Programmierung und Modellierung mit Haskell

Rekursion

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

14. April 2014

Rekursion

Man kann eine Funktion $f : A \to B$ durch einen Ausdruck definieren, der selbst Funktionsanwendungen von f enthält.
Beispiele:

- Fakultät:

$$\text{fakultät}(n) = \begin{cases} 1, \text{ falls } n = 0 \\ n \cdot \text{fakultät}(n-1), \text{ sonst} \end{cases}$$

- Summe der ersten n natürlichen Zahlen:

$$\text{summe}(n) = \begin{cases} 0, \text{ falls n=0} \\ n + \text{summe}(n-1), \text{ sonst} \end{cases}$$

Dies bezeichnet man als **rekursive Definition**. Auswerten durch Einsetzen der definierenden Gleichungen von links nach rechts:

$$\begin{aligned} \text{fakultät}(3) &\leadsto 3 \cdot \text{fakultät}(3-1) \leadsto 3 \cdot \text{fakultät}(2) \\ &\leadsto 3 \cdot 2 \cdot \text{fakultät}(2-1) \leadsto 3 \cdot 2 \cdot \text{fakultät}(1) \\ &\leadsto 3 \cdot 2 \cdot 1 \cdot \text{fakultät}(0) \leadsto 3 \cdot 2 \cdot 1 \cdot 1 \leadsto 6 \end{aligned}$$

Rekursion

Man kann eine Funktion $f : A \to B$ durch einen Ausdruck definieren, der selbst Funktionsanwendungen von f enthält.
Beispiele:

- Fakultät:

```
fakultät 0 = 1
fakultät n = n * fakultät (n-1)
```

- Summe der ersten n natürlichen Zahlen:

```
summe 0 = 0
summe n = n + summe (n-1)
```

Dies bezeichnet man als **rekursive Definition**. Auswerten durch Einsetzen der definierenden Gleichungen von links nach rechts:

$$\begin{aligned}\text{fakultät}(3) &\rightsquigarrow 3 \cdot \text{fakultät}(3-1) \rightsquigarrow 3 \cdot \text{fakultät}(2)\\ &\rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(2-1) \rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(1)\\ &\rightsquigarrow 3 \cdot 2 \cdot 1 \cdot \text{fakultät}(0) \rightsquigarrow 3 \cdot 2 \cdot 1 \cdot 1 \rightsquigarrow 6\end{aligned}$$

Substitutionsmodell

Das Substitutionsmodell erklärt das Verhalten von rein funktionalen Sprachen, solange keine Fehler auftreten:

- Man wählt einen Teilausdruck und wertet diesen gemäß den geltenden Rechenregeln aus.
- Eine Funktionsanwendung wird durch den definierenden Rumpf ersetzt. Dabei werden die formalen Parameter im Rumpf durch die Argumentausdrücke ersetzt ("substitutiert").
- Dies wiederholt man, bis keine auswertbaren Teilausdrücke mehr vorhanden sind.

Es gibt verschiedene unterschiedliche Strategien, den als nächstes auszuwertenden Teilausdruck auszuwählen.

Dies ist ein rekursiver Algorithmus!

Fibonacci-Zahlen

```
fib 0 = 0
fib n | n <= 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

> fib 8
21
> fib 9
34
```

- `fib` n liefert die n-te Fibonacci-Zahl F_n:
  $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- `fib` n liefert z.B. die Hasenpopulation nach n Monaten unter der Annahme, dass Hasen jeden Monat einen Nachkommen haben, dies aber erst ab dem zweiten Lebensmonat.
- `fib` n liefert auch die Zahl der Möglichkeiten, ein $2 \times n$ Zimmer mit 1×2 Kacheln zu fliesen.

Collatz Vermutung

Collatz-Vermutung: Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.

