Nichtprimitive Rekursion und die Fibonacci-Zahlen

Primitive und nichtprimitive Rekursion

Die rekursive Berechnung von $n!$ ist ein typisches Beispiel für eine primitive Rekursion. Man nennt eine rekursive Definition primitiv, wenn die Berechnung von $f(n)$ nur auf die Werte von n und von $f(n-1)$ zurückgeführt wird, also wenn sich die Funktion f in der folgenden Weise darstellen lässt

$$f(n) = g(n, f(n-1)).$$

Eine wichtige Konsequenz daraus ist, das $f(n)$ in linearer Zeit (gemessenen in Abhängigkeit von n) berechnet werden kann, sofern man die Funktion g in konstanter Zeit auswerten kann. Weitere primitiv rekursive Funktionen lassen sich finden, wenn man an Programme denkt, die aus einer einfachen Zählschleife bestehen, wie z.B. die Berechnung der n-ten Potenz einer Zahl a oder die Berechnung der Summe $\sum_{k=0}^{n} k^2$. Für das letzte Beispiel hat man die Verankerung $f(0) = 0$ und die Rekursion $f(n) = n^2 + f(n-1)$, d.h. die Darstellung $f(n) = g(n, f(n-1))$ ergibt sich, wenn man $g(x, y) = x^2 + y$ verwendet.

Der Euklidische Algorithmus ist nicht primitiv rekursiv, denn man geht nicht auf den unmittelbaren Vorgänger zurück, sondern kann auch größere Sprünge machen. Dennoch erreicht man auch hier eine Laufzeit, die (gemessenen in Abhängigkeit von $max(n, m)$) höchstens linear ist. Hauptgrund dafür ist, dass man sich mit jedem Schritt weiter in Richtung der Verankerung bewegt **und** jeweils **nur ein** neuer Aufruf von ggT erfolgt. Eine interessante Erweiterung des Euklidischen Algorithmus (wichtig insbesondere für kryptographische Anwendungen) besteht in der Berechnung einer Darstellung des größten gemeinsamen Teilers als ganzzahlige Linearkombination der Eingabewerte.

Satz: Sind $a, b \in \mathbb{Z}^+$, dann kann man $\mathrm{ggT}(a, b)$ als Linearkombination von a und b mit ganzzahligen Koeffizienten darstellen, d.h. es existieren $s, t \in \mathbb{Z}$, so daß

$$\mathrm{ggT}(a, b) \;=\; sa \;+\; tb.$$

Beweis: Die Idee ist sehr einfach – man muss den Euklidischen Algorithmus nur umkehren. Zur Illustration zeigen wir das zuerst am einem Beispiel zur Berechnung von $\mathrm{ggT}(252, 198)$:

$$\begin{aligned} 252 &= 1 \cdot 198 + 54 \\ 198 &= 3 \cdot 54 + 36 \\ 54 &= 1 \cdot 36 + 18 \\ 36 &= 2 \cdot 18 + 0 \qquad \rightsquigarrow \qquad \mathrm{ggT}(252, 198) = 18 \end{aligned}$$

Jetzt werden alle Gleichungen nach dem Rest umgestellt:

$$\begin{aligned} 18 = &\quad 54 \quad -1 \cdot 36 \\ 36 = &\quad 198 \quad -3 \cdot 54 \\ 54 = &\quad 252 \quad -1 \cdot 198 \end{aligned}$$

Durch schrittweise Substitution erhalten wir:

$$\begin{aligned} 18 = &\quad 54 - 1 \cdot 36 = 54 - 1 \cdot (198 - 3 \cdot 54) = 4 \cdot 54 - 1 \cdot 198 = \\ &= 4 \cdot (252 - 1 \cdot 198) - 1 \cdot 198 = 4 \cdot 252 - 5 \cdot 198 \end{aligned}$$

Der formale Beweis erfolgt durch vollständige Induktion nach der Anzahl n der Durchläufe der while–Schleife, die der Euklidische Algorithmus auf der Eingabe (a, b) ausführt.

