Inhalt Kapitel 2: Rekursion

Rekursion

Man kann eine Funktion $f : A \to B$ durch einen Term definieren, der selbst Aufrufe 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.
Wir rechnen aus:

$$\text{fakultät}(3) = 3 \cdot \text{fakultät}(2) = 3 \cdot 2 \cdot \text{fakultät}(1) = 3 \cdot 2 \cdot 1 \cdot \text{fakultät}(0) = 3 \cdot 2 \cdot 1 \cdot 1 = 6$$

Rekursion in SML

```
- fun summe(n) = if n = 0 then 0 else n + summe(n-1);
val summe = fn : int -> int
- summe(6);
val it = 21 : int
- fun fakultaet(n) = if n = 0 then 1 else n * fakultaet(n-1);
val fakultaet = fn : int -> int
- fakultaet(10);
val it = 3628800 : int
```

Alternativ auch mit "val":

```
- val rec fakultaet2 = fn n => if n = 0 then 1 else
            n * fakultaet2(n-1);
val fakultaet = fn : int -> int
```

Das `rec` nicht vergessen!

Fibonacci-Zahlen

```
- fun fib(n) = if n <= 1 then 1 else fib(n-1)+fib(n-2);
val fib = fn : int -> int
- fib(7);
val it = 21 : int
- fib(8);
val it = 34 : int
```

- `fib`(n) liefert die n-te Fibonacci-Zahl F_n: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- `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.

"$3n + 1$" rekursiv und in SML

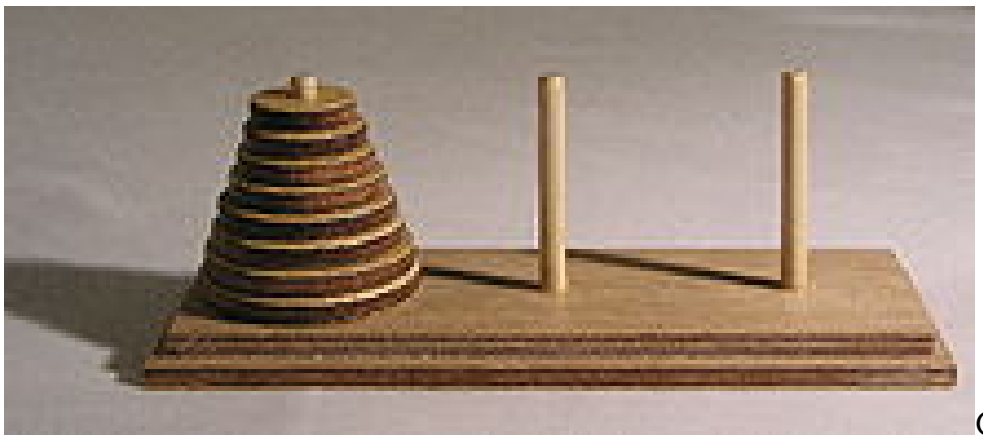
```
- fun g(n) = if n mod 2 = 0 then n div 2 else 3*n+1;
val g = fn : int -> int
- fun f(n) = if n = 1 then 0 else 1 + f(g(n));
val f = fn : int -> int
- f(2);
val it = 1 : int
- f(3);
val it = 7 : int
- f(4);
val it = 2 : int
- f(5);
val it = 5 : int
- f(27);
val it = 111 : int
```

Diese Funktion heißt nach ihrem Erstbeschreiber Collatz-Funktion (Lothar Collatz 1910-90, dt. Mathematiker).

Dies berechnet wieder die Funktion von Folie 9.

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 SML

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

```
fun hanoi(n,i,j) = if n =1 then [(i,j)] else
          hanoi(n-1,i,6-i-j) @ [(i,j)] @ hanoi(n-1,6-i-j,j);
val hanoi = fn : int * int * int -> (int * int) list
- hanoi(3,1,2);
val it = [(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)] :
                                        (int * int) list
- hanoi(4,1,2);
val it =
  [(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3),(1,2),...
```

- `[(i,j)]` ist die Einer-Liste mit einzigem Eintrag `(i,j)`.
- `@` (infixnotiert) bezeichnet die Verkettung von Listen.

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, als auch die definierte Funktion f selbst vorkommen dürfen.
Allerdings darf im Rumpf $E[f, a]$ auf f nur in Form von Aufrufen zugegriffen werden und (bis auf weiteres) dürfen auch nur eine endliche Zahl von Aufrufen getätigt werden.
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 `fakultaet` liefern bei negativen Eingaben kein Ergebnis:

  ```
  - summe(~2);
    C-c C-c
  Interrupt
  -
  ```

  Grund: $\texttt{summe}(-2) = -2 + \texttt{summe}(-3) = -2 - 3 + \texttt{summe}(-4) = -2 - 3 - 4 + \texttt{summe}(-5) = \ldots$.
- Gleiches gilt für

  ```
  - fun f(n) = f(n);
  ```
- Die "$3n + 1$"-Funktion (Folie 5); man weiß nicht, ob sie für alle n definiert ist.

