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

19. Januar 2009

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 **2**

2 Ein mathematisches Listenmodell **4**

3 Verkettete Listen in Java und ihre operationelle Semantik **7**

4 Abstraktionsfunktion für doppelt verkettete Listen **13**

5 Verfeinerungsbeweise für Listenoperationen **15**

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 $\mathrm{dom}(f)$, also $\mathrm{dom}(f) = \{1, \ldots, n\}$ 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:

$$\#f = \mathrm{card}(\mathrm{dom}(f))$$

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 $f_i : \{1, \ldots n_i\} \longrightarrow T, i = 1, 2, n_i = \#f_i$ bezeichnet:

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

Bei einer nicht leeren Folge f bezeichnet

- $\mathrm{head}(f) = f(1)$ das erste Element,
- $\mathrm{last}(f) = f(\#f)$ das letzte Element,
- $\mathrm{tail}(f) = \langle f(2), \ldots, f(\#f) \rangle$ die Folge, welche aus f durch Entfernen des ersten Elementes entsteht.

Offenbar gilt für nicht leere Folgen f

$$f = \langle \mathrm{head}(f) \rangle \frown \mathrm{tail}(f)$$

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 *aktuelle Element* der Liste, welches eine Read-Operation ausliest oder eine Delete-Operation löscht, ist gerade das letzte Element von f_1, d. h. $last(f_1) = f_1(\#f_1)$. Das *als nächstes zu bearbeitende Element"* der Liste ist $f_2(1) = head(f_2)$. Nach der Bearbeitung dieses Elementes ist dann $tail(f_2)$ der weiterhin zu bearbeitende Rest. Dies führt uns dazu, die *Menge aller Listen über dem Datentyp* T als

$$List(T) = T^* \times T^*$$

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 $name_A()$, wobei das "A" für "abstrakte Funktion" steht.

Die *Append-Funktion* hängt ein neues Element *hinter* dem aktuellen Element ein, falls der bearbeitete Teil nicht leer ist. In jedem Fall wird das mit $append_A$ eingefügt Element das nächste zu bearbeitende:

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

Die *Insert-Funktion* fügt ein neues Element x *vor dem aktuellen Element* ein, falls dieses existiert. Wenn der bearbeitete Teil leer ist, wird das neue Element per Konvention als das *letzte* der gesamten Liste eingehängt:

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

Funktion *length* ermittelt die Länge einer Liste:

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

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

$$\begin{aligned} &isEmpty_A : List(T) \longrightarrow \mathbb{B} \\ &(f_1, f_2) \mapsto (length_A(f_1, f_2) = 0) \end{aligned}$$

Funktion *next* setzt das aktuelle Element auf das erste des bisher noch nicht bearbeiteten Listenteils um. Der noch zu bearbeitende Listenteil wird entsprechend um dieses Element verkürzt. Ist der noch zu bearbeitende Listenteil leer, wird per Konvention wieder das erste Listenelement als aktuelles Element verwendet:

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

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

$$\begin{aligned}&\mathrm{read}_A : \mathrm{List}(T) \not\longrightarrow T;\\&(f_1 \frown \langle x\rangle, f_2) \mapsto x\end{aligned}$$

Die *delete-Funktion* löscht das aktuelle Element. Existiert dieses nicht, weil der bearbeitete Teil der Liste leer ist, ist die Funktion wirkungslos:

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

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. Das eingefügte Element wird das neue aktuelle Element.

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

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:

$$\begin{aligned}&\mathrm{top}_A : \mathrm{List}(T) \not\longrightarrow T;\\&(\langle x\rangle \frown f_1, f_2) \mapsto x\end{aligned}$$

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

$$\begin{aligned}&\mathrm{pop}_A : \mathrm{List}(T) \not\longrightarrow \mathrm{List}(T) \times T;\\&(\langle x\rangle \frown f_1, f_2) \mapsto \begin{cases}((\langle \mathrm{head}(f_1 \frown f_2)\rangle, \mathrm{tail}(f_1 \frown f_2)), x) & \text{Falls } f_1 \frown f_2 \neq \langle\,\rangle\\((\langle\,\rangle, \langle\,\rangle), x) & \text{Falls } f_1 \frown f_2 = \langle\,\rangle\end{cases}\end{aligned}$$

