15 Akkumulation von Wissen

Das Ergebnis eines Funktionsaufrufs hängt nur von den Funktionsparametern ab, nicht jedoch vom Kontext in dem eine Funktion aufgerufen wird. Auf der einen Seite macht diese Eigenschaft Funktionen sehr flexibel, denn sie können in jedem Kontext einfach aufgerufen werden. Auf der anderen Seite sind jedoch manche Probleme so, dass die Problemstellung eine gewisse Abhängigkeit vom Kontext erfordert.

Dies ist insbesondere beim Entwurf rekursiver Funktionen wichtig. Hierzu zwei Beispiele.

15.1 Beispiel: Relative Distanz zwischen Punkten

Nehmen Sie an, wir bekommen eine Liste von Distanzen zwischen Punkten, zum Beispiel `(list 0 50 40 70 30 30)`. Der erste Punkt hat also die Entfernung 0 vom Ursprung, der zweite Punkt die Entfernung 50. Der dritte Punkt ist vom zweiten Punkt 40 entfernt, also vom Ursprung 50+40=90 entfernt.

Nehmen wir an, wir möchten diese Liste umformen zu einer Liste mit absoluten Distanzen zum Ursprung. Wir können auf Basis dieser Informationen und des Beispiels mit der Implementierung dieser Funktion starten:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute dis-
tances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon) ...)
```

Gemäß unseres Entwurfsrezepts können wir versuchen, dieses Problem mittels struktureller Rekursion zu lösen:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute dis-
tances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [empty ...]
    [(cons x xs) ... x ... (relative-2-absolute xs) ...]))
```

Der Fall für die leere Liste ist trivial, aber wie können wir aus dem Ergebnis von `(relative-2-absolute xs)` und `x` das Gesamtergebnis berechnen? Das erste Element

des Ergebnisses ist `x`, aber offensichtlich müssen wir zu jedem Element von `(relative-2-absolute xs)` `x` addieren. Mit dieser Erkenntnis können wir die Funktionsdefinition vervollständigen:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute dis-
tances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [empty empty]
    [(cons x xs) (cons x
                       (map
                        (lambda (y) (+ y x))
                        (relative-2-absolute xs)))]))
```

Obwohl diese Funktion wie gewünscht funktioniert, ist sie problematisch. Wenn man beispielsweise mit Hilfe der `time` Funktion testet, wie lange die Funktion zur Bearbeitung von Listen verschiedener Länge benötigt, so kann man sehen, dass die benötigte Zeit quadratisch mit der Länge der Liste wächst.

Dieses Phänomen kann man auch direkt in der Funktionsdefinition sehen, denn durch den Aufruf von `map` wird in jedem Rekursionsschritt die komplette bisher berechnete Liste nochmal bearbeitet.

Wenn wir diese Berechnung von Hand durchführen, so gehen wir anders vor. Wir gehen die Liste nur einmal von links nach rechts durch und merken uns die Summe der Zahlen, die wir bisher gesehen haben.

Diese Form der Buchführung passt allerdings nicht zu unserem bisherigen Schema für Rekursion. Wenn wir zwei Listen `(cons x1 xs)` und `(cons x2 xs)` haben, so hat der rekursive Aufruf in beiden Fällen die Form `(f xs)`. Was `f` mit `xs` macht, kann also nicht von `x1` oder `x2` abhängen.

Um dieses Problem zu lösen, führen wir einen zusätzlichen Parameter ein: `accu-dist`. Dieser Parameter repräsentiert die akkumulierte Distanz der bisher gesehenen Punkte. Wenn wir mit der Berechnung starten, wird `accu-dist` mit `0` initialisiert. Während der Berechnung können wir die erste relative Distanz in eine absolute Distanz umwandeln, indem wir `accu-dist` hinzuaddieren; für die Berechnung im rekursiven Aufruf müssen wir den Wert des Akkumulators auf den neuen Kontext anpassen.

Insgesamt ergibt sich damit der folgende Code:

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
  (match alon
```

```
    [empty empty]
    [(cons x xs)
     (local [(define x-absolute (+ accu-dist x))]
       (cons x-absolute (relative-2-absolute-with-acc xs x-
absolute)))]))
```

