Mathematische Modellierung, operationelle Semantik und Verifikation von Listen in Java

Jan Peleska
Universität Bremen
FB 3 - Informatik
AG Betriebssysteme und verteilte Systeme
jp@tzi.de

29. Januar 2007

Zusammenfassung

Wir beschreiben ein mathematisches Modell für den abstrakten Datentyp der Listen. Auf diesem Modell werden die typischen Operationen zum sinnvollen Gebrauch von Listen zunächst abstrakt als mathematische Abbildungen definiert. Am Beispiel der einfach verketteten Listen in Java werden dann konkrete Listen durch eine Abstraktionsfunktion ihrem mathematischen Gegenstück zugeordnet. Die bereits eingeführte operationelle Java-Semantik für atomare Datentypen, Zuweisungen und Kontrollstrukturen wird um rekursive Datentypen und den new-Operator erweitert, so dass sich damit die operationelle Semantik konkreter Listenoperationen bestimmen lässt. Am Beispiel der `delete()`-Operation demonstrieren wir, wie man einen Verfeinerungsbeweis der Art *"konkrete Listenoperation ist korrekte Implementierung der zugehörigen abstrakten Operation"* durchführt.

Inhaltsverzeichnis

1 Endliche Folgen

Für eine gegebene Menge T (wir bezeichnen diese im folgenden als *Datentyp*) definiert

$$T^* = \{\langle\,\rangle\} \cup \{f : \{1, \ldots, n\} \longrightarrow T \mid n \in \mathbb{N}\}$$

die Menge aller *endlichen Folgen* über T. Dabei bezeichnet $\langle\,\rangle$ die *leere Folge*, formal mit der leeren Abbildung identifizierbar. Der Definitionsbereich einer nicht leeren Folge f – diesen bezeichnet man üblicherweise mit $dom(f)$, also $dom(f) = \{1, \ldots, n\}$ in der obigen Definition – ist also immer ein endlicher, bei 1 beginnender Abschnitt der natürlichen Zahlen. Die Länge einer Folge f wird mit $\#f$ bezeichnet und ist durch die Kardinalität des Definitionsbereichs – also n in der obigen Notation – gegeben:

$$\#f = card(dom(f))$$

Anstelle der Funktionsnotation wird – da der Definitionsbereich immer der Abschnitt $1, \ldots,$ *"Länge der Folge"* der natürlichen Zahlen ist – häufig die Aufzählung der Bildwerte unter f gewählt: Ist $f(1) = y_1, \ldots, f(n) = y_n$, notieren wir die Folge f durch

$$f = \langle y_1, \ldots, y_n \rangle$$

Mit $f_1 \frown f_2$ wird die *Konkatenation* der Folgen $f_i : \{1, \ldots n_i\} \longrightarrow T, i = 1, 2$ bezeichnet:

$$dom(f_1 \frown f_2) = \{1, \ldots, n_1 + n_2\}$$
$$(f_1 \frown f_2)(i) = \begin{cases} f_1(i) & \text{falls } i \in \{1, \ldots, n_1\} \\ f_2(i - n_1) & \text{falls } i \in \{n_1 + 1, \ldots, n_1 + n_2\} \end{cases}$$

Bei einer nicht leeren Folge f bezeichnet $head(f) = f(1)$ das erste Element und $tail(f) = \langle f(2), \ldots, f(\#f) \rangle$ die Folge, welche aus f durch Entfernen des ersten Elementes entsteht. Offenbar gilt für nicht leere Folgen f

$$f = \langle head(f) \rangle \frown tail(f)$$

Die Menge aller in der Folge f enthaltenen Elemente (also das Bild unter der Funktion f) bezeichnen wir mit $ran(f)$ *("Range of function f")*. Beispielsweise ist

$$ran(\langle a, a, a, b, c, c, d, e \rangle) = \{a, b, c, d, e\}$$

Da Elemente mehrfach in einer Folge auftreten duerfen, ist die Kardinalität des Bildbereichs kleiner oder gleich der Folgenlänge:

$$card(ran(f)) \leq \#f$$

2 Ein mathematisches Listenmodell

In mathematischen Einführungen werden Listen häufig einfach mit endlichen Folgen gleichgesetzt. Für die leicht verständliche Einführung vieler Operationen auf Listen, welche deren Elemente nacheinander, der Listensortierung folgend, bearbeiten, ist es jedoch sinnvoll, eine Liste als *ein Folgenpaar* (f_1, f_2) darzustellen: Folge f_1 bezeichnet dabei den *"bereits bearbeiteten Teil der Liste"*, Folge f_2 den *"noch zu bearbeitenden Teil der Liste"*. Das *als nächstes zu bearbeitende Element"* der Liste ist $f_2(1) = \mathrm{head}(f_2)$. Nach der Bearbeitung dieses Elementes ist dann $\mathrm{tail}(f_2)$ der weiterhin zu bearbeitende Rest. Dies führt uns dazu, die *Menge aller Listen über dem Datentyp* T als

$$\mathrm{List}(T) = T^* \times T^*$$

zu definieren. Mit diesem mathematische Modell werden jetzt die typischen Listenoperationen als Funktionen eingeführt. Um diese von den konkreten im unten genannten Java-Beispielprogramm definierten Methoden zu unterscheiden, benutzen wir für diese Funktionen die Namenskonvention $\mathrm{name}_A()$, wobei das "A" für "abstrakte Funktion" steht.

Die *Append-Funktion* hängt ein neues Element an eine gegebene Liste an. Dies ändert nichts an dem bereits bearbeiteten Listenteil:

$$\begin{aligned} &\mathrm{append}_A : \mathrm{List}(T) \times T \longrightarrow \mathrm{List}(T); \\ &((f_1, f_2), x) \mapsto (f_1, f_2 \frown \langle x \rangle) \end{aligned}$$

Die *Insert-Funktion* fügt ein neues Element x *am Ende des bearbeiteten Teils* ein. Ist bei Liste (f_1, f_2) der noch zu bearbeitende Teil nicht leer $(f_2 \neq \langle\ \rangle)$, wird x also *vor* dem aktuell zu bearbeitenden eingefügt; das aktuell zu bearbeitende Element bleibt unverändert:

$$\begin{aligned} &\mathrm{insert}_A : \mathrm{List}(T) \times T \longrightarrow \mathrm{List}(T); \\ &((f_1, f_2), x) \mapsto (f_1 \frown \langle x \rangle, f_2) \end{aligned}$$

