Vorlesung
Praktische Informatik I
Wintersemester 2002/03

**Abstraktionen**

Stefan Bisanz, Ulrich Hannemann und Jan Peleska

# Abstraktionsfunktionen $\rho$ für Listen

## Abstrakte Listenrepräsentation:

Wir benutzen die in der Vorlesung zu Listen eingeführte Notation – siehe

`http://www.informatik.uni-bremen.de/agbs/lehre/ws0203/pi1/hintergrund-info/java/ListHandler.java`

Listen werden abstrakt als endliche Folgen über Elementen aus Menge *MyDataType* angesehen. Da wir die Elemente einer Liste sequenziell bearbeiten möchten, müssen wir ein *aktuelles Element* kennzeichnen, welches als nächstes zu lesen oder zu löschen ist, oder vor dem ein neues Element einzufügen ist. Hierzu betrachten wir jede Liste als aus zwei Teilen ("part1, part2" in der Vorlesung) zusammengesetzt. Das aktuelle Element ist gerade der Kopf des zweiten Teils.

$$M = MyDataType^* \times MyDataType^* \times MyDataType^*$$

$$L_{abs} = \{\ (l_1, l_2, l_3) \in M \mid l_1 = l_2 \hat{\ } l_3\}$$

Hier bezeichnet $l_1$ die Gesamtliste, $l_2$ entspricht `part1(l1)` aus der Vorlesung, $l_3$ entspricht `part2(l1)`.

## Konkrete Listenrepräsentation:

(Hier bezeichnet z.B. *LIST* die Menge der Referenzen $l$ auf ein Java-Objekt vom Typ `LIST` vereinigt mit `null`.)

$$K = LIST \times Link^*$$

$$\begin{aligned} L_{konkret} = \{\ & (l, < m_0, \ldots, m_{n+1} >) \in K \mid \\ & n \geq 0 \wedge l.first \neq l.last \wedge \\ & l.first.data = null \wedge l.last.data = null \wedge \\ & m_0 = l.first \wedge m_{n+1} = l.last \wedge m_n = l.last.next \wedge \\ & l.predActPtr \in \{m_0, \ldots, m_n\} \wedge \\ & (\forall i \in \{0, \ldots, n\} \cdot m_i.next = m_{i+1}) \wedge \\ & (\forall i \in \{1, \ldots, n\} \cdot m_i.data \neq null)\ \} \end{aligned}$$

Die für Elemente von $L_{konkret}$[1] sogenannten **Integritätsbedingungen** enthalten alle Forderungen, die wir für eine korrekt gebildete Listen-Datenstruktur in Java benötigen:

[1] Außerdem gilt implizit $l \neq null \wedge \forall i \in \{0, \ldots, n+1\} \cdot m_i \neq null$, d.h., alle Referenzen zeigen auf Objekte.

- Es sind immer mindestens die Elemente $l.first$ und $l.last$ in der *Link*-Verkettung $< m_0, \ldots, m_{n+1} >$ vorhanden.
- Alle *Link*-Elemente sind miteinander über *next* verkettet, das letzte Element $m_{n+1}$ hat das vorletzte $m_n$ als Nachfolger.
- Die Referenz *l.predActPtr* auf den Vorgänger des aktuellen Elementes zeigt immer auf eines der $m_0, \ldots, m_n$. Hieraus folgt für $n = 0$, dass *l.predActPtr* auf $l.first$ zeigt.
- Nur die *Link*-Elemente $< m_1, \ldots, m_n >$ enthalten Nutzdaten; es muss hierzu $n \geq 1$ gelten.
- Wir werden weiter unten sehen, dass $(l, < m_0, \ldots, m_{n+1} >)$ für $n = 0$ auf die leere abstrakte Liste abgebildet wird.

Die Bedingungen für Elemente aus $L_{konkret}$ müssen *vor* und *nach* einer jeden Listenoperation erfüllt sein. Solche Bedingungen nennt man **Invarianten**.