Falls es ein i gibt, dann auch ein kleinstes. Wir versuchen dieses rekursiv zu berechnen:

```
f n | even n          = n `div` 2
    | otherwise       = 3*n+1

min_i 1 = 0
min_i n = 1 + min_i (f n)

> [min_i n | n <- [2..10]++[26,27,28,29]]
[1,7,2,5,8,16,3,19,6,10,111,18,18]
```

Collatz Vermutung

Collatz-Vermutung: Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.
Falls es ein i gibt, dann auch ein kleinstes. Wir versuchen dieses rekursiv zu berechnen:

```
f n | even n          = n `div` 2
    | otherwise       = 3*n+1

min_i 1 = 0
min_i n = 1 + min_i (f n)

> [min_i n | n <- [2..10]++[26,27,28,29]]
[1,7,2,5,8,16,3,19,6,10,111,18,18]
```

Collatz Vermutung

Collatz-Vermutung: Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.
Falls es ein i gibt, dann auch ein kleinstes. Wir versuchen dieses rekursiv zu berechnen:

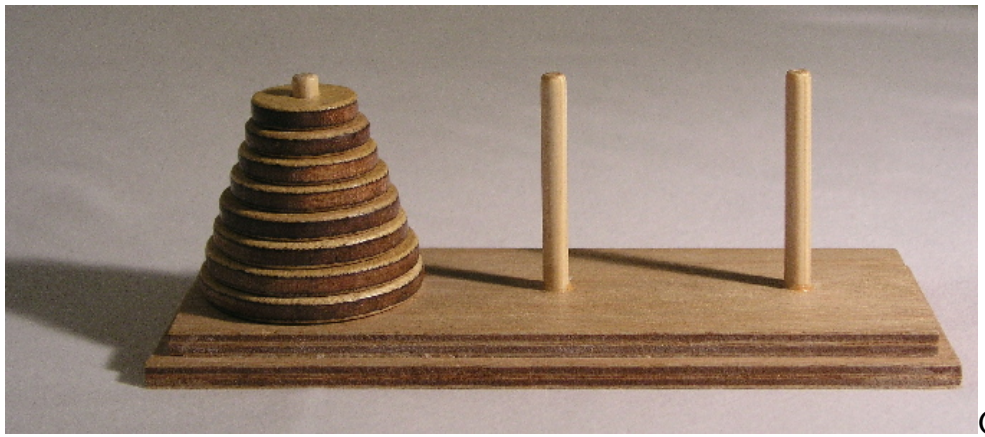
```
f n | n `mod` 2 == 0 = n `div` 2
    | otherwise      = 3*n+1

min_i 1 = 0
min_i n = 1 + min_i (f n)

> [min_i n | n <- [2..10]++[26,27,28,29]]
[1,7,2,5,8,16,3,19,6,10,111,18,18]
```

Türme von Hanoi

Es gibt drei senkrechte Stäbe. Auf dem ersten liegen n gelochte Scheiben von nach oben hin abnehmender Größe.

Man soll den ganzen Stapel auf den dritten Stab transferieren, darf aber immer nur jeweils eine Scheibe entweder nach ganz unten oder auf eine größere legen.

Angeblich sind in Hanoi ein paar Mönche seit Urzeiten mit dem Fall $n = 64$ befasst.



Quelle: Wikipedia

Lösung

Für $n = 1$ kein Problem.

Falls man schon weiß, wie es für $n - 1$ geht, dann schafft man mit diesem Rezept die obersten $n - 1$ Scheiben auf den zweiten Stab (die unterste Scheibe fasst man dabei als "Boden" auf.).

Dann legt man die größte nunmehr freie Scheibe auf den dritten Stapel und verschafft unter abermaliger Verwendung der Vorschrift für $n - 1$ die restlichen Scheiben vom mittleren auf den dritten Stapel.

Lösung in Haskell

- "Türme" werden durch Zahl 1,2,3 repräsentiert
- "Befehle" durch Paare (i, j): "Bewege Scheibe von i nach j"
- "Befehlsfolgen" werden durch Listen repräsentiert.

```
hanoi 1 i j = [(i,j)]
hanoi n i j = hanoi n' i otherT ++ [(i,j)] ++ hanoi n' otherT j
  where n'      = n-1
        otherT = 1+2+3-i-j                          -- other tower
```

```
> hanoi 3 1 2
[(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)]
```

```
> hanoi 4 1 2
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)
,(1,2)
,(3,2),(3,1),(2,1),(3,2),(1,3),(1,2),(3,2)]
```

Allgemeines Muster einer Rekursion

Eine rekursive Definition einer Funktion $f : A \to B$ hat die Form

$$f(a) = E[f, a]$$

wobei im Ausdruck $E[f, a]$, also dem Funktionsrumpf, sowohl das Argument a, als auch (endlich viele) Aufrufe an die definierte Funktion f selbst vorkommen dürfen.

