14 Generative Rekursion

Häufig richtet sich die Struktur von Funktionen nach der Struktur der Daten, auf denen die Funktionen arbeiten. Beispielsweise sieht unser Entwurfsrezept für algebraische Datentypen vor, für jede Alternative des Datentyps eine Hilfsfunktion zu definieren. Wenn wir einen rekursiven Datentyp haben, so sieht unser Entwurfsrezept den Einsatz struktureller Rekursion vor.

In manchen Fällen jedoch muss man von dieser Parallelität von Daten und Funktionen abweichen: Die Struktur der Daten passt nicht zu der Art und Weise, wie das Problem in Teilprobleme aufgeteilt werden soll.

14.1 Wo kollidiert der Ball?

Betrachten Sie als Beispiel die Fallstudie zum Ball in Bewegung in §6.5 “Fallstudie: Ein Ball in Bewegung”. Nehmen wir an, wir möchten eine Funktion, die zu einem Ball berechnet, an welcher Position der Ball das erste Mal eine Kollision mit der Wand hat. Wenn wir einen Ball `ball` haben, so können wir durch Aufruf von `(move-ball ball)`, `(move-ball (move-ball ball))` usw. die Bewegung des Balls simulieren. Wie lange wollen wir diese Simulation durchführen? So lange, bis es eine Kolllision gibt, also bis `(collision current-ball)` nicht `"none"` ist.

Dies rechtfertigt die folgende Definition:

```
; Ball -> Posn
; computes the position where the first collision of the ball oc-
curs
(define (first-collision ball)
  (cond [(string=? (collision (ball-loc ball)) "none")
         (first-collision (move-ball ball))]
        [else (ball-loc ball)]))
```

Wenn wir uns diese Definition anschauen, so stellen wir zwei Besonderheiten fest: 1) Die Fallunterscheidung im Body der Funktion hat nichts mit der Struktur der Eingabe zu tun. 2) Das Argument, welches wir im rekursiven Funktionsaufruf übergeben, ist kein Teil der ursprünglichen Eingabe. Stattdessen generiert `(move-ball ball)` einen völlig neuen Ball, nämlich den Ball der um einen Schritt vorgerückt ist. Offensichtlich ist es nicht möglich, mit unserem bisherigen Entwurfsrezept eine Funktion dieser Art zu generieren.

14.2 Schnelles Sortieren

Betrachten Sie das Problem, eine Liste von Zahlen zu sortieren. Eine Verwendung unseres Entwurfsrezepts ergibt folgendes Template:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (sort l)
  (match l
    [empty ...]
    [(cons x xs) ... x ... (sort xs) ...]))
```

Im Basisfall ist die Vervollständigung des Templates trivial. Im rekursiven Fall müssen wir offensichtlich `x` in die bereits sortierte Liste `(sort xs)` einfügen. Hierzu könne wir die bereits in §11.3 "Funktionen als Funktionsparameter" definierte Funktion `insert` verwenden. Insgesamt ergibt sich damit die folgende Definition:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (sort l)
  (match l
    [empty empty]
    [(cons x xs) (insert x (sort xs))]))
```

Dieser Algorithmus, den man auch *insertion sort* nennt, ergibt sich zwangsläufig, wenn man mittels struktureller Rekursion eine Liste sortieren möchte. Allerdings ist dieser Algorithmus nicht sehr effizient. Wenn wir eine Liste `(list x-1 ... x-n)` haben, so ergibt die Expansion von `(sort (list x-1 ... x-n))` den Ausdruck `(insert x-1 (insert x-2 ... (insert x-n empty) ...))`. Im schlechtesten Fall (beispielsweise einer rückwärts sortierten Liste) benötigt `insert` so viele Berechungsschritte, wie die Liste lang ist. Da n + (n-1) + ... + 1 = n*(n+1)/2, ergibt sich, dass die Anzahl der Berechnungsschritte des Sortieralgorithmus im schlechtesten Fall quadratisch mit der Länge der Eingabe wächst.

Ein besserer Algorithmus ergibt sich, wenn wir das Problem geschickter in Teilprobleme zerlegen, als die Struktur der Daten dies suggeriert. Ein gängiger Algorithmus ist *quick sort*. Bei diesem Algorithmus wählen wir in jedem Schritt ein Element aus, beispielsweise das erste Listenelement. Dieses Element wird Pivot-Element genannt. Dann unterteilen wir den Rest der Liste in Listenelemente die kleiner (oder gleich) und solche die größer als das Pivot-Element sind. Wenn wir diese neu generierten Listen rekursiv sortieren, so können wir die Gesamtliste sortieren, indem wir die beiden sortierten Listen mit dem Pivot-Element in der Mitte aneinanderhängen.