### Zwei Hilfsfunktionen

Wir führen zwei Hilfsfunktionen $\varphi$ und $\psi$ ein, die aus einer konkreten Listenrepräsentation in eine abstrakte Liste abbilden, in $part1$ bzw. $part2$.

$$\varphi : L_{konkret} \mapsto MyDataType^*$$

$$\varphi((l, < m_0, \ldots, m_{n+1} >)) = \begin{cases} <> & \text{falls } l.predActPtr = l.first \\ < m_1.data, \ldots, m_k.data > & \text{falls } l.predActPtr = m_k \end{cases}$$

für $k \in \{1, \ldots, n\}$.

$$\psi : L_{konkret} \mapsto MyDataType^*$$

$$\psi((l, < m_0, \ldots, m_{n+1} >)) = \begin{cases} <> & \text{falls } l.predActPtr = m_n \\ < m_{k+1}.data, \ldots, m_n.data > & \text{falls } l.predActPtr = m_k \end{cases}$$

für $k \in \{0, \ldots, n-1\}$.

$\varphi((l, < m_0, \ldots, m_{n+1} >))$ – d. h., der erste, bereits bearbeitete Listenteil – ist immer leer wenn *l.predActPtr* auf den Listenkopf $l.first$ zeigt. $\psi((l, < m_0, \ldots, m_{n+1} >))$ – d. h., der zweite, noch zu untersuchende Listenteil – ist genau dann leer, wenn *l.predActPtr* auf das letzte Nutzdaten enthaltende *Link*-Element, also $m_n$, zeigt, bzw. wenn keine Nutzdaten vorhanden sind ($m_n = m_0$).

### Abstraktionsfunktion

Wir führen jetzt die **Abstraktionsfunktion** $\rho$ ein, welche die konkrete Java-Datenstruktur $(l, < m_0, \ldots, m_{n+1} >)$ auf die von ihr repräsentierte abstrakte Liste $(l_1, l_2, l_3)$ abbildet.

$$\rho : L_{konkret} \mapsto L_{abs}$$

$$\rho(l, < m_0, \ldots, m_{n+1} >)) = \begin{cases} (<>, <>, <>) & \text{für } n = 0 \\ (< m_1.data, \ldots, m_n.data >, & \\ \varphi((l, < m_0, \ldots, m_{n+1} >)), & \\ \psi((l, < m_0, \ldots, m_{n+1} >))) & \text{für } n > 0 \end{cases}$$

## Bemerkungen

Jetzt haben wir unser *abstrakten* Operationen durch Pre- und Postconditions definiert, etwa:

```
//PRE: l in MyDataType* and l = part1(l)^part2(l)

      void lstAppend(MyDataType x, LIST l)

//POST:    l = part1(l)^part2(l) and part1(l) = (part1(l)@PRE)
//         and part2(l) = (part2(l)@PRE)^<x> AND x = x@PRE
```

und zeigen, daß die konkrete Operation eine korrekte Implementation ist. Hierzu benutzen wir bei der konkreten Operation auch Pre-/Postconditions über die *konkreten* Datenrepräsentation aus $L_{konkret}$:

```
//PRE: l_konkret = (l,<m0,m1,...,m_n,m_n+1>) and l_konkret in L_konkret
  public static void lstAppend(MyDataType x, LIST l) {

      Link newElement = new Link();
      newElement.data = x;
      newElement.next = l.last;
      l.last.next.next = newElement;
      l.last.next = newElement;

  }
  //POST: EXISTS newElement in Link .
            (l_konkret = (l,<m0,m1,...,m_n,newElement,m_n+1>)
               and l_konkret in L_konkret
               and newElement.data = x
               and (FORALL i in {1,...,n} . m_i.data = (m_i.data)@PRE)
```

```
and l.predActPtr = (l.predActPtr)@PRE)
```

Die konkreten Pre-/Postconditions drücken folgende Eigenschaften aus:

- Wir erwarten, dass die Java-Datenstruktur `l_konkret` *vor* Methodenaufruf wohlgeformt ist. Dies ist durch die Forderung `l_konkret in L_konkret` vollständig ausgedrückt.
- Wir erwarten, dass auch nach der Operation noch wohlgeformt ist – dies wird wieder durch `l_konkret in L_konkret` ausgedrückt.
- Das neue Element soll das *letzte* Nutzdaten tragende Element der *Link*-Verkettung sein. Das ist durch `l_konkret = (l,<m0,m1,...,m_n,newElement,m_n+1>)` ausgedrückt. Aus `l_konkret in L_konkret` folgt automatisch, dass die Verkettung wieder korrekt sein muss.
- Das neue *Link*-Element enthält die Referenz auf die einzufügenden Nutzdaten: `newElement.data = x`.
- Die vorher in der Liste eingetragenen Nutzdaten bleiben unverändert: `FORALL i in {1,...,n} . m_i.data = (m_i.data)@PRE`.
- Der Zeiger auf den Vorgänger des aktuellen Elementes bleibt unverändert: `l.predActPtr = (l.predActPtr)@PRE`.

Dass die Java-Implementierung `public static void lstAppend(MyDataType x, LIST l)` tatsächlich bei Vorliegen der Precondition auch die Postcondition erfüllt, muss noch bewiesen werden. Die hierzu erforderlichen Techniken werden im nächsten Semester erläutert. Was wir tatsächlich schon beweisen können, ist, dass die konkrete Append-Operation tatsächlich die abstrakte Operation korrekt implementiert, wenn sie die oben angegebenen konkreten Pre-/Postconditions erfüllt. Nehmen wir hierzu eine konkrete Listenrepräsentation

$$l_{konkret} = (l, < m_0, m_1, ..., m_n, m_{n+1} >) \in L_{konkret}$$

und ihre abstrakte Repräsentation

$$\begin{aligned} l_{abs} &= \rho(l_{konkret}) = (< m_1.data, ..., m_n.data >, \varphi(l_{konkret}), \psi(l_{konkret})) \\ &= (< m_1.data, ..., m_n.data >, < m_1.data, ..., m_k.data >, < m_{k+1}.data, ..., m_n.data >) \end{aligned}$$

Die *abstrakte* Append-Operation `lstAppend(x,l_{abs})` erzeugt nach Spezifikation die Nachbedingung

$$l_{abs} = (< m_1.data, ..., m_n.data, x >, < m_1.data, ..., m_k.data >, < m_{k+1}.data, ..., m_n.data, x >)$$

Die *konkrete* Append-Operation erzeugt die veränderte konkrete Java-Datenstruktur

$$l_{konkret} = (l, < m_0, m_1, ..., m_n, newElement, m_{n+1} >)$$

Die konkrete Postcondition sichert zu, dass sich $l.predActPtr$ nicht verändert hat, Aus Definition von $\varphi$ folgt hieraus

$$\varphi(l_{konkret}) = \varphi(l_{konkret}@PRE) =< m_1.data, ..., m_k.data >$$

Der zweite Teil von $l_{konkret}$ ist laut Postcondition genau um $newElement$ verlängert worden. Nach der Ausführung der konkreten Append-Operation gilt also wegen $newElement.data = x$

$$\psi(l_{konkret}) = (\psi(l_{konkret})@PRE)\hat{}< x >=< m_{k+1}.data, ..., m_n.data, x >$$

Infolgedessen gilt nach der konkreten Append-Operation

$$\begin{aligned}
&\rho(l_{konkret}) \\
&\quad = (< m_1.data, ..., m_n.data, x >, \varphi(l_{konkret}), \psi(l_{konkret})) \\
&\quad = (< m_1.data, ..., m_n.data, x >, < m_1.data, ..., m_k.data >, < m_{k+1}.data, ..., m_n.data, x >)
\end{aligned}$$

Damit stimmt $\rho(l_{konkret})$ mit dem Zustand der abstrakten Liste $l_{abs}$ überein, die durch die abstrakte Append-Operation erzeugt wurde.

Nitpick: Für die `lstNext`- Operation und der Rückgabe der `null`-Referenz ist *MyDataType* (konkret) genaugenommen eine Teilmenge der Referenzen, einschließlich `null`, die dann nach $(Key \times Name \times Salary) \cup \{\perp\}$ (abstrakt) abbildet.