Keine rekursiven Definitionen sind also:

- $f(n) = \begin{cases} 0, \text{ falls Programm für } f \text{ kürzer als } n \text{ KB} \\ 1, \text{ sonst} \end{cases}$

  Zugriff nicht in Form von Aufrufen

- $f(n) = \begin{cases} 1, \text{ falls } f(i) = 0 \text{ für alle } i \in \mathbb{N} \\ 0, \text{ sonst} \end{cases}$

  unendliche viele Aufrufe

Allgemeines Muster einer Rekursion

Eine rekursive Definition einer Funktion $f : A \to B$ hat die Form

$$f(a) = E[f, a]$$

wobei im Ausdruck $E[f, a]$, also dem Funktionsrumpf, sowohl das Argument a, als auch (endlich viele) Aufrufe an die definierte Funktion f selbst vorkommen dürfen.

Keine rekursiven Definitionen sind also:

- $f(n) = \begin{cases} 0, \text{ falls Programm für } f \text{ kürzer als } n \text{ KB} \\ 1, \text{ sonst} \end{cases}$

  Zugriff nicht in Form von Aufrufen

- $f(n) = \begin{cases} 1, \text{ falls } f(i) = 0 \text{ für alle } i \in \mathbb{N} \\ 0, \text{ sonst} \end{cases}$

  unendliche viele Aufrufe

Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `summe` und `fakultät` liefern bei negativen Eingaben kein Ergebnis:

```
> summe (-3)
Heap exhausted;
```

Tipp: Manche PCs werden unbenutzbar, wenn ein Programm den kompletten Speicher belegt. GHC und GHCI erlaubt es, den maximal verwendeten Speicher zu beschränken, hier z.B. auf 2 GB: `> ghci file.hs +RTS -M2g`

Grund: $\texttt{summe}(-2) \leadsto -2 + \texttt{summe}(-3) \leadsto$
$-2 - 3 + \texttt{summe}(-4) \leadsto -2 - 3 - 4 + \texttt{summe}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "3n + 1"-Funktion (siehe 1. Folie 6): weiß man nicht

Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `summe` und `fakultät` liefern bei negativen Eingaben kein Ergebnis:

```
> summe (-3)
Heap exhausted;
```

  Tipp: Manche PCs werden unbenutzbar, wenn ein Programm den kompletten Speicher belegt. GHC und GHCI erlaubt es, den maximal verwendeten Speicher zu beschränken, hier z.B. auf 2 GB: `> ghci file.hs +RTS -M2g`

  Grund: $\texttt{summe}(-2) \leadsto -2 + \texttt{summe}(-3) \leadsto$
  $-2 - 3 + \texttt{summe}(-4) \leadsto -2 - 3 - 4 + \texttt{summe}(-5) \leadsto \ldots$

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "3n + 1"-Funktion (min. i. Folie 6): weiß man nicht

Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `summe` und `fakultät` liefern bei negativen Eingaben kein Ergebnis:

```
> summe (-3)
Heap exhausted;
```

  Grund: $\texttt{summe}(-2) \leadsto -2 + \texttt{summe}(-3) \leadsto$
  $-2 - 3 + \texttt{summe}(-4) \leadsto -2 - 3 - 4 + \texttt{summe}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n+1$"-Funktion (`min_i`, Folie 6); weiß man nicht, ob sie für alle n definiert ist.

Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `summe` und `fakultät` liefern bei negativen Eingaben kein Ergebnis:

```
> summe (-3)
Heap exhausted;
```

  Grund: $\texttt{summe}(-2) \leadsto -2 + \texttt{summe}(-3) \leadsto$
  $-2 - 3 + \texttt{summe}(-4) \leadsto -2 - 3 - 4 + \texttt{summe}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n+1$"-Funktion (`min_i`, Folie 6); weiß man nicht, ob sie für alle n definiert ist.

Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `summe` und `fakultät` liefern bei negativen Eingaben kein Ergebnis:

```
> summe (-3)
Heap exhausted;
```

  Grund: $\texttt{summe}(-2) \leadsto -2 + \texttt{summe}(-3) \leadsto$
  $-2 - 3 + \texttt{summe}(-4) \leadsto -2 - 3 - 4 + \texttt{summe}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n + 1$"-Funktion (`min_i`, Folie 6); weiß man nicht, ob sie für alle n definiert ist.