Induktionsanfang: Im Fall $n = 1$ ist der Rest $r = a \ \boldsymbol{mod}\ b$ gleich Null und $\mathrm{ggT}(a, b) = b$. Wie man leicht sieht, leistet die Linearkombination $b = 0 \cdot a + 1 \cdot b$ das Gewünschte.

Induktionsschritt: Wenn die while–Schleife mehr als einmal durchlaufen wird, ist der Rest $r = a \ \boldsymbol{mod}\ b$ größer als Null und $\mathrm{ggT}(a, b) = \mathrm{ggT}(b, r)$. Da der Euklidische Algorithmus die while–Schleife für die Eingabe (b, r) einmal weniger durchläuft als für (a, b) kann man die Induktionsvoraussetzung auf (b, r) anwenden:

$$\exists s, t \in \mathbb{Z}\ \mathrm{ggT}(b, r) \ = \ sb \ + \ tr$$

Wir stellen die Gleichung $a = qb + r$ nach r um, setzen das Ergebnis in obige Linearkombination ein

$$\mathrm{ggT}(a, b) = \mathrm{ggT}(b, r) \ = \ sb \ + \ tr \ = \ sb \ + \ t(a - qb) = ta \ + \ (s - tq)b$$

und haben damit die Induktionsbehauptung nachgewiesen. □

Man nennt die oben beschriebene Methode auch den erweitereten Euklidischen Algorithmus. Die Berechnung der Linearkombination in rekursiver Form ist nicht ganz offensichtlich, aber wir werden dieses Problem später als Übungsaufgabe lösen. Dabei wird sich zeigen, dass die Laufzeit nur um einen konstanten Faktor über der Laufzeit des normalen Euklidischen Algorithmus liegt.

Die Situation, dass rekursive Algorithmen gute Laufzeiten erzielen, kann sich ändern, wenn ein Rekursionsschritt zu mehren Funktionsaufrufen führt, z.B. wenn der Rekursionsschritt die allgemeine Form $f(n) = g(f(n-1), f(n-2))$ hat.

Das bekannteste Beispiel einer solchen Rekursion ist die Folge der Fibonacci-Zahlen, die mit $0, 1, 1, 2, 3, 5, 8, \ldots$ beginnt und durch die Rekursion $f(n) = f(n-1) + f(n-2)$ definiert ist. Diese Folge wurde vom Mathematiker Leonardo von Pisa (bekannter unter dem Namen Fibonacci) als nicht ganz ernsthafte Beschreibung der Entwicklung einer Kaninchenpopulation eingführt. Man trifft diese Zahlen in der Natur wieder, z.B. bei der Abzählung von rechts- und linksdrehenden Spiralen in Sonnenblumenblüten.

Es ist also nicht schwer, ein Haskell-Programm zur Berechnung dieser Zahlen zu schreiben.

```
fib ::  Int -> Int
fib n
    | n==0 = 0 -- man beginnt mit fib 0, das ist eine Festlegung
    | n==1 = 1 -- diese Rekursion braucht zwei Verankerungen
    | otherwise = fib (n-1) + fib (n-2)
```