Diese Definition ist noch nicht ganz äquivalent zu `relative-2-absolute`, aber wir können die ursprüngliche Signatur leicht rekonstruieren:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute dis-
tances

(check-expect (relative-2-absolute-2 (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))

(define (relative-2-absolute-2 alon)
  (local
    ; (list-of Number) Number -> (list-of Number)
    [(define (relative-2-absolute-with-acc alon accu-dist)
       (match alon
         [empty empty]
         [(cons x xs)
          (local [(define x-absolute (+ accu-dist x))]
            (cons x-absolute
                  (relative-2-absolute-with-acc xs x-
absolute)))]))]
    (relative-2-absolute-with-acc alon 0)))
```

Einige Experimente mit `time` bestätigen, dass `relative-2-absoluste-2` viel effizienter als `relative-2-absoluste` ist und statt einem quadratischen nur noch ein lineares Wachstum der Laufzeit aufweist.

15.2 Beispiel: Suche in einem Graphen

Als zweites Beispiel wollen wir uns das Problem anschauen, in einem Graphen einen Weg zwischen zwei Knoten zu finden (sofern dieser existiert).

Ein gerichteter Graph ist eine Menge von Knoten und eine Menge von gerichteten Kanten zwischen den Knoten. Es gibt unterschiedliche Möglichkeiten, Graphen zu repräsentieren. Wir entscheiden uns für eine Repräsentation, bei der zu jedem Knoten gespeichert wird, welche Kanten von diesem Knoten ausgehen:

```
; A Node is a symbol
```

```
; A Node-with-neighbors is a (list Node (list-of Node))

; A Graph is a (list-of Node-with-neighbors)
```

Hier ist ein Beispiel für einen Graphen, der dieser Datendefinition entspricht:

```
(define graph1
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

Wir suchen also eine Funktion, deren Aufgabe wir wie folgt definieren können:

```
; Node Node Graph  ->  (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G) ...)
```

Zunächst einmal können wir feststellen, dass wir dieses Problem mit struktureller Rekursion nicht sinnvoll lösen können. Wenn wir zum Beispiel in `graph1` eine Route von B nach E suchen, nützt uns die Information, ob es in `(rest graph1)` eine solche Route gibt, nicht viel, denn es könnte ja sein, dass die Route durch A geht.

Um dieses Problem mit generativer Rekursion zu lösen, müssen wir die in §14.3 “Entwurf von generativ rekursiven Funktionen” beschriebenen fünf Fragen beantworten:

1. Ein trivial lösbares Problem ist die Frage nach einer Route von einem Knoten `n` zu sich selbst.
2. Die Lösung ist im trivialen Fall `(list n)`.
3. Wenn das Problem nicht-trivial ist, können wir für jeden Nachbarn des Knoten das Problem generieren, eine Route von dem Nachbarn zum Ziel zu finden.
4. Wenn eines dieser Probleme erfolgreich gelöst wird, so ist das Gesamtergebnis der Weg zum Nachbarn gefolgt von der gefundenen Route vom Nachbarn aus.
5. Das Terminierungsargument verschieben wir auf später.

Insgesamt ergibt sich damit das folgende Programm:

```
; Node Node Graph  ->  (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G) destination G)))
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))

; (list-of Node) Node Graph  ->  (list-of Node) or false
; to create a path from some node on lo-Os to D
; if there is no path, the function produces false
(check-member-of (find-route/list '(E F) 'G graph1) '(F G) '(E F G))
(define (find-route/list lon D G)
  (match lon
    [empty false]
    [(cons n ns)
     (local ((define possible-route (find-route n D G)))
       (cond
         [(boolean? possible-route) (find-route/list ns D G)]
         [else possible-route]))]))


; Node Graph -> (list-of Node)
; computes the set of neighbors of node n in graph g
(check-expect (neighbors 'B graph1) '(E F))
(define (neighbors n g)
  (match g
    [(cons (list m m-neighbors) rest)
     (if (symbol=? m n)
         m-neighbors
         (neighbors n rest))]
    [empty (error "node not found")]))
```

Algorithmen wie `find-route` nennt man auch *Backtracking Algorithmen*, weil sie systematisch Alternativen ausprobieren und beim Erschöpfen der lokal sichtbaren Alternativen zurückspringen zum nächsten Punkt, an dem noch nicht alle Alternativen erschöpft sind.

