Operationen:
leer: $\rightarrow$ MULTI
einfügen: MULTI $\times$ INT $\rightarrow$ MULTI
entfernen: MULTI $\times$ INT $\rightarrow$ MULTI
wieviel: MULTI $\times$ INT $\rightarrow$ NAT

Axiome:
wieviel(leer(), x) $= 0$

$$\text{wieviel(einfügen}(M, x), y) = \begin{cases} \text{plus}(1, \text{wieviel}(M, y)) & \text{für } x = y \\ \text{wieviel}(M, y) & \text{für } x \neq y \end{cases}$$

$$\text{wieviel(entfernen}(M, x), y) = \begin{cases} \max(0, \text{plus}(-1, \text{wieviel}(M, y))) & \text{für } x = y \\ \text{wieviel}(M, y) & \text{für } x \neq y \end{cases}$$

In der algebraischen Spezifikation eines Datentypen geben wir kein Modell an, sondern spezifizieren den Datentypen durch Operationen. Bei den Operationen unterscheiden wir zwischen *erzeugenden*, *verändernden* und *sondierenden* Operationen. Erzeugende Operationen erzeugen ein Element des Datentyps, verändernde Operationen verändern es und sondierende Operationen liefern uns seine Eigenschaften. Die Operationen werden durch *Axiome* spezifiziert, und zwar die erzeugenden und verändernden mit Hilfe der sondierenden.

Zwei Eigenschaften sind für eine algebraische Spezifikation wichtig: Sie muss *vollständig* und *widerspruchsfrei* sein. Vollständigkeit bedeutet, dass jede mögliche Folge von Operationen spezifiziert ist. Widerspruchsfreiheit bedeutet, dass sich aus einer Folge nicht zwei verschiedene Ergebnisse ableiten lassen.

In der vorliegenden Spezifikation ist `leer` eine erzeugende Operation, sind `einfügen` und `entfernen` verändernde Operationen und `wieviel` eine sondierende Operation. Die Spezifikation ist vollständig, da jede Multimenge als Eingabe für `wieviel` (dies ist die einzige sondierende Funktion) auf `leer` abgeleitet werden kann. Die Spezifikation ist widerspruchsfrei, da es für jeden Fall genau eine mögliche Ableitung gibt (hinreichende Bedingung).

Anmerkung: Diese Spezifikation ist nicht äquivalent zu der modellierenden aus dem ersten Teil. Dort war der Aufruf von entfernen (leer, x) undefiniert und wurde mit dem Auslösen eines Fehlers implementiert. Hier wird dieser Aufruf einfach ignoriert. Möchte man die algebraische Spezifikation äquivalent zur modellierenden machen, müsste man noch einen dritten Fall einfügen anstatt das Maximum zu nehmen. Möchte man die modellierende Spezifikation äquivalent zur algebraischen machen, gibt man im entsprechenden Fall einfach die Eingabe unverändert zurück.

(b) Implementation in Haskell

```
data Multi = Leer | Einfügen Int Multi | Entfernen Int Multi

wieviel :: Multi -> Int -> Int
wieviel Leer x = 0
wieviel (Einfügen x m) y | x == y    = 1 + wieviel m y
                         | otherwise = wieviel m y

wieviel (Entfernen x m) y | x == y    = max 0 (wieviel m y - 1)
                          | otherwise = wieviel m y
```

(c) Korrektheit der Implementation

Die Korrektheit der Implementation beweist man wie im ersten Fall in zwei Schritten. Zuerst zeigt man, dass die Invarianten für eine gültige Darstelung erfüllt sind. In unserer Implementierung gibt es keine solche Invarianten, also ist nichts zu zeigen.

Anschließend zeigt man, dass die Implementation der Spezifikation entspricht. Für die modellierende Spezifikation haben wir zu diesem Zweck eine Abstraktionsfunktion definiert, die jedem `Multimenge`-Objekt ein Modell vom Typ MULTI zuordnet. Für eine Algebraische Spezifikation ist es nicht möglich, eine Abstraktionsfunktion zu definieren – auf was sollte man auch abbilden, es gibt ja kein Modell. Eine Abstraktionsfunktion ist aber auch nicht nötig, denn wir haben ja die Axiome.

Um die Übereinstimmung einer Implementation mit einer algebraischen Spezifikation nachzuweisen, gehen wir die Axiome der Reihe nach durch. Jedes einzelne Axiom muss für jede beliebige Eingabe gelten. Als Beweisverfahren bietet sich hierzu die strukturelle Induktion an. In unserer Haskell-Implementierung ist nichts zu zeigen, da sie sich nur syntaktisch von der Spezifikation unterscheidet. Interessanter wäre es, den Beweis für die Java-Implementierung zu führen. Dies sei dem Leser überlassen.

Das bekannteste Zitat zum Thema Spezifikation stammt von Donald Knuth:

I have proven that this program is correct, but I haven't tested it.