Insgesamt ergibt sich damit folgende Definition:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; alon sorted in ascending order
(define (qsort l)
  (match l
    [empty empty]
    [(cons x xs)
     (append
      (qsort (smaller-or-equal-than x xs))
      (list x)
      (qsort (greater-than x xs)))]))

; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; smaller or equal than x
(define (smaller-or-equal-than x xs)
  (filter (lambda (y) (<= y x)) xs))

; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; greater than x
(define (greater-than x xs)
  (filter (lambda (y) (> y x)) xs))
```

Die Rekursionsstruktur in diesem Algorithmus unterscheidet sich ebenfalls deutlich von dem bekannten Muster der strukturellen Rekursion. Die Eingabe des rekursiven Aufrufs ist zwar in gewissem Sinne ein Teil der Eingabe (in dem Sinne dass die Listenelemente in den rekursiven Aufrufen eine Teilmenge der Listenelemente der Originaleingabe bilden), aber sie sind kein Teil der Eingabe im Sinne der Struktur der Eingabe. Die Fallunterscheidung in diesem Beispiel ist die gleiche, die wir auch bei struktureller Rekursion haben, aber statt einem rekursiven Aufruf wie bei struktureller Rekursion auf Listen haben wir zwei rekursive Aufrufe.

Es ist nicht ganz einfach, zu sehen, dass quick sort in der Regel viel schneller ist als insertion sort (und auch nicht Thema dieser Lehrveranstaltung), aber Sie können sehen, dass für den Fall, dass die beiden Listen `(smaller-or-equal-than x xs)` und `(greater-than x xs)` stets etwa gleich große Listen erzeugen, die Rekursionstiefe nur logarithmisch in der Länge der Liste ist. Man kann zeigen, dass die Anzahl der benötigten Rechenschritte zur Sortierung einer Liste der Länge n im Durchschnitt proportional zu n*log(n) ist.

14.3 Entwurf von generativ rekursiven Funktionen

Wir nennen Rekursionsstrukturen die nicht (notwendigerweise) dem Muster der strukturellen Rekursion entsprechen *generative Rekursion*. Eine generativ rekursive Rekursion

hat eine Struktur, die etwa wie folgt aussieht:

```
(define (generative-recursive-fun problem)
      (cond
        [(trivially-solvable? problem)
         (determine-solution problem)]
        [else
         (combine-solutions
           ... problem ...
           (generative-recursive-fun (generate-problem-1 problem))
           ...
           (generative-recursive-fun (generate-problem-
n problem)))]))
```

Dieses Template soll verdeutlichen, dass wir über beim Entwurf einer generativ rekursiven Funktion die folgenden fünf Fragen beantworten müssen:

1. Was ist ein trivial lösbares Problem?
2. Was ist die Lösung für ein trivial lösbares Problem?
3. Wie generieren wir neue Probleme, die leichter lösbar sind als das Originalproblem? Wie viele neue Probleme sollen wir generieren?
4. Wie berechnen wir aus den Lösungen der generierten Probleme die Lösung des Originalproblems? Benötigen wir hierzu nochmal das Originalproblem (oder einen Teil davon)?
5. Wieso terminiert der Algorithmus?

Die letzte Frage werden wir in nächsten Abschnitt separat behandeln. Die Antwort auf die ersten vier Fragen für das erste Beispiel oben lautet: 1) Ein Ball der bereits kollidiert. 2) Die aktuelle Position des Balles. 3) Indem wir einen Bewegungsschritt des Balles vornehmen (und ihn damit näher zur Kollisionsstelle bringen). 4) Die Lösung des generierten Problems ist die Lösung des Originalproblems; keine weitere Berechnung ist nötig.

Die Antwort auf die ersten vier Fragen für das zweite Beispiel lautet: 1) Das Sortieren einer leeren Liste. 2) Die leere Liste. 3) Indem wir ein Pivotelement auswählen und zwei neue Probleme generieren: Das Sortieren der Liste aller Elemente des Originalproblems, die kleiner (oder gleich) als das Pivotelement sind, und das Sortieren der Liste aller Elemente des Originalproblems, die größer als das Pivotelement sind. 4) Indem wir die beiden sortierten Listen mit dem Pivotelement in der Mitte zusammenhängen.

Eine generativ rekursive Funktion sollte in folgenden Situationen erwogen werden: 1) Die Eingabe hat eine rekursive Struktur, aber es ist nicht möglich oder zu kompliziert, mittels struktureller Rekursion das Problem zu lösen (beispielsweise weil das Ergebnis des rekursiven Aufrufs nicht hilft, das Originalproblem zu lösen). 2) Es gibt eine strukturell rekursive

Funktion, die das Problem löst, aber sie ist nicht effizient genug. 3) Die Eingabe ist nicht rekursiv, aber die Anzahl der Berechnungsschritte zur Lösung des Problems ist nicht proportional zur Größe der Eingabe.

Wenn Sie ein Problem mittels generativer Rekursion lösen möchten, sollten Sie als erstes die vier Fragen oben beantworten und dann im Template-Schritt des Entwurfsrezepts das oben stehende Template verwenden (angepasst auf die Antworten auf Frage 1 und 3). Mit den Antworten auf die Frage 2 und 4 können Sie dann die Implementierung des Templates vervollständigen. Die Tests für die Funktion sollten auf jeden Fall sowohl Beispiele für triviale Probleme als auch für den generativ rekursiven Fall enthalten.

Ein wichtiger Unterschied zwischen struktureller Rekursion und generativer Rekursion ist, dass der Entwurf generativ rekursiver Funktionen mehr Kreativität erfordert. Inbesondere ist eine besondere gedankliche Einsicht erforderlich, wie aus dem Problem sinnvolle kleinere Teilprobleme generiert werden können. Bei struktureller Rekursion hingegen ergibt sich die Funktionsdefinition oft schon fast zwingend aus dem Template.

14.4 Terminierung

Eine wichtige Eigenschaft von strukturell rekursiven Funktionen ist, dass diese immer terminieren: Da die Eingabedaten eine endliche Größe haben und in jedem rekursiven Aufruf die Eingabe echt kleiner wird, muss irgendwann ein nichtrekursiver Basisfall erreicht werden.

Dies ist bei generativer Rekursion anders: Wir müssen explizit überlegen, warum eine generativ rekursive Funktion terminiert.

Betrachten Sie hierzu eine Variante des `qsort` Algorithmus von oben, in dem wir den Ausdruck `(smaller-or-equal-than x xs)` ersetzen durch `(smaller-or-equal-than x l)`. Statt also nur aus der Restliste (ohne Pivotelement) die kleiner-oder-gleichen Elemente herauszusuchen, suchen wir in der Liste, die auch das Pivotelement noch enthält:

```
(define (qsort l)
  (match l
    [empty empty]
    [(cons x xs)
     (append
      (qsort (smaller-or-equal-than x l))
      (list x)
      (qsort (greater-than x xs)))]))
```

Betrachten wir nun einen Aufruf wie `(qsort (list 5))`. Da `(smaller-or-equal-than 5 (list 5))` die Liste `(list 5)` ergibt, wird dieser Aufruf zu einem rekursiven Aufruf der Form `(qsort (list 5))` führen. Wir haben also eine Endlosschleife produziert.

Wie können wir diesen Fehler vermeiden? Die Terminierung einer generativ rekursiven Funktion kann durch zwei Schritte gezeigt werden:

1. Wir definieren eine Abbildung, die den Satz von Funktionsargumenten auf eine natürliche Zahl abbildet. Diese Abbildung misst quasi die Größe der Eingabe, wobei "Größe" nicht notwendigerweise die physikalische Größe der Daten im Speicher beschreibt sondern die Größe des Problems aus Sicht des Algorithmus.
2. Wir zeigen, dass die Größe der Eingabe bzgl. der Abbildung aus dem ersten Schritt in allen rekursiven Funktionsaufrufen strikt kleiner wird.

Falls die Größe der Originaleingabe (bzgl. der definierten Abbildung) also n ist, so ist sichergestellt, dass die maximale Rekursionstiefe ebenfalls n ist.

Im Falle von quick sort können wir als Abbildung im ersten Schritt die Länge der Liste `l` verwenden. Falls die Länge von `(cons x xs)` n ist, so ist die Länge von `xs` n-1 und damit sind auch `(smaller-or-equal-than x xs)` und `(greater-than x xs)` nicht größer als n-1. Daher wird die Größe der Eingabe in allen rekursiven Aufrufen strikt kleiner.

Im Falle von `first-collision` ist es deutlich komplizierter, die Terminierung zu zeigen. Überlegen Sie, wie in diesem Fall die Größe der Eingabe gemessen werden kann, so dass die Bedingung aus dem zweiten Schritt gilt. Hinweis: Tatsächlich terminiert `first-collision` nicht immer. Verwenden Sie die Suche nach einem Terminierungsbeweis dazu, um diesen Fehler zu finden und zu beheben.