Kommen wir nun zurück zu der Frage, ob der Algorithmus immer terminiert. Woher wissen wir, dass wir wirklich "näher am Ziel" sind, wenn wir von einem Knoten zu seinem Nachbarn

gehen? "Näher zum Ziel" heißt in diesem Fall, dass wir entweder näher am Zielknoten sind, oder dass wir die Zahl der Alternativen, die noch ausprobiert werden müssen, verringert haben.

Allerdings kann man relativ leicht sehen, dass dieser Algorithmus tatsächlich nicht immer terminiert, nämlich dann, wenn der Graph Zyklen enthält. Wenn wir beispielsweise den Graph

```
(define graph2
  '((A (B E))
    (B (A E F))
    (C (D))
    (D (C))
    (E (C F))
    (F (D G))
    (G (D))))
```

betrachten und den Ausdruck

```
(find-route 'A 'G graph2)
```

auswerten, so stellen wir fest, dass die Auswertung nicht terminiert, weil wir entlang der Route `'(A B A B A B ...)` im Kreis laufen und keinerlei Fortschritt machen: Ein Aufruf von

```
(find-route 'A 'G graph2)
```

bewirkt einen Aufruf von

```
(find-route/list '(B E) 'G graph2)
```

; dieser wiederum zieht einen Aufruf von

```
(find-route 'A 'G graph2)
```

nach sich.

Das Problem ist, genau wie im `relative-2-absolute` Beispiel oben, dass wir nichts über den Kontext wissen, in dem `find-route` aufgerufen wird. In diesem Beispiel wollen wir wissen, ob `find-route` mit einem Startknoten aufrufen, von dem wir bereits gerade versuchen, eine Route zum Ziel zu finden.

Der Akkumulationsparameter, den wir in diesem Beispiel benötigen, repräsentiert die Liste der Knoten, die wir bereits auf dem Pfad, der konstruiert wird, besucht haben. Ein solcher Parameter läßt sich leicht hinzufügen. Da `find-route` und `find-route/list` wechselseitig rekursiv sind, wird der zusätzliche Parameter durch beide Funktionen durchgereicht. Wenn innerhalb von `find-route` `find-route/list` aufgerufen wird, so wird durch den Ausdruck `(cons origination visited)` die Invariante, dass `visited` stets die Menge der bereits besuchten Knoten repräsentiert, sichergestellt.

```
(define (find-route origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G)
                                     destination
                                     G
                                     (cons origination visited))))
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))

(define (find-route/list lon D G visited)
  (match lon
    [empty false]
    [(cons n ns)
     (local ((define possible-route (find-route n D G visited)))
       (cond
         [(boolean? possible-route)
          (find-route/list ns D G visited)]
         [else possible-route]))]))
```

Das reine Berechnen und Durchreichen des Akkumulationsparameters ändert allerdings nichts grundlegendes am Programmverhalten. Insbesondere terminiert der Algorithmus weiterhin nicht bei zyklischen Graphen.

Aber es ist nun leicht, das Programm so zu ändern, dass bereits besuchte Knoten nicht noch einmal besucht werden, nämlich indem wir in `find-route` von der Liste der Nachbarn `(neighbors origination G)` die Knoten abziehen, die bereits in `visited` enthalten sind.