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/ws0809/`
`./pi1/hintergrund/listen/MyList.java`

das doppelt ringverkettete 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 doppelt verkettete Listenelemente aus o. g. Beispielprogramm:

```
  class Rlist {
    // Referenz auf das Listenelement:
    String  data;
    // Rekursiver Verweis auf ein Element vom
    // Typ class Rlist, d.h. Verweis
    // auf das nächste Listenelement.
    Rlist n;
    // Verweis auf Vorgaenger
    Rlist p;
}
```

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

```
    Rlist nde;
```

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

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

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

```
nde = null;
```

verändert sich der Zustand in

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

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

```
nde = new Rlist();
```

Die damit verbundene Allokation eines neuen Objektes vom Klassentyp `Rlist` 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 `nde.data` und `nde.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 nde 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

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

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           Rlist l = new Rlist();
2           l.n = l;
3           l.p = l;
```

aus der Methode `public static List rlCreate()`: Nach der new-Anweisung in Zeile 1 können nach den oben gegebenen Regeln für den new-Operator Zuweisungen auf `l.n, l.p` erfolgen. Anwendung der Zuweisungsregel auf Zeilen 2 und 3 impliziert, dass $\sigma(l.n) = \sigma(l)$ und $\sigma(l.p) = \sigma(l)$ gelten, also sind auch `l.n.n, l.n.p, l.p.n, l.p.p` definiert und so weiter.

Die präzise Definition, welche Symbole 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 doppelt 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

$$R_0(\sigma) = \{(l, l.n) \mid l \in dom(\sigma) \wedge \sigma(l) \in Ref \wedge D(l) = Rlist\} \quad (2)$$

$$R_1(\sigma) = \{(l, l.p) \mid l \in dom(\sigma) \wedge \sigma(l) \in Ref \wedge D(l) = Rlist\} \quad (3)$$

$$R_2(\sigma) = \{(l, l.data) \mid l \in dom(\sigma) \wedge \sigma(l) \in Ref \wedge D(l) = Rlist\} \quad (4)$$

Die Relation $R_0(\sigma)$ assoziiert also zu jedem definierten Symbol, welches vom Listenknotentyp Rlist ist, und dem eine gültige Objektreferenz l mit $\sigma(l) \in Ref$ (also $\sigma(l) \notin \{\perp, null\}$) ein neues Symbol, welches durch Anhängen des Suffixes `.n` entsteht. Analog assoziiert R_1 zu jeder gültige Objektreferenz l das neue Symbol $l.p$ und R_2 das neue Symbol $l.data$.

Nun *fordern* wir für jeden gültigen Programmzustand σ, dass dieser immer bzgl. $K(\sigma) = \{R_0(\sigma), R_1(\sigma), R_2(\sigma)\}$ abgeschlossen sein muss:

$$dom(\sigma) = Closure_{K(\sigma)}(dom(\sigma)) \quad (5)$$

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 Vorgänger- bzw. Nachfolgerelemente selbstverständlich wieder auf das selbe Listenelement oder beide auf `null` zeigen. Dies führt zur letzten Wohldefiniertheitsbedingung für einen gültigen Programmzustand σ:

$$\forall l_1, l_2 \in dom(\sigma) : D(l_1) = D(l_2) = Rlist \wedge \sigma(l_1) = \sigma(l_2) \neq null \\ \Longrightarrow \sigma(l_1.data) = \sigma(l_2.data) \wedge \sigma(l_1.n) = \sigma(l_2.n) \wedge \sigma(l_1.p) = \sigma(l_2.p) \quad (6)$$

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

$$R_3(\sigma) \subseteq (V \times Ref) \times (V \times Ref) \times (V \times (Rlist \cup \{null\})) \quad (7)$$

$$R_4(\sigma) \subseteq (V \times Ref) \times (V \times Ref) \times (V \times (Rlist \cup \{null\})) \quad (8)$$

$$R_5(\sigma) \subseteq (V \times Ref) \times (V \times Ref) \times (V \times (Object \cup \{null\})) \quad (9)$$

die folgendermaßen definiert sind:

$$\begin{aligned} R_3(\sigma) &= \{(l_1 \mapsto r_1, l_2 \mapsto r_2, l_2.n \mapsto m) \mid l_i \mapsto r_i \in \sigma \wedge && (10)\\ &\quad r_i \in \mathrm{Ref}, i = 1,2 \wedge r_1 = r_2 \wedge m = \sigma(l_1.n)\} && (11)\\ R_4(\sigma) &= \{(l_1 \mapsto r_1, l_2 \mapsto r_2, l_2.p \mapsto m) \mid l_i \mapsto r_i \in \sigma \wedge && (12)\\ &\quad r_i \in \mathrm{Ref}, i = 1,2 \wedge r_1 = r_2 \wedge m = \sigma(l_1.p)\} && (13)\\ R_5(\sigma) &= \{(l_1 \mapsto r_1, l_2 \mapsto r_2, l_2.\mathrm{data} \mapsto o) \mid l_i \mapsto r_i \in \sigma \wedge && (14)\\ &\quad r_i \in \mathrm{Ref}, i = 1,2 \wedge r_1 = r_2 \wedge o = \sigma(l_1.\mathrm{data})\} && (15) \end{aligned}$$

Mit $L(\sigma) = \{R_3(\sigma), R_4(\sigma), R_5(\sigma)\}$ lässt sich (6) damit äquivalent durch die Forderung

$$\sigma = \mathrm{Closure}_{L(\sigma)}(\sigma) \tag{16}$$

ausdrücken.

Werden in einem Java-Programm andere rekursive Datentypen verwendet, müssen die beiden Forderungen aus (5) und (16) um entsprechende Relationen $R(\sigma)$ und Konsistenzbedingungen erweitert werden.

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:

$$\begin{aligned} \mathrm{dom}(\mathcal{C}(\sigma)) &= \mathrm{Closure}_K(\mathrm{dom}(\mathcal{C}(\sigma))) && (17)\\ \mathcal{C}(\sigma) &= \mathrm{Closure}_L(\mathcal{C}(\sigma)) && (18) \end{aligned}$$

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

$$[\![l = e]\!](\sigma) = \mathcal{C}(\sigma \oplus \{l \mapsto \sigma(e)\}) \tag{19}$$

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 `.n`-Referenzierung erforderlich ist, definieren wir

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

Für $k > 0$ ist also

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

Analog definieren wir die mehrfache Anwendung der `.p`-Referenzierung:

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

4 Abstraktionsfunktion für doppelt verkettete Listen

Nach den Ausführungen des vorigen Abschnitts können wir jetzt für eine gegebene doppelt 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 `String` referenzieren, ist das mathematische Modell über dem selben Typ definiert:

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

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

$$\begin{aligned}
&\forall \sigma \in V \not\longrightarrow D, l \in V : \\
&\quad l \in \mathrm{dom}(\alpha_\sigma) \iff \\
&\qquad D(l) = \mathrm{Rlist} \wedge \\
&\qquad \mathrm{null} \notin \{\sigma(l), \sigma(l.n), \sigma(l.p)\} \wedge \\
&\qquad (\sigma(l.n) = \sigma(l) \implies \sigma(l.p) = \sigma(l) \wedge \sigma(l.\mathrm{data}) = \mathrm{null}) \wedge \\
&\qquad (\sigma(l.p) = \sigma(l) \implies \sigma(l.n) = \sigma(l) \wedge \sigma(l.\mathrm{data}) = \mathrm{null}) \wedge \\
&\qquad (\exists m \in \mathbb{N}_0 : \sigma(\eta(l,m).\mathrm{data}) = \mathrm{null}) \wedge \\
&\qquad (\forall m \in \mathbb{N} : \sigma(l.\mathrm{data}) = \mathrm{null} \wedge \sigma(\eta(l,m).\mathrm{data}) = \mathrm{null} \implies \\
&\qquad\qquad \sigma(l) = \sigma(\eta(l,m))) \wedge \\
&\qquad (\forall m \in \mathbb{N} : \sigma(l.\mathrm{data}) = \mathrm{null} \wedge \sigma(\pi(l,m).\mathrm{data}) = \mathrm{null} \implies \\
&\qquad\qquad \sigma(l) = \sigma(\pi(l,m))) \wedge \\
&\qquad (\forall m \in \mathbb{N} : \sigma(\eta(l,m-1)) = \sigma(\eta(l,m).p)) \wedge \\
&\qquad (\forall m \in \mathbb{N} : \sigma(\pi(l,m-1)) = \sigma(\pi(l,m).n))
\end{aligned} \tag{20}$$

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

$$\alpha_\sigma(l) = \begin{cases} (\langle\,\rangle, \langle\,\rangle) \\ \quad \text{falls } \sigma(l.n) = \sigma(l) \\ (\langle\,\rangle, \langle \sigma(\eta(l,k).data) \mid k \in \{1,\ldots,max-1\}\rangle) \\ \quad \text{falls } \sigma(l.data) = null \wedge \sigma(\eta(l,max)) = \sigma(l) \wedge max \geq 2 \\ (\langle \sigma(\eta(l,k).data) \mid k \in \{2,\ldots,max\}\rangle, \langle\,\rangle) \\ \quad \text{falls } \sigma(l.n.data) = null \wedge \sigma(\eta(l,max)) = \sigma(l) \wedge max \geq 2 \\ (\langle \sigma(\eta(l,k).data) \mid k \in \{m_0,\ldots,max\}\rangle, \\ \langle \sigma(\eta(l,k).data) \mid k \in \{1,\ldots,m_0-2\}\rangle) \\ \quad \text{falls } \sigma(\eta(l,m_0-1).data) = null \wedge \sigma(\eta(l,max)) = \sigma(l) \wedge \\ \quad m_0 \geq 3 \wedge m_0 \leq max \end{cases} \tag{21}$$

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:

Korrektheitsbedingungen für rlDel(): Für alle Programmzustände σ_1 gilt: Wenn $l \in dom(\alpha_{\sigma_1})$, dann gilt auch für den Nachzustand $\sigma_2 = [\![rlDel(l)]\!](\sigma_1)$ der Operationsausführung `rlDel(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 rlDel(Rlist l)`[2]:

```
1           if ( l.data == null ) return;
2           l.n.p = l.p;
3           l.p.n = l.n;
```

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 `rlDel()`-Funktion wird also mit $P(1, 3)$ bezeichnet, und $P(3)$ steht für `l.p.n = l.n;`

Die Spezifikation von in α_{σ_1} in (21) legt nahe, beim Beweis für die mittels `rlDel()` 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:
   $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle \sigma_1(\eta(l, k).data) \mid k \in \{1, \ldots, max - 1\}\rangle)$
3. Der bearbeitete Teil der Liste ist *nicht* leer, der noch nicht bearbeitete ist leer:
   $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l, k).data) \mid k \in \{2, \ldots, max\}\rangle, \langle\,\rangle)$

[2] Der im Original-Quellcode enthaltene Sanity Check wurde hier zur Steigerung der Übersichtlichkeit weggelassen.

4. Der unbearbeitete Teil der Liste sowie der bearbeitete Teil sind beide nicht leer:
   $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l,k).data) \mid k \in \{n_0, \ldots, max\}\rangle,$
   $\langle \sigma_1(\eta(l,k).data) \mid k \in \{1, \ldots, m_0 - 2\}\rangle)$

mit geeignetem $max, m_0 > 0$ wie oben für α_σ spezifiziert. 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 der Definition von α_{σ_1} folgt für $\alpha_{\sigma_1}(l) = (\langle\,\rangle, \langle\,\rangle)$: $\sigma_1(l.n) = \sigma_1(l)$. Aus der Wohlgeformtheitsbedingung (20) folgt hieraus $\sigma_1(l.data) = null$. Wir können direkt aus Zeile 1 ablesen (Anwendung der if-Regel), dass damit die Methode sofort mittels `return` verlassen wird; die Liste bleibt also unverändert. Das entspricht genau dem Funktionswert der abstrakten Funktion $delete_A(\langle\,\rangle, \langle\,\rangle)$, wie aus obiger Definition von $delete_A$ abzulesen ist.

Fall 2.: In diesem Fall ist der bearbeitete Teil der Liste leer, und es gilt $\sigma_1(l.data) = null$. Genau wie im Fall 1 verlassen wir die Methode mittels `return`, ohne die Liste zu verändern. Dies entspricht wieder dem definierten oben Verhalten von $delete_A(\langle\,\rangle, f_2)$.

Fall 3. $\alpha_{\sigma_1}(l) = (\langle \sigma_1(\eta(l,k).data) \mid k \in \{1, \ldots, max - 1\}\rangle, \langle\,\rangle)$: Aus (21) folgt in diesem Fall $\sigma_1(l.n.data) = null \wedge \sigma_1(\eta(l, max)) = \sigma_1(l) \wedge max \geq 2$. Aus (20) folgt $\sigma_1(l.data) \neq null$, denn nur das Ankerelement darf eine Null-Referenz auf die `.data`-Komponente besitzen. Damit können wir berechnen:

$$\begin{aligned}
&[\![P(1,3)]\!](\sigma_1) = \\
&\qquad [\![P(2,3)]\!](\sigma_1) = \\
&\qquad [\![P(3)]\!](\mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l.p)\})) = \\
&\qquad \mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l.p), l.p.n \mapsto \sigma_1(l.n)\}) =_{def} \sigma_2
\end{aligned} \tag{22}$$

Wir müssen jetzt folgendes zeigen:

1. Die Liste ist nach der Ausführung von `rlDel(l)` noch wohldefiniert: $\sigma_2(l.p)$ genügt den Bedingungen von (21)
2. Das beim Aufruf verwendete aktuelle Element l wird durch den Aufruf von `rlDel(l)` gelöscht, und das neue aktuelle Element ist $l.p$: $\alpha_{\sigma_2}(l.p) = (\langle \sigma_1(\eta(l,k).data) \mid k \in \{1, \ldots, max - 2\}\rangle, \langle\,\rangle)$

Dazu unterscheiden wir 2 Unterfälle:

Fall 3.1 $\sigma_1(l.p) = \sigma_1(l.n)$. Dieser Fall kann nur für $max = 2$ auftreten, die Liste bestand also vor Aufruf von `rlDel(l)` nur aus einem Element. Aus der abstrakten $delete_{\mathcal{A}}$-Operation folgern wir, dass die Liste nach Operationsausführung leer sein müsste. Wir zeigen, dass tatsächlich $\sigma_2(l.p) = (\langle\,\rangle, \langle\,\rangle)$ gilt:

Im Fall 3 gilt $\sigma_1(l.n.data) = null$, für den Unterfall 3.1 gilt zusätzlich $\sigma_1(l.p) = \sigma_1(l.n)$. Damit ist $l_0 = \sigma_1(l.p) = \sigma_1(l.n)$ das Ankerelement der Liste. Unter Verwendung der Wohldefiniertheitsbedingung für gültige Programmzustände (6) können wir die Ableitung (22) jetzt in Hinblick auf l_0 erweitern: Aus der durch den Abschlussoperator garantierten Wohldefiniertheitsbedingung (e2) folgt aus $\sigma_1(l.p) = \sigma_1(l_0)$ und $\sigma_2(l.p.n) = \sigma_1(l_0)$ auch $\sigma_2(l_0.n) = \sigma_1(l_0)$, und aus $\sigma_1(l.n) = \sigma_1(l_0)$ und $\sigma_2(l.n.p) = \sigma_1(l_0)$ auch $\sigma_2(l_0.p) = \sigma_1(l_0)$:

$$
\begin{aligned}
&[\![P(1,3)]\!](\sigma_1) = \\
&\quad [\![P(2,3)]\!](\sigma_1) = \\
&\quad [\![P(3)]\!](\mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l_0)\})) = \\
&\quad \mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l_0), l.p.n \mapsto \sigma_1(l_0)\}) = \\
&\quad \mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l_0), l.p.n \mapsto \sigma_1(l_0), l_0.p \mapsto \sigma_1(l_0), l_0.n \mapsto \sigma_1(l_0)\}) \\
&\quad =_{def} \sigma_2
\end{aligned}
\tag{23}
$$

Das Ankerelement l_0 der Liste erfüllt folglich $\sigma_2(l_0) = \sigma_2(l_0.p) = \sigma_2(l_0.n)$. Damit repräsentiert das Ankerelement laut (21) eine leere Liste.

Fall 3.2 $\sigma_1(l.p) \neq \sigma_1(l.n)$. In diesem Fall ist $max > 0$. Weiterhin gilt jedoch $\sigma_1(l.n.data) = null$, $\sigma_1(l.n)$ zeigt also wieder auf das Ankerelement l_0. Wir wiederholen mit diesem Wissen analog zu 3.1 die Ableitung (22):

$$
\begin{aligned}
&[\![P(1,3)]\!](\sigma_1) = \\
&\quad [\![P(2,3)]\!](\sigma_1) = \\
&\quad [\![P(3)]\!](\mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l.p)\})) = \\
&\quad \mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l.p), l.p.n \mapsto \sigma_1(l.n)\}) = \\
&\quad \mathcal{C}(\sigma_1 \oplus \{l.n.p \mapsto \sigma_1(l.p), l.p.n \mapsto \sigma_1(l_0), l_0.p \mapsto \sigma_1(l.p)\}) = \\
&\quad =_{def} \sigma_2
\end{aligned}
\tag{24}
$$

Da die Liste den Wohldefiniertheitsbedingungen (20) im Zustand σ_1 genügte, gilt dies auch noch für $l.p$ in σ_2, denn laut (24), denn:

1. Die ersten 4 Bedingungen gelten trivialerweise weiter.
2. $(\exists m \in \mathbb{N}_0 : \sigma_2(\eta(l.p, m).data) = null)$ gilt mit $m = 1$, denn $\sigma_2(l.p.n) = \sigma_1(l_0)$, und l_0 ist das Ankerelement.

3. Da in σ_1 nur ein Ankerelement existierte und durch `rlDel()` kein neues Element eingekettet wurde und keine `.data`-Komponente verändert wurde, gilt die Eindeutigkeit des Ankerelementes l_0 auch weiterhin.

4. $(\forall m \in \mathbb{N} : \sigma_2(\eta(l.p, m-1)) = \sigma_2(\eta(l.p, m).p))$ gilt aus folgenden Gründen:

   $\sigma_2(\eta(l.p, 0)) = \sigma_1(\eta(l.p, 0)) = \sigma_2(l_0.p)\sigma_2(\eta(l.p, 1).p))$, denn $\sigma_2(l.p.n) = \sigma_1(l_0) = \sigma_2(l_0)$. Da sich nur $l.p.n$ ind $l_0.p$ und ihre syntaktischen Äquivalente beim Übergang von σ_1 nach σ_2 verändert haben, ist damit alles gezeigt.

5. Analog gilt auch weiter $(\forall m \in \mathbb{N} : \sigma_2(\pi(l, m-1)) = \sigma_2(\pi(l, m).n))$.

Da in $\sigma_1(\eta(l, max)) = \sigma_1(l)$ galt, l dann ausgekettet wurde und wie eben gezeigt die Verkettung immer noch korrekt ist, muss $\sigma_2(\eta(l.p, max-1)) = \sigma_2(l.p)$ gelten. Damit folgt

$$\alpha_{\sigma_2}(l.p) = (\langle \sigma(\eta(l.p, k).data) \mid k \in \{2, \ldots, max-1\} \rangle, \langle\ \rangle)$$

was zu zeigen war.

Fall 4. verläuft analog zu den Fällen 2 und 3. Damit ist der Beweis abgeschlossen. □