Abstiegsfunktion

Man kann eine **Abstiegsfunktion** verwenden, um festzustellen ob eine rekursiv Funktion für ein Argument definiert ist.

Sei

$$f : A \to B$$
$$f(x) = E[f, x]$$

eine rekursive Definition einer Funktion $f : A \to B$.

AUF) Sei $A' \subseteq A$ eine Teilmenge von A und werde in $E[f, x]$ wobei $x \in A'$ die Funktion f nur für Argumente $y \in A'$ aufgerufen.

DEF) Sei für $x \in A'$ der Ausdruck $E[f, x]$ definiert unter der Annahme, dass die getätigten Aufrufe von f alle definiert sind.

Damit weiß man erst einmal, dass die Aufrufe definiert sind (also keine unvollständigen Patterns, etc.), aber es muss noch nicht $A' \subseteq \mathrm{dom}(f)$ gelten! Gegenbeispiel: $E[f, x] = f(x)$

Abstiegsfunktion

Man kann eine **Abstiegsfunktion** verwenden, um festzustellen ob eine rekursiv Funktion für ein Argument definiert ist.

Sei

$$f : A \to B$$
$$f(x) = E[f, x]$$

eine rekursive Definition einer Funktion $f : A \to B$.

AUF) Sei $A' \subseteq A$ eine Teilmenge von A und werde in $E[f, x]$ wobei $x \in A'$ die Funktion f nur für Argumente $y \in A'$ aufgerufen.

DEF) Sei für $x \in A'$ der Ausdruck $E[f, x]$ definiert unter der Annahme, dass die getätigten Aufrufe von f alle definiert sind.

Damit weiß man erst einmal, dass die Aufrufe definiert sind (also keine unvollständigen Patterns, etc.), aber es muss noch nicht $A' \subseteq \mathrm{dom}(f)$ gelten! Gegenbeispiel: $E[f, x] = f(x)$

Abstiegsfunktion

Sei nun zusätzlich $m : A \to \mathbb{N}$ eine Funktion mit $A' \subseteq \mathrm{dom}(m)$ und der folgenden Eigenschaft:

ABST) Im Rumpf $E[f, x]$ wird f nur für solche $y \in A'$ aufgerufen, für die gilt $m(y) < m(x)$.

AUF+DEF+ABST: Dann gilt $A' \subseteq \mathrm{dom}(f)$.

Man bezeichnet so ein m als **Abstiegsfunktion**.

Die Abstiegsfunktion bietet also ein Maß für die Argumente einer Funktion, welches für die Argumente von rekursiven Ausdrücken immer kleiner wird.

Hinweis: Mit $\mathrm{dom}(f)$ bezeichnet man üblicherweise den Definitionsbereich (engl. Domain) einer Funktion f

Beispiel Fakultät

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = \begin{cases} 1, \text{ falls } x = 0 \\ x \cdot f(x-1), \text{ sonst} \end{cases}$$

Wir nehmen $A' = \mathbb{N}$ und $m(x) = \max(x, 0)$.
In $E[f, x]$ wird f einmal mit Argument $x - 1$ aufgerufen, falls $x \neq 0$ und gar nicht aufgerufen, falls $x = 0$.

AUF Wenn $x \in A'$ und $x \neq 0$ (nur dann kommt es zum Aufruf), dann ist $x - 1 \in A'$;

DEF Der Rumpf $E[f, x]$ enthält nur überall definierte Ausdrücke;

ABST Wenn $x \in A'$ und $x \neq 0$, so ist $m(x - 1) < m(x)$.

Also gilt $\mathbb{N} \subseteq \mathrm{dom}(f)$: die durch obiges Schema definierte rekursive Funktion terminiert für alle $x \in \mathbb{N}$.