Funktion *length* ermittelt die Länge einer Liste:

$$\begin{aligned} &\mathrm{length}_A : \mathrm{List}(T) \longrightarrow \mathbb{N} \\ &(f_1, f_2) \mapsto (\#f_1 + \#f_2) \end{aligned}$$

Funtion *isEmpty* prüft, ob die Liste leer ist:

$$\begin{aligned} &\mathrm{isEmpty}_A : \mathrm{List}(T) \longrightarrow \mathbb{B} \\ &(f_1, f_2) \mapsto (\mathrm{length}_A(f_1, f_2) > 0) \end{aligned}$$

Funktion *haveNext* prüft, ob es noch ein aktuelles zu bearbeitendes Element gibt:

$$\begin{aligned} &\mathrm{haveNext}_A : \mathrm{List}(T) \longrightarrow \mathbb{B} \\ &(f_1, f_2) \mapsto (\#f_2 > 0) \end{aligned}$$

Mit der *rewind*-Funktion wird der noch zu bearbeitende Teil einer Liste wieder auf den vollständigen Listeninhalt gesetzt:

$$\begin{aligned}&\mathrm{rewind}_A : \mathrm{List}(T) \longrightarrow \mathrm{List}(T);\\&(f_1, f_2) \mapsto (\langle\,\rangle, f_1 \frown f_2)\end{aligned}$$

Die *read-Funktion* ist partiell auf allen Listen definiert, deren noch zu bearbeitender Teil nicht leer ist. Sie gibt den Wert des aktuellen Elements x zurück. Der noch zu bearbeitende Teil wird hierdurch nicht verändert:

$$\begin{aligned}&\mathrm{read}_A : \{(f_1, f_2) \in \mathrm{List}(T) \mid f_2 \neq \langle\,\rangle\} \longrightarrow T;\\&(f_1, f_2) \mapsto \mathrm{head}(f_2)\end{aligned}$$

Die *readNext-Funktion* ist partiell auf allen Listen definiert, deren noch zu bearbeitender Teil nicht leer ist. Sie gibt den Wert des aktuellen Elements x zurück und setzt den noch zu bearbeitenden Teil auf den Listenrest hinter x:

$$\begin{aligned}&\mathrm{readNext}_A : \{(f_1, f_2) \in \mathrm{List}(T) \mid f_2 \neq \langle\,\rangle\} \longrightarrow \mathrm{List}(T) \times T;\\&(f_1, f_2) \mapsto ((f_1 \frown \langle \mathrm{head}(f_2)\rangle, \mathrm{tail}(f_2)), \mathrm{head}(f_2))\end{aligned}$$

Die *delete-Funktion* ist ebenso partiell definiert, denn sie löscht den Kopf des noch zu bearbeitenden Teils:

$$\begin{aligned}&\mathrm{delete}_A : \{(f_1, f_2) \in \mathrm{List}(T) \mid f_2 \neq \langle\,\rangle\} \longrightarrow \mathrm{List}(T);\\&(f_1, f_2) \mapsto (f_1, \mathrm{tail}(f_2))\end{aligned}$$

Die *cat-Funktion* konkateniert zwei Listen. Dabei wird der bereits bearbeitete Teil der ersten Liste auch der bearbeitete Teil der neuen Liste:

$$\begin{aligned}&\mathrm{cat}_A : \mathrm{List}(T) \times \mathrm{List}(T) \longrightarrow \mathrm{List}(T);\\&((f_1, f_2), (g_1, g_2)) \mapsto (f_1, f_2 \frown g_1 \frown g_2)\end{aligned}$$

Die *push*-Operation fügt ein Element an den Listenanfang, d. h. *vor* einem bei nicht leerer Liste vorhandenen ersten Element ein. Diese Wirkung ist äquivalent zu der von *insert*, wenn der schon bearbeitete Teil leer ist. Wir können *push* daher durch *rewind* und *insert* definieren:

$$\begin{aligned}&\mathrm{push}_A : \mathrm{List}(T) \times T \longrightarrow \mathrm{List}(T);\\&((f_1, f_2), x) \mapsto \mathrm{insert}_A(\mathrm{rewind}_A(f_1, f_2), x)\end{aligned}$$

Funktion *top* gibt das erste Element einer nicht leeren Liste zurück, ohne den noch zu bearbeitenden Teil zu verändern; dies lässt sich offensichtlich durch *rewind* und *read* definieren:

$$\begin{aligned}&\mathrm{top}_A : \{(f_1, f_2) \in \mathrm{List}(T) \mid \neg\mathrm{isEmpty}(f_1, f_2)\} \longrightarrow T;\\&(f_1, f_2) \mapsto \mathrm{read}_A(\mathrm{rewind}_A(f_1, f_2))\end{aligned}$$

Funktion *pop* hat den selben Rückgabewert wie *top*, löscht aber gleichzeitig das gelesene Element aus der Liste:

$$\begin{aligned}&\mathrm{pop}_A : \{(f_1, f_2) \in \mathrm{List}(T) \mid \neg\mathrm{isEmpty}(f_1, f_2)\} \longrightarrow \mathrm{List}(T) \times T;\\&(f_1, f_2) \mapsto ((\langle\,\rangle, \mathrm{tail}(f_1 \frown f_2)), \mathrm{read}_A(\mathrm{rewind}_A(f_1, f_2)))\end{aligned}$$

Hinweis: Listen, auf denen nur mit den Funktionen *push, pop, top* operiert wird, heissen auch *Stapel (eng. Stacks)*.

3 Verkettete Listen in Java und ihre Operationelle Semantik

Ziel dieses Abschnitts ist, die bereits eingeführte operationelle Java-Sematik für atomare Datentypen, Zuweisungen und Kontrollstrukturen auf rekursive Datentypen zu erweitern, so dass sich die Semantik von Listenoperationen, in denen Java-Referenzen auf Nachfolger- und ggf. Vorgängerelemente verwendet werden erklären lässt. Die illustrierenden Beispiele in diesem Abschnitt beziehen sich auf das Programm

`http://www.informatik.uni-bremen.de/agbs/lehre/ws0607/`
`./pi1/hintergrund/listen/MyList.java`

das einfach verkettete Listen und zugehörige Operationen realisiert.