Hierzu definieren wir eine kleine Hilfsfunktion:

```
; [X] (list-of X) (list-of X) -> (list-of X)
; removes all occurences of members of l1 from l2
(check-expect (remove-all '(a b) '(c a d b e a b g))
              '(c d e a b g))
(define (remove-all l1 l2)
```

```
  (match l1
    [empty l2]
    [(cons x xs) (remove-all xs (remove x l2))]))
```

und können damit `find-route` wie oben beschrieben ändern:

```
; Node Node Graph (list-of Node) ->  (list-of Node) or false
; to create a path from origination to destination in G
; that does not use nodes in visited.
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(check-member-of (find-route 'A 'G graph2) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local
            [(define possible-route
                    (find-route/list
                     (remove-all visited (neighbors origination G))
                     destination
                     G
                     (cons origination visited)))]
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))
```

Der Test mit `graph2` suggeriert, dass der Algorithmus nun auch auf Graphen mit Zyklen terminiert. Wie sieht das Argument für den allgemeinen Fall aus?

Gemäß der in §14.4 “Terminierung” beschriebenen Methodik müssen wir eine Abbildung definieren, die die Funktionsargumente von `find-route` auf eine natürliche Zahl abbildet. Sei n die Zahl der Knoten in `G` und m die Zahl der Knoten in `visited`. Dann definieren wir die Größe der Eingabe von `find-route` als n-m, also die Anzahl der Knoten, die noch nicht besucht worden.

Die Differenz n-m ist stets eine natürliche Zahl, weil durch das Entfernen der bereits besuchten Knoten aus den Nachbarn (`(remove-all visited (neighbors origination G))`) im Aufruf `(cons origination visited)` stets nur Knoten aus dem Graph hinzugefügt werden, die vorher noch nicht in `visited` enthalten waren. Daher ist `visited` stets eine Teilmenge der Knoten des Graphs; kein Knoten kommt in `visited` mehrfach vor.

Bezüglich des zweiten Schritts aus §14.4 “Terminierung” können wir festhalten, dass die Größe n-m durch das Hinzufügen eines neuen Knoten in `(cons origination visited)` stets strikt kleiner wird.

Der rekursive Aufruf in `find-route` ist jedoch indirekt via `find-route/list`. Diese

Funktion ist strukturell rekursiv und terminiert daher immer. Desweiteren können wir festhalten, dass `find-route/list` sowohl `G` als auch `visited` stets unverändert durchreicht. Wenn `find-route/list` daher `find-route` aufruft, so tut es dies mit unveränderten Werten für `G` und `visited`.

Insgesamt können wir festhalten, dass wir damit gezeigt haben, dass die Rekursionstiefe durch die Anzahl der Knoten des Graphen begrenzt ist und damit der Algorithmus für alle Graphen terminiert.

15.3 Entwurf von Funktionen mit Akkumulatoren

Nachdem wir zwei Beispiele gesehen haben, in denen ein Akkumulator sinnvoll ist, wollen wir nun diskutieren, wann und wie man im Allgemeinen Funktionen mit Akkumulatoren entwerfen sollte.

Zunächst mal sollte man einen Funktionsentwurf mit Akkumulator nur dann in Erwägung ziehen, wenn der Versuch, mit dem Standard-Entwurfsrezept die Funktion zu entwerfen, gescheitert ist, oder zu zu kompliziertem oder zu langsamem Code führt.

Die Schlüsselaktivitäten beim Entwurf einer Funktion mit Akkumulator sind: 1) zu erkennen, dass die Funktion einen Akkumulator benötigt (oder davon profitieren würde), und 2) zu verstehen, was genau der Akkumulator repräsentiert (die *Akkumulator-Invariante*).

15.3.1 Wann braucht eine Funktion einen Akkumulator

Wir haben zwei Gründe gesehen, wieso Funktionen einen Akkumulator benötigen:

1. Wenn eine Funktion strukturell rekursiv ist und das Ergebnis des rekursiven Aufrufs wieder von einer rekursiven Hilfsfunktion verarbeitet wird. Häufig kann durch einen Akkumulator die Funktionalität der rekursiven Hilfsfunktion in die Hauptfunktion mit eingebaut werden und statt verschachtelter Iterationen (die häufig zu quadratischer Laufzeit führen) ist dann häufig eine einfache Iteration ausreichend.