Abstiegsfunktion

Um festzustellen, ob eine rekursiv definierte Funktion für ein Argument definiert ist, kann man eine Abstiegsfunktion verwenden.

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.

Dann muss noch nicht unbedingt gelten $A' \subseteq D(f)$.
(Gegenbeispiel: $E[f, x] = f(x)$)

Abstiegsfunktion

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

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

Dann ist $A' \subseteq D(f)$.
Man bezeichnet so ein m als Abstiegsfunktion.

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

Komplizierte Abstiegsfunktion

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

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

$$f(3,6,10) = f(4,6,13) = f(5,6,17) = f(6,6,22) = 22$$

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.
- 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 enthält
- verschachtelt rekursiv, wenn die Argumente y mit denen f in $E[f, x]$ aufgerufen werden, selbst weitere Aufrufe von f enthalten.

Lineare Rekursion

Im Rumpf findet höchstens ein rekursiver Aufruf statt:

Beispiele

- Summe, Fakultät
- ```
  fun f(x) = if x <= 1 then 1 else
             if x mod 2 = 0 then 1+f(x div 2) else 1+f(3*x+1)
  ```
  Nur ein Aufruf pro Zweig. Gilt auch.

Gegenbeispiele

- Fibonacci
- McCarthy's 91-Funktion

  ```
  fun M(n) = if n>100 then n-10 else M(M(n+11))
  ```
  ```

Endständige Rekursion

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

Beispiele

- `fun f1(i,n,a) = if i=n then a else f1(i+1,n,a+i)`
- `fun f2(n,a) = if n<=1 then a else if n mod 2 = 0 then f2(n d`

Gegenbeispiele

- Alle nicht-linearen Rekursionen
- `summe`, `fakultaet` (Ergebnis der Aufrufe wird weiterverarbeitet durch Addition / Multiplikation von `n`)

Endständige Versionen von Summe und Fakultät

Endständige Rekursionen können sehr effizient abgearbeitet werden.

Grund: Im Substitutionsmodell entstehen keine großen Zwischenergebnisse.

Beispiel

$$\texttt{f2}(11,0) = \texttt{f2}(34,1) = \texttt{f2}(17,2) = \texttt{f2}(52,3) = \\ \texttt{f2}(26,4) = \texttt{f2}(13,5) = \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, z.B. ist `f1` eine endständige Version von `summe`

```
fun f1(i,n,a) = if i=n then a else f1(i+1,n,a+i)
```

$$\texttt{f1}(0,5,0) = \texttt{f1}(1,5,0) = \texttt{f1}(2,5,1) = \\ \texttt{f1}(3,5,2) = \texttt{f1}(4,5,5) = \texttt{f1}(5,5,10) = 10$$

Endständige Rekursion und Iteration

Eine endständige Rekursion entspricht im wesentlichen einer While-Schleife:

```
fun f1(i,n,a) = if i=n then a else f1(i+1,n,a+i)
```

entspricht:

```
int f1(int i, int n, int a) {
while (!(i=n)) {
   a = a+i;
   i = i+1;
}
return a;
}
```

Einbettung

Häufig muss für eine rekursive Lösung die Problemstellung leicht verallgemeinert werden.

- Endständig rekursive Version der Summe: `f1(i,n,a)` berechnet $a + \sum_{j=i}^{n-1} j$.
- Versuch eines rekursiven Primzahltest:

```
fun istPrim(n) = if n<=1 then false
        else if n = 2 then true else ...???
```

Eine Zahl n ist prim, wenn sie keine Teiler im Bereich $1 \ldots n-1$ hat. Wir verallgemeinern das auf den Bereich $i \ldots n-1$

keineTeiler

```
fun keineTeiler(i,n) = if i=n then true else
     n mod i > 0 andalso keineTeiler(i+1,n);
fun istPrim(n) = if n<=1 then false
         else if n = 2 then true else keineTeiler(2,n)
fun naechstePrimzahl(n) = if istPrim(n) then n else naechstePrim
```

Z.B.: `naechstePrimzahl`(123456789)=123456791.
Es gibt wesentlich effizientere Methoden um auf Prim-heit zu testen.

Endständige Rekursion durch Einbettung

Rekursives Programm zur Potenzierung:

```
fun potenz(x,n) = if n=0 then 1.0 else x*potenz(x,n-1)
```

Endrekursive Version:

```
fun potenz2(x,n,a) = if n=0 then a else
                     potenz2(x,n-1,x*a)
```

`potenz2`(x, n, a) berechnet $x^n \cdot a$. Man ruft dann mit $a = 1$ auf.

Schnelle Potenzierung

```
fun square(x:real) = x*x
fun potenz(x,n) = if n=0 then 1.0 else
            if n mod 2 = 0 then square(potenz(x,n div 2))
                           else square(potenz(x,n div 2))*x
```

Beweis: $x^{2k} = (x^k)^2$ und $x^{2k+1} = (x^k)^2 \cdot x$.
Schafft es jemand, eine endrekursive Version davon anzugeben?