Zur Erinnerung: Ein *Programmzustand* ist eine partielle Abbildung

$$\sigma : V \not\longrightarrow D$$

die jedem im aktuellen Scope definierten Variablensymbol $x \in V$ einen Wert $\sigma(x) \in D(x)$ zuordnet, wobei $D(x)$ der Datentyp ist, welcher x bei seiner Deklaration zugeordnet wurde. Menge D ist die Vereinigung aller dieser Datentypen, zusammen mit dem Symbol $\perp$, welches den undefinierten Zustand einer deklarierten Variable darstellt, der noch kein Wert zugewiesen wurde.

Wie in der Vorlesung eingeführt, werden verkettete Listen mit Hilfe *rekursiver Datentypen* eingeführt; in Java sind dies Klassen, die wiederum Komponenten (in Java *Feldelemente* genannt) vom selben Klassentyp enthalten. Betrachten wir hierzu die Deklaration für einfach verkettete Listenelemente aus o. g. Beispielprogramm:

```
class Node {
  // Referenz auf das Listenelement:
  Object  data;
  // Rekursiver Verweis auf ein Element vom
  // Typ class Node, d.h. Verweis
  // auf das nächste Listenelement.
  Node next;
}
```

Feldelement `next` ist eine *Referenz* auf ein Element des gerade deklarierten Klassentyps `Node`. Eine Deklaration

```
Node n;
```

auf dem Stack der gerade ausgeführten Methode im Vorzustand σ_1 führt dazu, dass das Symbol n in den Definitionsbereich des neuen Zustands aufgenommen wird: Der Nachzustand dieser Deklaration ist

$$\sigma_2 = \sigma_1 \oplus \{n \mapsto \bot\}$$

Mit den elementaren Operationen können wir jetzt nur noch die Nullreferenz zuweisen: durch die Anweisung

```
n = null;
```

verändert sich der Zustand in

$$\sigma_3 = \sigma_1 \oplus \{n \mapsto null\}$$

Ein echtes Objekt vom Klassentyp `Node` muss in Java mit Hilfe des new-Operators erzeugt werden, etwa durch die Anweisung

```
n = new Node();
```

Die damit verbundene Allokation eines neuen Objektes vom Klassentyp `Node` im Speicher hat zur Folge, dass damit gleich mehrere neue Symbole in den Definitionsbereichs des Nachzustandes aufgenommen werden, denn man kann jetzt auf die Symbole `n.data` und `n.next` zugreifen. Weiterhin wird eine neue *Objektreferenz*, also eine neue Adresse im virtuellen Java-Adressraum erzeugt, die auf das neu erzeugt Objekt zeigt und der Referenzvariablen n zugewiesen wird. Die Menge aller definierten Objektreferenzen bezeichnen wir mit dem Hilfssymbol Ref, welches wir als Element der Symbolmenge V ansehen. Da der new-Operator alle Feldelemente mit Defaultwerten vorbelegt, führt diese Anweisung auf einen Nachzustand σ_4, der

$$\begin{aligned} &\sigma_3 \oplus \{n.data \mapsto null, n.next \mapsto null\} \subseteq \sigma_4 \wedge \\ &(\exists r \in \mathbb{N} : \sigma_4(n) = r \wedge r \notin \sigma_3(Ref) \wedge \sigma_4(Ref) = \sigma_3(Ref) \cup \{r\}) \end{aligned} \tag{1}$$

erfüllt. Da der Wert der Objektreferenz nicht vorhergesagt werden kann, können wir nur die oben gemachte Existenzaussage treffen:

- Die neue Referenz ist eine natürliche Zahl.
- Die neue Referenz wurde vorher – das heisst im Zustand σ_3 – noch nicht benutzt.
- Die new-Operation erzeugt genau eine neue Referenz[1].

[1] Die Situation wird komplexer, wenn nicht der *Defaultkonstruktor*, sondern spezielle Konstruktoren verwendet werden, die ihrerseits wieder den new-Operator verwenden.

Die neue semantische Komplexitätsstufe, welche durch rekursive Datentypen eingeführt wird, zeigt sich, wenn Objektinstanzen solcher Typen auf sich gegenseitig verweisen: Es werden durch bestimmte Zuweisungen plötzlich konzeptuell *unendliche viele neue Symbole* erzeugt! Betrachte hierzu die Anweisungsfolge

```
1           l.first = new Node();
2           l.last = new Node();
3           l.first.next = l.last;
4           l.last.next = l.first;
```

aus der Methode `public static List create()`: Nach der oben eingeführten semantischen Regel für den `new`-Operator sind nach den beiden `new`-Anweisungen die Symbole `l.first.next` und `l.last.next` definiert. Mit den Zuweisungen in Zeilen 3 und 4 folgt aus der semantischen Regel für die Zuweisung, dass im Nachzustand σ von Zeile 4

$$\sigma(\mathrm{l.first}) = \sigma(\mathrm{l.last.next})$$
$$\sigma(\mathrm{l.last}) = \sigma(\mathrm{l.first.next})$$

gilt. Da nach obiger Regel (1) das Symbol `l.first.next` existiert, muss also auch `l.last.next.next` existieren (und natürlich den selben Wert wie `l.first.next` tragen). Da analog `l.last.next` existiert, muss ebenso `l.first.next.next` existieren, also auch `l.last.next.next.next` usw. Diese Konstruktion führt auf die unbeschränkte Folge neuer Symbole

```
l.first.next, l.first.next.next, l.first.next.next.next, ...
l.last.next,  l.last.next.next,  l.last.next.next.next ...
```

Die präzise Definition, welche Symbolle jetzt tatsächlich im Definitionsbereich von σ liegen, erfolgt über den Begriff der *Abgeschlossenheit von Mengen.*

Abgeschlossenheit von Mengen: Gegeben sei eine Menge D, sowie $(n+1)$-stellige Relationen (n hängt von R ab)

$$R \subseteq \underbrace{D \times \ldots \times D}_{n+1}$$

Eine Teilmenge $B \subseteq D$ heisst *abgeschlossen unter den Relationen* R, wenn für alle R gilt:

$$\forall b_1, \ldots, b_n \in B, b_{n+1} \in D : (b_1, \ldots, b_n, b_{n+1}) \in R \implies b_{n+1} \in B$$

Abschlussoperator: Wir können umgekehrt zu einer bliebigen Teilmenge $B \subseteq D$ und gegebenen $(n+1)$-stelligen Relation $R \in K$ den *Abschluss* $\mathrm{Closure}_K(B)$ *von* B *bzgl.* K definieren: Betrachte hierzu die Kollektion aller Mengen $S \subseteq D$, welche folgenden beiden Bedingungen erfüllen:

1. $B \subseteq S$
2. Für alle $R \in K$ gilt $b_1, \ldots, b_n \in S \wedge b_{n+1} \in D \wedge (b_1, \ldots, b_n, b_{n+1}) \in R \implies b_{n+1} \in S$

Dann definieren wir

$$\mathrm{Closure}_K(B) = \bigcap S$$

Die die Relationen R enthaltende Menge K heisst die *erzeugende Menge* von $\mathrm{Closure}_K(B)$.