   Hier ein weiteres Standardbeispiel:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
; structural recursion
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item (first alox) (invert (rest alox)))]))
```

```
; make-last-item : X (listof X)  ->  (listof X)
; to add an-x to the end of alox
; structural recursion
(define (make-last-item an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (make-last-item an-
x (rest alox)))]))
```

2. Wenn wir eine generativ rekursive Funktion haben, dann kann es schwierig sein, diese Funktion so zu entwerfen, dass sie für alle Eingaben eine korrekte Ausgabe berechnet. Im Beispiel oben haben wir gesehen, dass wir uns merken mussten, welche Knoten wir bereits besucht haben, um Terminierung bei Zyklen im Graph sicherzustellen. Im Allgemeinen können wir mit einem Akkumulator beliebiges Wissen über die aktuelle Iteration akkumulieren und nutzen. Wenn es also Wissen gibt, welches nicht lokal verfügbar ist sondern nur im Verlauf der Iteration angesammelt werden kann, so ist ein Akkumulator das richtige Mittel.

Diese beiden Möglichkeiten sind nicht die einzigen, aber sie sind sehr häufig.

15.3.2 Template für Funktionen mit Akkumulatoren

Wenn wir uns dafür entschieden haben, eine Funktion mit Akkumulator auszustatten, so ist es sinnvoll, ein Template für die Funktionsdefinition zu erstellen. Dieses sieht so aus, dass wir die Funktion mit Akkumulator zu einer mit `local` definierten lokalen Funktion der eigentlich zu definierenden Funktion machen und diese Funktion dann im Body des `local` Ausdrucks aufrufen.

Im Beispiel von oben sieht diese Template so aus:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator ...
          (define (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) ... ( first alox) ... accumulator)
                ...])))
    (rev alox0 ...)))
```

15.3.3 Die Akkumulator-Invariante

Als nächstes ist es sinnvoll, die Akkumulator-Invariante zu formulieren. Die Akkumulator-Invariante sagt, was der Akkumulator in jedem Iterationsschritt repräsentiert.

Zu diesem Zweck müssen wir uns überlegen, welche Daten wir akkumulieren wollen, so dass der Akkumulator uns bei der Implementation der Funktion hilft.

Im Falle von `invert` würde es uns helfen, wenn der Akkumulator die Listenelemente repräsentiert, die wir bisher gesehen haben (in umgekehrter Reihenfolge), denn dann können wir diese im `empty` Fall zurückgeben statt `make-last-item` aufzurufen.

Diese Akkumulatorinvariante sollten wir in den Code hineinschreiben:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; ;; accumulator is the reversed list of all those items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) ... ( first alox) ... accumulator)
                ...])))
    (rev alox0 ...)))
```

15.3.4 Implementation der Akkumulator-Invariante

Der nächste Schritt ist, dafür zu sorgen, dass die Akkumulator-Invariante auch tatsächlich eingehalten wird. Dies bedeutet, dass wir uns den initialen Wert für den Akkumulator überlegen müssen sowie die Berechnung, wie wir im rekursiven Aufruf den neuen Akkumulator erhalten. In unserem Beispiel ist offensichtlich der initiale Akkumulator `empty` und der neue Akkumulator im rekursiven Aufruf `(cons (first alox) accumulator)`:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) (cons (first alox) accumulator))
```

```
                ...])))
    (rev alox0 empty)))
```

15.3.5 Nutzung des Akkumulators

Die Akkumulator-Invariante zu implementieren ändert noch nichts am externen Verhalten der Funktion. Der Sinn der ganzen Operation liegt darin, dass wir nun den Akkumulator nutzen können, um die Funktion zu implementieren. Wenn wir die Akkumulator-Invariante präzise formuliert haben, ist dieser Schritt typischerweise einfach:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) accumulator]
              [else
                (rev (rest alox) (cons (first alox) accumulator))])))
    (rev alox0 empty)))
```