Komplizierte Abstiegsfunktion

Gegeben ist die rekursive Funktion f als:

$$foosum(i, n, a) = \begin{cases} a, \text{ falls } i = n \\ foosum(i+1, n, a+i), \text{ sonst} \end{cases}$$

$$foosum(3,6,10) \rightsquigarrow foosum(4,6,13) \rightsquigarrow foosum(5,6,17)$$
$$\rightsquigarrow foosum(6,6,22) \rightsquigarrow 22$$

Wenn man wählt

- $A' = \{(i, n, a) \mid n \in \mathbb{N}, 0 \leq i \leq n, a \in \mathbb{Z}\}$
- $m(i, n, a) = \max(n - i, 0)$

kann man die Terminierung von f auf A' beweisen.

Komplizierte Abstiegsfunktion

Gegeben ist die rekursive Funktion f als:

$$foosum(i, n, a) = \begin{cases} a, \text{ falls } i = n \\ foosum(i + 1, n, a + i), \text{ sonst} \end{cases}$$

$$foosum(3, 6, 10) \rightsquigarrow foosum(4, 6, 13) \rightsquigarrow foosum(5, 6, 17)$$
$$\rightsquigarrow foosum(6, 6, 22) \rightsquigarrow 22$$

Wenn man wählt

- $A' = \{(i, n, a) \mid n \in \mathbb{N}, 0 \leq i \leq n, a \in \mathbb{Z}\}$
- $m(i, n, a) = \max(n - i, 0)$

kann man die Terminierung von f auf A' beweisen.

Formen der Rekursion

Eine rekursive Definition $f(x) = E[f, x]$ heißt:

- **linear**, wenn in $E[f, x]$ die Funktion f höchstens einmal aufgerufen wird. Z.B.: wenn in jedem Zweig höchstens ein Aufruf steht.
- **endständig**, wenn $E[f, x]$ eine Fallunterscheidung ist und jeder Zweig, in dem f aufgerufen wird, von der Form $f(G)$ ist, wobei G keine weiteren Aufrufe von f enthält
- **mehrfach rekursiv**, wenn $E[f, x]$ möglicherweise mehrere Aufrufe von f (im selben Zweig) enthält
- **verschachtelt rekursiv**, wenn die Argumente y mit denen f in $E[f, x]$ aufgerufen wird, selbst weitere Aufrufe von f enthalten
- **wechselseitig rekursiv**, wenn im Rumpf eine andere (rekursive) Funktion aufgerufen wird, welche ihrerseits f in ihrem Rumpf verwendet

Lineare Rekursion

Bei **linearer Rekursion** findet im Rumpf höchstens ein rekursiver Aufruf statt.

Beispiele:

- Summe, Fakultät
- `collatz x`

```
  | x <=1      = 0
  | even x     = 1 + collatz (x `div` 2)
  | otherwise = 1 + collatz (3 * x + 1)
```

  Auch linear, da es nur einen Aufruf *pro Zweig* gibt.

Gegenbeispiele:

- Fibonacci hat zwei Aufrufe im Rumpf
- McCarthy's 91-Funktion

```
mc91 n
  | n > 100   = n-10
  | otherwise = mc91(mc91(n+11))
```

John McCarty, 1927-2011, amer. Informatiker

Endständige Rekursion

Lineare Rekursion + Zweige der Fallunterscheidung sind rekursive Aufrufe, deren Ergebnisse nicht weiterverarbeitet werden.

Beispiele

- ```
  foosum i n a
    | i == n    = a
    | otherwise = foosum (i+1) n (a+i)
  ```
- ```
  collatz' acc x
    | x <=1     = acc
    | even x    = collatz' (acc+1) (x `div` 2)
    | otherwise = collatz' (acc+1) (3 * x + 1)
  ```

Gegenbeispiele

- Alle nicht-linearen Rekursionen
- `summe`, `fakultät`, `collatz`, da das Ergebnis der Aufrufe dort noch weiterverarbeitet wird (durch Addition / Multiplikation von `n`)

Endrekursiv dank Akkumulator

Endrekursion kann sehr effizient abgearbeitet werden.
Grund: Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

$$\texttt{collatz}'\ 0\ 11 \rightsquigarrow \texttt{collatz}'\ 1\ 34 \rightsquigarrow \texttt{collatz}'\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz}'\ 3\ 52 \rightsquigarrow \texttt{collatz}'\ 4\ 26 \rightsquigarrow \texttt{collatz}'\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument, den **Akkumulator**, aufsammelt.