Abschlussoperator für einfach verkettete Listen: Die oben allgemein eingeführte Abschlussoperation wenden wir jetzt konkret auf einfach verkettete Listen an. Unsere R sind jetzt zweistellige Relationen auf Variablensymbolen. Natürlich hängen diese Relationen vom Programmzustand ab, denn der definiert ja, welche Objekte aktuell im Zugriff liegen; daher schreiben wir $R(\sigma) \subset V \times V$ und definieren

$$R_1(\sigma) = \{(l, l.\mathit{next}) \mid l \in \mathrm{dom}(\sigma) \wedge \sigma(l) \in \mathit{Ref} \wedge D(l) = \mathit{Node}\} \quad (2)$$

$$R_2(\sigma) = \{(l, l.\mathit{data}) \mid l \in \mathrm{dom}(\sigma) \wedge \sigma(l) \in \mathit{Ref} \wedge D(l) = \mathit{Node}\} \quad (3)$$

Die Relation $R_1(\sigma)$ assoziiert also zu jedem definierten Symbol, welches vom Listenknotentyp *Node* ist, und dem eine gültige Objektreferenz l mit $\sigma(l) \in \mathit{Ref}$ (also $\sigma(l) \notin \{\bot, \mathit{null}\}$) ein neues Symbol, welches durch Anhängen des Suffixes `.next` entsteht. Analog assoziiert R_2 zu jeder gültige Objektreferenz l das neue Symbol $l.\mathit{data}$.

Nun *fordern* wir für jeden gültigen Programmzustand σ, dass dieser immer bzgl. $K(\sigma) = \{R_1(\sigma), R_2(\sigma)\}$ abgeschlossen sein muss:

$$\mathrm{dom}(\sigma) = \mathrm{Closure}_{K(\sigma)}(\mathrm{dom}(\sigma)) \quad (4)$$

Zu beachten ist dabei, dass wir zwar in einem Programmzustand unendlich viele Symbole, aber nur endlich viele Objekte haben können, auf welche diese Symbole zeigen. Wenn zwei Symbole auf das selbe Listenelement zeigen, müssen ihre Referenzen auf Nutzdaten und Nachfolgerelemente selbstverständlich wieder auf das selbe Listenelement oder beide auf `null` zeigen. Dies führt zur letzten Wohldefiniertheitsbedingung für einen gültigen Programmzustand σ:

$$\begin{aligned} &\forall l_1, l_2 \in \mathrm{dom}(\sigma) : D(l_1) = D(l_2) = \mathit{Node} \wedge \sigma(l_1) = \sigma(l_2) \neq \mathit{null} \\ &\quad \implies \sigma(l_1.\mathit{data}) = \sigma(l_2.\mathit{data}) \wedge \sigma(l_1.\mathit{next}) = \sigma(l_2.\mathit{next}) \end{aligned} \quad (5)$$

Auch diese Konsistenzbedingunge lässt sich wieder durch einen Abschlussoperator definieren: Wir wählen hierzu zwei 3-stellige Relationen

$$R_3(\sigma) \subseteq (V \times Ref) \times (V \times Ref) \times (V \times (Node \cup \{null\})) \quad (6)$$
$$R_4(\sigma) \subseteq (V \times Ref) \times (V \times Ref) \times (V \times (Object \cup \{null\})) \quad (7)$$

die folgendermaßen definiert sind:

$$R_3(\sigma) = \{(l_1 \mapsto r_1, l_2 \mapsto r_2, l_2.next \mapsto m) \mid l_i \mapsto r_i \in \sigma \wedge \quad (8)$$
$$r_i \in Ref, i = 1,2 \wedge r_1 = r_2 \wedge m = \sigma(l_1.next)\} \quad (9)$$
$$R_4(\sigma) = \{(l_1 \mapsto r_1, l_2 \mapsto r_2, l_2.data \mapsto o) \mid l_i \mapsto r_i \in \sigma \wedge \quad (10)$$
$$r_i \in Ref, i = 1,2 \wedge r_1 = r_2 \wedge o = \sigma(l_1.data)\} \quad (11)$$

Mit $L(\sigma) = \{R_3(\sigma), R_4(\sigma)\}$ lässt sich (5) damit äquivalent durch die Forderung

$$\sigma = Closure_{L(\sigma)}(\sigma) \quad (12)$$

ausdrücken.

Werden in einem Java-Programm andere rekursive Datentypen verwendet, müssen die beiden Forderungen aus (4) und (12) um entsprechende Relationen $R(\sigma)$ und Konsistenzbedingungen erweitert werden. In unserem Beispielprogramm `MyList.java` wird beispielsweise noch die Klasse

```
class List {

    /** Ankerelement für den Listenanfang */
    Node first;
    /** Ankerelement für das Listenende */
    Node last;

    /** Referenz auf den Vorgänger des aktuellen Elemets */
    Node predecessorActualPtr;

    /**
     *  Die aktuelle Länge der Liste wird bei allen ändernden
     *  Funktionen mit gerechnet, damit die Längenabfrage
     *  immer in konstanter Zeit beantwortet werden kann.
     */
    int len;

}
```

verwendet. Eine Operationsfolge

```
1       List l = new List();
2       l.first = new Node();
3       l.last = new Node();
4       l.first.next = l.last;
5       l.last.next = l.first;
6       l.predecessorActualPtr = l.first;
7       l.len = 0;
```

wie sie in Methode `public static List create()` verwendet wird, hat also bei sinngemäßer Anwendung der Regeln (4) und (12) die folgende Wirkung:

- Nach Ausführung von Zeile 1 sind die Symbole `l`, `l.first`, `l.last`, `l.predecessorActualPtr`, `l.len` bekannt.
- Nach Ausführung von Zeile 6 ist die unbeschränkte Symbolkollektion

  ```
  l.first.next, l.first.next.next, l.first.next.next.next, ...
  l.first.data, l.first.next.data, l.first.next.next.data, ...
  l.last.next,  l.last.next.next,  l.last.next.next.next ...
  l.last.data,  l.last.next.data,  l.last.next.next.data, ...
  l.predecessorActualPtr.next, l.predecessorActualPtr.next.next, ...
  l.predecessorActualPtr.data, l.predecessorActualPtr.next.data, ...
  ```

  bekannt, und der Nachzustand σ erfüllt

$$\begin{aligned}
&\sigma(\mathrm{l.last}) = \sigma(\mathrm{l.first.next})\\
&\sigma(\mathrm{l.last.data}) = \sigma(\mathrm{l.first.next.data})\\
&\sigma(\mathrm{l.last.next}) = \sigma(\mathrm{l.first.next.next})\\
&\sigma(\mathrm{l.last.next.data}) = \sigma(\mathrm{l.first.next.next.data})\\
&\ldots
\end{aligned}$$

Wir führen zu gegebener Funktion $\sigma : V \not\longrightarrow D$ daher den *Abschlussoperator* $\mathcal{C}$ ein: $\mathcal{C}(\sigma)$ ist die kleinste Funktion, welche folgende Eigenschaften erfüllt:

$$\mathrm{dom}(\mathcal{C}(\sigma)) = \mathrm{Closure}_K(\mathrm{dom}(\mathcal{C}(\sigma))) \quad (13)$$

$$\mathcal{C}(\sigma) = \mathrm{Closure}_L(\mathcal{C}(\sigma)) \quad (14)$$

Mit diesem Abschlussoperator kann die semantische Regel für Variablenzuweisung, welche bei atomaren Datentypen bekanntlich

$$[\![x = e]\!](\sigma) = \sigma \oplus \{x \mapsto \sigma(e)\}$$

lautet, für Variablen l von rekursivem Datentyp zu

$$[\![x = e]\!](\sigma) = \mathcal{C}(\sigma \oplus \{x \mapsto \sigma(e)\}) \tag{15}$$

verallgemeinert werden.

Die oben beschriebenen theoretischen Sachverhalte lassen sich informell folgendermaßen zusammenfassen. Gegeben sei dazu eine beliebige Klasse, die nur Feldelemente enthält.

```
class C {  T1 x1; ...; Tn xn; }
```

1. Operation `C c = new C();`

   - erweitert den Definitionsbereich des Vorzustands σ um die Symbole $c, c.x_1, \ldots, c.x_n$,
   - Erweitert σ um ein *Argument* $\mapsto$ *Bildwert* Paar $c \mapsto r$, wobei $r \in \mathbb{N}$ eine bisher noch nicht verwendete Referenz ist,
   - Erweitert σ um die *Argument* $\mapsto$ *Bildwert* Paare $c.x_1 \mapsto init_1, \ldots, c.x_n \mapsto init_n$, welche allen Feldelementen typgemäße Initialwerte zuordnen: Für Klassentypen $null$, für Zahlen 0 bzw. 0.0, für Boolsche Werte `false`.

2. **Induktive Regel 1:** Wenn Symbol z im Definitionsbereich von σ enthalten und vom Klassentyp C ist und $\sigma(z) \neq null$ gilt, sind immer auch $z.x_1, \ldots, z.x_n$ in $dom(\sigma)$ enthalten.

3. **Induktive Regel 2:** Wenn Symbole z, w im Definitionsbereich von σ enthalten und vom Klassentyp C sind und $\sigma(z) = \sigma(w) \neq null$ gilt, haben alle Feldwerte dieselbe Valuation:

$$\forall i \in \{1, \ldots, n\} : \sigma(z.x_i) = \sigma(w.x_i)$$

Notation für n-fache Anwendung der Nachfolgeroperation: Da im folgenden häufig eine mehrfache Anwendung der `.next`-Referenzierung erforderlich ist, definieren wir

$$\begin{aligned} &\eta : V \times \mathbb{N}_0 \longrightarrow V \\ &\eta(x, 0) = x \\ &\eta(x, k) = \eta(x, k-1).next \text{ für } k > 0 \end{aligned}$$

Für $k > 0$ ist also

$$\eta(x, k) = x.\underbrace{next.\ldots.next}_{k}$$

4 Abstraktionsfunktion für einfach verkettete Listen

Nach den Ausführungen des vorigen Abschnitts können wir jetzt für eine gegebene einfach verkettete Java-Liste aus Programm `MyList.java` eine *Abstraktionsfunktion* α in das zugehörige mathematische Listenmodell geben. Diese Abstraktionsfunktion hängt natürlich vom aktuellen Programmzustand σ ab. Da unsere Java-Listenklassen Nutzdaten aus `Object` referenzieren, ist das mathematische Modell über dem selben Typ definiert:

$$\alpha_\sigma : V \not\longrightarrow \mathrm{List}(\mathrm{Object})$$

Die Abstraktionsfunktion kann nur diejenigen Symbole auf eine mathematische Liste abbilden, die vom Typ `class List` sind und eine gültige Objektreferenz sowie weitere Wohlgeformtheitsbedingungen in Bezug auf die Verkettung darstellen. Im Detail lauten diese Bedingungen folgendermaßen:

$$\begin{array}{l}
\forall \sigma \in V \not\longrightarrow D, l \in V : \\
\quad l \in \mathrm{dom}(\alpha_\sigma) \iff \\
\qquad D(l) = \mathrm{List} \wedge \\
\qquad \mathrm{null} \notin \{\sigma(l), \sigma(l.\mathrm{first}), \sigma(l.\mathrm{last}), \sigma(l.\mathrm{first.next}), \sigma(l.\mathrm{last.next})\} \wedge \\
\qquad ((\sigma(l.\mathrm{first.next}) = \sigma(l.\mathrm{last}) \wedge \sigma(l.\mathrm{last.next}) = \sigma(l.\mathrm{first}) \wedge \\
\qquad\qquad \sigma(l.\mathrm{predecessorActualPtr}) = \sigma(l.\mathrm{first}) \wedge \sigma(l.\mathrm{len}) = 0) \vee \\
\qquad (\exists n \in \mathbb{N} : \forall k \in \{1, \ldots, n\} : \\
\qquad\quad \sigma(\eta(l.\mathrm{first}, k)) \neq \mathrm{null} \wedge \\
\qquad\quad \sigma(\eta(l.\mathrm{first}, k).\mathrm{data}) \neq \mathrm{null} \wedge \\
\qquad\quad \sigma(\eta(l.\mathrm{first}, n).\mathrm{next}) = \sigma(l.\mathrm{last}) \wedge \\
\qquad\quad \sigma(l.\mathrm{last.next}) = \sigma(\eta(l.\mathrm{first}, n)) \wedge \\
\qquad\quad \sigma(l.\mathrm{predecessorActualPtr}) \in \\
\qquad\qquad \{\sigma(l.\mathrm{first}), \sigma(l.\mathrm{last})\} \cup \{\sigma(\eta(l.\mathrm{first}, k)) \mid k \in \{1, \ldots, n\}\} \wedge \\
\qquad\quad \sigma(l.\mathrm{len}) = n))
\end{array}$$

Mit dieser Festlegung des Definitionsbereichs können wir jetzt für $l \in dom(\alpha_\sigma)$ den Funktionswert explizit angeben:

$$\alpha_\sigma(l) = \begin{cases} (\langle\,\rangle, \langle\,\rangle) \\ \quad \text{falls } \sigma(l.len) = 0 \\ (\langle\,\rangle, \langle \sigma(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma(l.len)\}\rangle) \\ \quad \text{falls } \sigma(l.len) > 0 \wedge \sigma(l.predecessorActualPtr) = \sigma(l.first) \\ (\langle \sigma(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma(l.len)\}\rangle, \langle\,\rangle) \\ \quad \text{falls } \sigma(l.len) > 0 \wedge \sigma(l.predecessorActualPtr.next) = \sigma(l.last) \\ (\langle \sigma(\eta(l.first, k).data) \mid k \in \{1, \ldots, m\}\rangle, \\ \langle \sigma(\eta(l.first, k).data) \mid k \in \{m+1, \ldots, \sigma(l.len)\}\rangle) \\ \quad \text{falls } \sigma(l.len) > 0 \wedge \sigma(l.predecessorActualPtr) = \sigma(\eta(l.first, m)) \end{cases} \tag{16}$$

5 Verfeinerungsbeweise für Listenoperationen

In diesem Abschnitt soll die Konstruktion von Verfeinerungsbeweisen erläutert werden, welche zeigen, dass eine konkrete Listenoperation – beispielsweise die aus dem o. g. Java-Beispielprogramm – korrekte Implementierungen der zugehörigen abstrakten mathematisch definierten Listenoperationen sind.

Wir illustrieren den Verfeinerungsbeweis am Beispiel der abstrakten und konkreten *delete*-Funktionen:

Korrektheit von delete(): Für alle Programmzustände σ_1 gilt: Wenn $l \in dom(\alpha_{\sigma_1})$, dann gilt auch für den Nachzustand $\sigma_2 = [\![delete()]\!](\sigma_1)$ der Operationsausführung

```
boolean b = delete(l);
```

$l \in dom(\alpha_{\sigma_2})$; und es gilt weiterhin

$$\alpha_{\sigma_2}(l) = delete_A(\alpha_{\sigma_1}(l))$$

Beweis: Betrachte im gegebenen Vorzustand σ_1 den Programmcode von `public static boolean delete(List l)`:

```
1       if ( l.predecessorActualPtr.next == l.last )
2              return false;
3       l.predecessorActualPtr.next
4                = l.predecessorActualPtr.next.next;
5       if ( l.predecessorActualPtr.next == l.last ) {
6                l.last.next = l.predecessorActualPtr;
7       }
8       l.len = l.len - 1;
9       return true;
```

Für die Anwendung der bekannten semantischen Regeln $[\![P]\!](\sigma) = \sigma'$ wählen wir die Bezeichnung $P(n, m)$ für die Zeilenabschnitte n bis m im obigen Code, sowie $P(n)$ für die Bezeichnung einer einzelnen Zeile: Der gesamte Code der `delete()`-Funktion wird also mit $P(1, 9)$ bezeichnet, und $P(8)$ steht für

```
l.len = l.len - 1;
```

Die Spezifikation von in α_{σ_1} in (16) legt nahe, beim Beweis für die mittels `delete()` zu bearbeitende Liste `l` 4 Fälle zu unterscheiden:

1. Die Liste ist leer: $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle\,\rangle)$

2. Der bearbeitete Teil der Liste ist leer, der noch nicht bearbeitete ist aber *nicht* leer:
   $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma_1(l.len)\}\rangle)$

3. Der bearbeitete Teil der Liste ist *nicht* leer, der noch nicht bearbeitete ist leer:
   $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma_1(l.len)\}\rangle, \langle\,\rangle)$

4. Der unbearbeitete Teil der Liste sowie der bearbeitete Teil sind beide nicht leer:
   $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, m\}\rangle,$
   $\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{m+1, \ldots, \sigma_1(l.len)\}\rangle)$ mit geeignetem $m > 0$.

In jedem Fall können wir fordern, dass l im Vorzustand σ_1 wohlgeformt ist, d. h. alle logischen Bedingungen für die Elemente aus $dom(\alpha_{\sigma_1})$ erfüllt.

Fall 1.: Aus $l \in dom(\alpha_{\sigma_1}) \wedge l.len = 0$ leiten wir ab:

$$\begin{aligned} &\sigma_1(l.first.next) = \sigma_1(l.last) \wedge \sigma_1(l.last.next) = \sigma_1(l.first) \wedge \\ &\quad \sigma_1(l.len) = 0 \wedge \sigma_1(l.predecessorActualPtr) = \sigma_1(l.first) \end{aligned} \tag{17}$$

Wir berechnen zuerst unter Anwendung der semantischen Regeln für `if` und ";":

$$\begin{aligned} [\![P(1,9)]\!](\sigma_1) &= [\![P(1,2); P(3,9)]\!](\sigma_1) \\ &\quad \text{[Semantik von ";"]} \\ &= [\![P(3,9)]\!]([\![P(1,2)]\!](\sigma_1)) \\ &\quad \text{[if-Regel, siehe Erläuterung unten]} \\ &= [\![P(2)]\!](\sigma_1) \\ &= \sigma_1 \\ &\quad \text{return statement ändert } \sigma_1 \text{ nicht} \end{aligned}$$

Bei der Anwendung der if-Regel haben wir (17) ausgenutzt: Wegen

$$\sigma_1(l.first.next) = \sigma_1(l.last) \wedge \sigma_1(l.predecessorActualPtr) = \sigma_1(l.first)$$

folgt

$$\begin{aligned} &\sigma_1(l.predecessorActualPtr.next == l.last) = \\ &\qquad \text{[Wirkung von } \sigma_1 \text{ auf Terme]} \\ &\qquad (\sigma_1(l.predecessorActualPtr.next) == \sigma_1(l.last)) = \\ &\qquad (\sigma_1(l.first.next) == \sigma_1(l.last)) = \\ &\qquad \texttt{true} \end{aligned}$$

Also wird der if-Zweig ausgeführt, und die Funktion kehrt mit Rückgabewert `false` zurück. Damit bleibt `delete()` also ohne Wirkung, wie es für das mathematische Pendent $delete_A$ gefordert wurde, da die leere liste kein Element des Definitionsbereichs von $delete_A$ ist.

Fall 2.: Wir unterscheiden 2 Unterfälle, je nachdem, ob im noch zu bearbeitenden Teil der Liste nur ein Element enthalten ist (die Liste ist nach der Löschoperation also leer), oder ob noch ein zweites Element existiert.

Fall 2.1: $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma_1(l.len)\}\rangle) \wedge \sigma_1(l.len) = 1$: Dieser Ausdruck lässt sich jetzt vereinfachen zu

$$\alpha_{\sigma_1}(l) \quad = \quad (\langle\,\rangle, \langle \sigma_1(l.first.next.data)\rangle) \tag{18}$$

und damit gilt für die Anwendung der abstrakten delete-Funktion

$$delete_A(\alpha_{\sigma_1}(l)) = delete_A(\langle\,\rangle, \langle \sigma_1(l.first.next.data)\rangle) = \langle\,\rangle \tag{19}$$

Aus (16), Fall 2, folgt weiterhin

$$\sigma_1(l.len) \quad = \quad 1 \tag{20}$$

$$\sigma_1(l.predecessorActualPtr) \quad = \quad \sigma_1(l.first) \tag{21}$$

Hieraus folgt

$$\sigma_1(l.predecessorActualPtr.next) \quad = \quad \sigma_1(l.first.next) \tag{22}$$

und da $\sigma_1(l.first.next.data)$ definiert ist, muss $\sigma_1(l.first.next) \neq \sigma_1(l.last)$ gelten. Das bedeutet also

$$\sigma_1(l.predecessorActualPtr.next == l.last) \quad = \quad \texttt{false} \tag{23}$$

Wir berechnen nach dem selben Prinzip wie in Fall 1:

$$
\begin{aligned}
[\![P(1,9)]\!](\sigma_1) &= [\![P(1,2);P(3,9)]\!](\sigma_1) \\
&\quad [\text{Semantik von ";"}] \\
&= [\![P(2,9)]\!]([\![P(1,2)]\!](\sigma_1)) \\
&\quad [\text{if-Regel und (23)}] \\
&= [\![P(3,9)]\!](\sigma_1) \\
&= [\![P(3,4);P(5,9)]\!](\sigma_1) \\
&\quad [\text{Semantik von ";"}] \\
&= [\![P(5,9)]\!]([\![P(3,4)]\!](\sigma_1)) \\
&\quad [\text{Zuweisungsregel}] \\
&= [\![P(5,9)]\!](\mathcal{C}(\sigma_1 \oplus \\
&\qquad \{l.predecessorActualPtr.next \mapsto \\
&\qquad\quad \sigma_1(l.predecessorActualPtr.next.next)\})) \\
&\quad [\text{Gleichung (22)}] \\
&= [\![P(5,9)]\!](\mathcal{C}(\sigma_1 \oplus \\
&\qquad \{l.predecessorActualPtr.next \mapsto \\
&\qquad\quad \sigma_1(l.first.next.next)\})) \\
&\quad [\text{Wohlgeformtheitsbedingung, } n = m = 1] \\
&= [\![P(5,9)]\!](\mathcal{C}(\sigma_1 \oplus \\
&\qquad \{l.predecessorActualPtr.next \mapsto \sigma_1(l.last)\}))
\end{aligned}
$$

Mit der Abkürzung σ_5 für den Vorzustand von Zeile 5 gilt also

$$\sigma_5 = \mathcal{C}(\sigma_1 \oplus \{l.predecessorActualPtr.next \mapsto \sigma_1(l.last)\}) \qquad (24)$$

Da sich σ_5 und σ_1 nur im Funktionswert für Symbol $l.predecessorActualPtr.next$ unterscheiden, folgt

$$\sigma_5(l.last) = \sigma_1(l.last) \qquad (25)$$

und infolgedessen

$$\sigma_5(l.predecessorActualPtr.next == l.last) = \texttt{true} \qquad (26)$$

Wir berechnen damit weiter ab Programmzeile 5:

$$
\begin{aligned}
[\![P(5,9)]\!](\sigma_5) &= [\![P(5,6);P(8,9)]\!](\sigma_5) \\
&\quad [\text{Semantik von ";"}] \\
&= [\![P(8,9)]\!]([\![P(5,6)]\!](\sigma_5)) \\
&\quad [\text{If-Regel und (26)}] \\
&= [\![P(8,9)]\!]([\![P(6)]\!](\sigma_5)) \\
&\quad [\text{Zuweisungsregel}] \\
&= [\![P(8,9)]\!](\mathcal{C}(\sigma_5 \oplus \\
&\qquad \{l.last.next \mapsto \sigma_5(l.predecessorActualPtr)\})) \\
&\quad [\text{Zuweisungsregel}] \\
&= \mathcal{C}(\sigma_5 \oplus \{l.last.next \mapsto \sigma_5(l.predecessorActualPtr), \\
&\qquad l.len \mapsto \sigma_5(l.len) - 1\})) \\
&\quad [\text{Gleichungen (24,25)}] \\
&= \mathcal{C}(\sigma_1 \oplus \{l.predecessorActualPtr.next \mapsto \sigma_1(l.last), \\
&\qquad l.last.next \mapsto \sigma_1(l.predecessorActualPtr), \\
&\qquad l.len \mapsto \sigma_1(l.len) - 1\}) \\
&\quad [\text{Formeln (521,20)}] \\
&= \mathcal{C}(\sigma_1 \oplus \{l.first.next \mapsto \sigma_1(l.last), \\
&\qquad l.last.next \mapsto \sigma_1(l.first), \\
&\qquad l.len \mapsto 0\})
\end{aligned}
$$