Beispiel: `foosum` ist eine endständige Version von `summe`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,0) \rightsquigarrow \texttt{foosum}(1,5,0) \rightsquigarrow \texttt{foosum}(2,5,1) \rightsquigarrow$$
$$\texttt{foosum}(3,5,3) \rightsquigarrow \texttt{foosum}(4,5,6) \rightsquigarrow \texttt{foosum}(5,5,10) = 10$$

Endrekursiv dank Akkumulator

Endrekursion kann sehr effizient abgearbeitet werden.
Grund: Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

$$\texttt{collatz}'\ 0\ 11 \rightsquigarrow \texttt{collatz}'\ 1\ 34 \rightsquigarrow \texttt{collatz}'\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz}'\ 3\ 52 \rightsquigarrow \texttt{collatz}'\ 4\ 26 \rightsquigarrow \texttt{collatz}'\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument, den **Akkumulator**, aufsammelt.

Beispiel: `foosum` ist eine endständige Version von `summe`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,0) \rightsquigarrow \texttt{foosum}(1,5,0) \rightsquigarrow \texttt{foosum}(2,5,1) \rightsquigarrow$$
$$\texttt{foosum}(3,5,3) \rightsquigarrow \texttt{foosum}(4,5,6) \rightsquigarrow \texttt{foosum}(5,5,10) = 10$$

Endrekursiv dank Akkumulator

Endrekursion kann sehr effizient abgearbeitet werden.
Grund: Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

$$\texttt{collatz}'\ 0\ 11 \rightsquigarrow \texttt{collatz}'\ 1\ 34 \rightsquigarrow \texttt{collatz}'\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz}'\ 3\ 52 \rightsquigarrow \texttt{collatz}'\ 4\ 26 \rightsquigarrow \texttt{collatz}'\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument, den **Akkumulator**, aufsammelt.

Beispiel: `foosum` ist eine endständige Version von `summe`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,0) \rightsquigarrow \texttt{foosum}(1,5,0) \rightsquigarrow \texttt{foosum}(2,5,1) \rightsquigarrow$$
$$\texttt{foosum}(3,5,3) \rightsquigarrow \texttt{foosum}(4,5,6) \rightsquigarrow \texttt{foosum}(5,5,10) = 10$$

Endständige Rekursion und Iteration

Endrekursion entspricht im wesentlichen einer While-Schleife:

```
foosum i n a | i == n    = a
             | otherwise = foosum (i+1) n (a+i)
```

kann man ohne Rekursion in einer imperativen Sprache schreiben als

```
int foosum(int i, int n, int a) {
    while (!(i=n)) {
      a = a+i;
      i = i+1;
    }
  return a;
}
```

Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' n 1
  where
    potenz2' :: Int -> Double -> Double
    potenz2' 0 acc = acc
    potenz2' m acc = potenz2' (m-1) (x*acc)
```

Die Funktion `potenz2' n acc` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, braucht die Variable `x` nicht mit übergeben zy werden.

Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' n 1
  where
    potenz2' :: Int -> Double -> Double
    potenz2' 0 acc = acc
    potenz2' m acc = potenz2' (m-1) (x*acc)
```

Die Funktion `potenz2' n acc` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, braucht die Variable `x` nicht mit übergeben zu werden.

Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' n 1
  where
    potenz2' :: Int -> Double -> Double
    potenz2' 0 acc = acc
    potenz2' m acc = potenz2' (m-1) (x*acc)
```

Die Funktion `potenz2' n acc` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, braucht die Variable `x` nicht mit übergeben zy werden.

Schnelle Potenzierung

Tatsächlich geht es aber noch besser:

```
potenz3 :: Double -> Int -> Double
potenz3 _ 0 = 1
potenz3 x n
  | even n    =     x_to_halfn_sq
  | otherwise = x * x_to_halfn_sq
  where
    x_to_halfn_sq = square (potenz3 x (n `div` 2))
    square y      = y * y
```

Begründung: $x^{2k} = (x^k)^2$ und $x^{2k+1} = (x^k)^2 \cdot x$.

Aufgabe: Schafft es jemand, eine endrekursive Version davon anzugeben, ohne dabei in die Standardbibliothek zu spicken?

Rekursion mit Listen

Rekursion ist nicht auf Zahlen beschränkt:

```
length :: [a] -> Int
length []     = 0
length (_h:t) = 1 + length t
```

```
length [1,2,3]
⇝ 1 + (length [2,3])
⇝ 1 + 1 + (length [3])
⇝ 1 + 1 + 1 + (length [])
⇝ 1 + 1 + 1 + 0
⇝ 3
```

Rekursion mit Listen

Auch das Ergebnis muss keine Zahl sein:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse "fly"
⇝ reverse "ly" ++ "f"
⇝ reverse "y" ++ "l" ++ "f"
⇝ reverse "" ++ "y" ++ "l" ++ "f"
⇝ "" ++ "y" ++ "l" ++ "f"
⇝ "ylf"
```

Rekursion eignet sich hervorragend dazu, Listen zu bearbeiten!

Beispiel: Sortieren durch Einfügen

```
insert :: Int -> [Int] -> [Int]
insert x    [] = [x]
insert x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert x ys
```

```
insert 2 [1,3,4,5]
⇝ 1 : insert 2 [3,4,5]
⇝ 1 : 2 : 3 : [4,5]
= [1,2,3,4,5]
```

```
sort :: [Int] -> [Int]
sort    []  = []
sort (x:xs) = insert x (sort xs)
```

Beispiel: Sortieren durch Einfügen

```
insert :: Int -> [Int] -> [Int]
insert x    [] = [x]
insert x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert x ys
```

```
insert 2 [1,3,4,5]
⇝ 1 : insert 2 [3,4,5]
⇝ 1 : 2 : 3 : [4,5]
= [1,2,3,4,5]
```

```
sort :: [Int] -> [Int]
sort    []  = []
sort (x:xs) = insert x (sort xs)
```

Rekursion mit mehreren Listen

Wir können auch mehrere Listen auf einmal berarbeiten:

```
zip :: [a] -> [b] -> [(a,b)]
zip    []     _    = []
zip    _      []   = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip "abc" [1,2,3,4]
⇝ ('a',1): zip "bc" [2,3,4]
⇝ ('a',1):('b',2): zip "c" [3,4]
⇝ ('a',1):('b',2):('c',3): zip "" [4]
⇝ ('a',1):('b',2):('c',3):[]
=     [('a',1),('b',2),('c',3)]
```

Richtlinien zur Rekursion

Viele Studenten haben anfangs Probleme damit, bewusst rekursive Funktionen zu schreiben.
G.Hutton schreibt dazu: Es ist wie Fahrrad fahren, wenn man es nicht kann sieht es unmöglich aus; und wenn man es kann, ist wirklich einfach.

1. Explizit **Typ der Funktion** hinschreiben hilft!
2. **Fallunterscheidungen** auflisten. Bei Listen ist fast immer klar, dass zwischen leerer und nicht-leerer Liste unterschieden werden muss; bei natürlichen Zahlen erfordert oft die Null eine Sonderbehandlung
3. **Einfach Fälle** zuerst programmieren, die rekursiven Fälle kommen dann oft von alleine.
4. In **rekursiven Fällen** überlegen, was zur Verfügung steht!
5. Am Ende dann **verallgemeinern und vereinfachen**.

Beispiel: drop

`drop` ist eine Funktion der Standardbibliothek, welche eine Zahl *n* und eine Liste *xs* als Argumente nimmt, und als Ergebnis eine Liste zurückliefert, bei der die ersten *n* Elemente fehlen.

Online-Demonstration ergab den Code:

```
myDrop :: Int -> [a] -> [a]
myDrop 0 []     = []
myDrop 0 (h:t)  = h:t
myDrop n []     = []
myDrop n (h:t)  = myDrop (n-1) t
```

. . . welchen wir anschliessend leicht vereinfachen konnten zu:

```
myDrop 0 xs     = xs
myDrop _ []     = []
myDrop n (_:t)  = myDrop (n-1) t
```

Zusammenfassung

- Rekursion als Mittel zur Definition von Funktionen
- Rekursion im Substitutionsmodell
- Rekursive Funktionen sind oft nicht überall definiert. Mit einer Abstiegsfunktion kann man die Definiertheit an bestimmten Stellen nachweisen.
- Verschiedene Arten der rekursiven Definition, insbesondere endständige Rekursion.
- Rekursion als Mittel zur Problemlösung (Bsp. Hanoi)
- Rekursion für den Umgang mit Listen (Bsp. Sortieren)