Die obigen Berechnungen zusammen mit Formel (21) ergeben den Nachzustand

$$
\begin{aligned}
[\![P(1,9)]\!](\sigma_1) &= [\![P(5,9)]\!](\sigma_5) \\
&= \mathcal{C}(\sigma_1 \oplus \{l.first.next \mapsto \sigma_1(l.last), \\
&\qquad l.last.next \mapsto \sigma_1(l.first), \\
&\qquad l.predecessorActualPtr \mapsto \sigma_1(l.first), \\
&\qquad l.len \mapsto 0\})
\end{aligned}
$$

Gemäß der Wohlgeformtheitsbedingungen sind das gerade die Eigenschaften der leeren Liste; es gilt also mit (19)

$$\alpha_{[\![P(1,9)]\!](\sigma_1)}(l) = \langle\,\rangle = delete_A(\alpha_{\sigma_1}(l))$$

was für Fall 2.1 zu zeigen war.

Fall 2.2: $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma_1(l.len)\}\rangle) \wedge$ $\sigma_1(l.len) > 1$: Der Ausdruck lässt sich wegen $\sigma_1(l.len) > 1$ umschreiben zu

$$\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle \sigma_1(l.first.next.data)\rangle \frown \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{2, \ldots, \sigma_1(l.len)\}\rangle) \tag{27}$$

und nach der Definition der abstrakten Funktion $delete_A$ folgt

$$delete_A(\alpha_{\sigma_1}(l)) = (\langle\,\rangle, \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{2, \ldots, \sigma_1(l.len)\}\rangle) \tag{28}$$

Genau wie in Fall 2.1 gelten auch hier die Formeln (21,22,23). Wir berechnen wieder nach dem selben Prinzip wie in Fall 2.1:

$$\begin{aligned}
[\![P(1,9)]\!](\sigma_1) &= [\![P(1,2); P(3,9)]\!](\sigma_1) \\
&\quad [\text{Semantik von ";"}] \\
&= [\![P(2,9)]\!]([\![P(1,2)]\!](\sigma_1)) \\
&\quad [\text{if-Regel und (23)}] \\
&= [\![P(3,9)]\!](\sigma_1) \\
&= [\![P(3,4); P(5,9)]\!](\sigma_1) \\
&\quad [\text{Semantik von ";"}] \\
&= [\![P(5,9)]\!]([\![P(3,4)]\!](\sigma_1)) \\
&\quad [\text{Zuweisungsregel}] \\
&= [\![P(5,9)]\!](\mathcal{C}(\sigma_1 \oplus \\
&\qquad \{l.predecessorActualPtr.next \mapsto \\
&\qquad\quad \sigma_1(l.predecessorActualPtr.next.next)\})) \\
&\quad [\text{Gleichung (22)}] \\
&= [\![P(5,9)]\!](\mathcal{C}(\sigma_1 \oplus \\
&\qquad \{l.predecessorActualPtr.next \mapsto \\
&\qquad\quad \sigma_1(l.first.next.next)\}))
\end{aligned}$$

Wegen $\sigma_1(l.len) > 1$ ist $\sigma_1(l.first.next.next) \neq \sigma_1(l.last)$; daher gilt im Vorzustand

$$\sigma_5 = \mathcal{C}(\sigma_1 \oplus \{l.predecessorActualPtr.next \mapsto \sigma_1(l.first.next.next)\})) \tag{29}$$

von Zeile 5:

$$\sigma_5(l.predecessorActualPtr.next == l.last) = \texttt{false} \tag{30}$$

Die Berechnung des Nachzustands von `delete()` lässt sich also folgender-

maßen fortsetzen:

$$
\begin{aligned}
[\![P(5,9)]\!](\sigma_5) \quad & \text{[If-Regel und (30)]} \\
= \; & [\![P(8,9)]\!](\sigma_5) \\
& \text{[Zuweisungsregel]} \\
= \; & \mathcal{C}(\sigma_5 \oplus \{l.len \mapsto \sigma_5(l.len) - 1\}) \\
& \text{[l.len wurde in } P(1,7) \text{ nicht verändert, (29)]} \\
= \; & \mathcal{C}(\sigma_1 \oplus \{l.predecessorActualPtr.next \mapsto \sigma_1(l.first.next.next), \\
& \qquad\qquad l.len \mapsto \sigma_1(l.len) - 1\}) \\
& \text{[Gültigkeit von (21)]} \\
= \; & \mathcal{C}(\sigma_1 \oplus \{l.first.next \mapsto \sigma_1(\eta(l.first, 2)), \\
& \qquad\qquad l.len \mapsto \sigma_1(l.len) - 1\})
\end{aligned}
$$

Mit dieser Berechnung des Nachzustands und mit Formeln (27,28) folgt

$$
\begin{aligned}
\alpha_{[\![P(1,9)]\!](\sigma_1)}(l) &= (\langle\,\rangle, \langle \sigma_1(\eta(l.first, k).data) \mid k \in \{2, \ldots, \sigma_1(l.len)\}\rangle) \\
&= delete_A(\alpha_{\sigma_1}(l))
\end{aligned}
$$

was für Fall 2.2 zu zeigen war.

Fall 3. $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, \sigma_1(l.len)\}\rangle, \langle\,\rangle)$: Hier zeigt man genau wie in Fall 1, dass die `delete()`-Operation keine Wirkung hat, da

$$\sigma_1(l.predecessorActualPtr.next == l.last) = \texttt{true}$$

gilt.

Fall 4. $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{1, \ldots, m\}\rangle,$ $\langle \sigma_1(\eta(l.first, k).data) \mid k \in \{m+1, \ldots, \sigma_1(l.len)\}\rangle)$ mit geeignetem $m > 0$: Die Beweisführung verfläuft hier analog zu Fall 2, wobei man wieder unterscheiden muss, ob im noch zu bearbeitenden Teil der Liste nur ein Element ($\sigma_1(l.len) = m + 1$) oder mindestens 2 Elemente enthalten sind. Damit ist der Beweis abgeschlossen. □