Exponentielles Wachstum der Fibonacci-Zahlen und Laufzeit

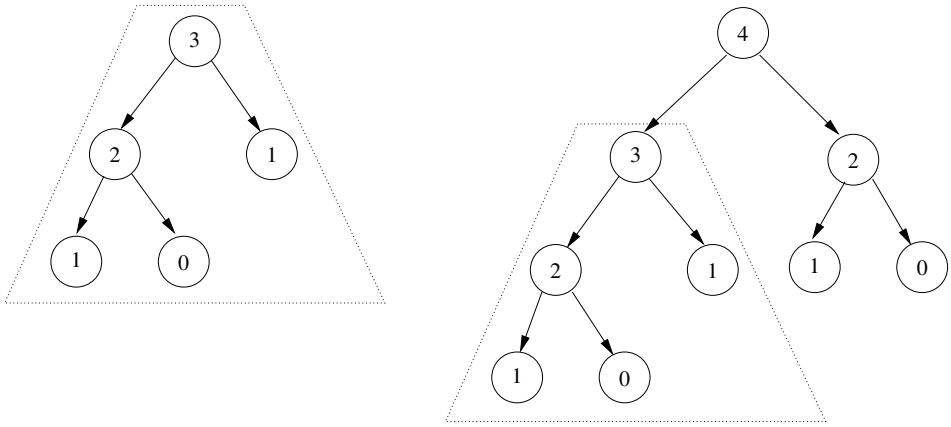
Sieht man sich ein etwas längeres Anfangsstück der Fibonacci-Folge an, so fällt auf, dass sie schnell wächst: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$. Es ist offensichtlich, dass die Folge monoton wachsend ist, und man überzeugt sich leicht, dass sie exponentiell wächst, denn durch die Monotonität ergibt sich:

1) $f(n) = f(n-1) + f(n-2) \leq 2f(n-1)$ und folglich $f(n) \leq 2^n$,
2) $f(n) = f(n-1) + f(n-2) \geq 2f(n-2)$ und folglich $f(2n) \geq 2^{n-1}$ (beginnend mit $f(2) = 2^0$), d.h. $f(n) \geq (\sqrt{2})^{n-1}$.

Die Basis des exponentiellen Wachstums liegt also zwischen $\sqrt{2}$ und 2, eine genauere Bestimmung geben wir am Ende der Vorlesung.

Beim Aufruf des Programms kann man beobachten, dass auch die Laufzeit stark wächst, sie liegt schon bei `fib 37` je nach Rechner bei mehreren Minuten. Zum Verständnis dessen machen wir eine Laufzeitanalyse, d.h. wir zählen ab, wie oft die Funktion `fib` aufgerufen wird, wenn wir `fib n` berechnen, und bezeichnen diese Anzahl mit $t(n)$.

Offensichtlich ist $t(0) = t(1) = 1$, denn wir bekommen mit den Aufruf auch gleich das Ergebnis. Dagegen ist $t(2) = 3$, denn der Aufruf `fib 2` zieht die Aufrufe `fib 1` und `fib 0` nach sich. Für `fib 3` und `fib 4` kann man die Aufrufstruktur durch die folgenden zwei Bäume nachzeichnen und erhält $t(3) = 5,\ t(4) = 9$.



Diese Bäume zeigen auch, wie man $t(n)$ rekursiv beschreiben kann, denn der Aufruf

`fib n` führt zu den Aufrufen `fib (n-1)` und `fib (n-2)`, d.h.

$$t(n) = 1 + t(n-1) + t(n-2).$$

Da diese Rekursion sehr ähnlich zur Fibonacci-Rekursion ist, ist zu vermuten, dass sich auch $t(n)$ und $f(n)$ stark ähneln. Zum Beweis verwendet man vollständige Induktion in der verallgemeinerten Variante, d.h. beim Induktionsschritt ist es erlaubt, nicht nur auf den unmittelbaren Vorgänger, sondern auf beliebige kleinere Zahlen zurückzugreifen. Für den Induktionsanfang wird die folgende Tabelle sehr nützlich sein.

n	0	1	2	3	4	5	6
$f(n)$	0	1	1	2	3	5	8
$t(n)$	1	1	3	5	9	15	25

Behauptung: $t(n) \geq f(n+1)$ und $t(n) \leq f(n+3) - 1$ für alle natürlichen Zahlen n.
Den Induktionsanfang für $n = 0$ und $n = 1$ kann man bei beiden Ungleichungen aus der Tabelle ablesen.
Sei nun $n \geq 1$ und die Behauptung für n und alle kleineren Zahlen bewiesen. Wir zeigen zuerst, dass die linke Ungleichung auch für $n + 1$ gilt:

$$t(n+1) = 1 + t(n) + t(n-1) \geq 1 + f(n+1) + f(n) \geq f(n+2)$$

Dabei wurde die Rekursion für $t(n+1)$, die Induktionsvoraussetzung und die Rekursion für $f(n+2)$ verwendet. Für die rechte Ungleichung funktioniert das ähnlich:

$$t(n+1) = 1 + t(n) + t(n-1) \leq 1 + f(n+3) - 1 + f(n+2) - 1 \leq f(n+4) - 1$$

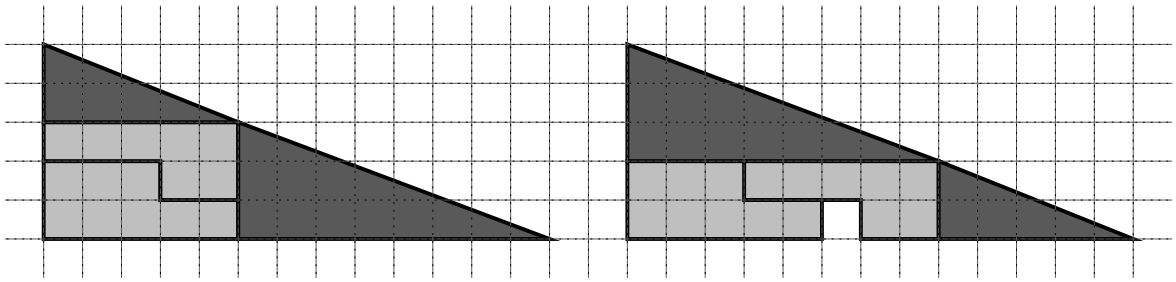
Genauere Laufzeitabschätzungen

Mit etwas anspruchsvolleren mathematischen Methoden kann man die folgende geschlossene Formel für die Fibonacci-Zahlen herleiten:

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

Aus dieser Formel lässt sich sehr gut das exponentielle Wachstum der Fibonacci-Zahlen und ihr Konvergenzverhalten ablesen. Der zweite Summand kann für große n vernachlässigt werden, so dass der Quotient aus zwei aufeinanderfolgenden Fibonaccizahlen gegen $\frac{1+\sqrt{5}}{2}$ geht. Diese Zahl beschreibt bekanntlich den goldenen Schnitt. Man kann also sagen, dass sich die Laufzeit der Fibonacci-Rekursion (gemessen in Funktionsaufrufen) proportional zum Exponentialausdruck $(\frac{1+\sqrt{5}}{2})^n$ verhält.

Eine geometrische Demonstration dafür, dass diese Konvergenz schon für relativ kleine Zahlen einsetzt, findet man in der folgenden Abbildung:



Aus vier Puzzleteilen wird jeweils ein 5×13 Dreieck zusammengesetzt, aber im rechten Dreieck bleibt ein Feld frei. Wie kann man das erklären?
Beim genaueren Hinsehen oder besser beim Nachrechnen entdeckt man, dass die Hypotenusen der beiden kleinen Dreiecke verschiedene Anstiege haben und somit keine Gerade bilden: $\frac{2}{5} = 0,4$ und $\frac{3}{8} = 0,375$. Unter Berücksichtigung von $2 = f_3,\ 3 = f_4,\ 5 = f_5$ und $8 = f_6$ erhält man eine geometrische Idee davon, dass sich die Quotienten f_3/f_5 und f_4/f_6 nur wenig unterscheiden.

Verbesserung der Laufzeit

Wie man sich leicht überzeugen kann, liegt der Hauptgrund für die schlechte Laufzeit nicht primär an der Programmiersprache Haskell, sondern an der nichtprimitiven Form der Rekursion, die für jeden Rekursionsaufruf zwei weitere Funktionsaufrufe nach sich zieht.
Bei der Verwendung einer imperativen Programmiersprache gibt es einen einfachen Ausweg, nämlich den Verzicht auf Rekursion und die Zwischenspeicherung aller Fibonacci-Zahlen bis zu der zu berechnenden Zahl.
Ein solches Zwischenspeichern ist bei funktionalen Programmiersprachen nicht möglich, aber man kann es über einen kleinen Trick erreichen. Man verwendet für die Ausgabe einen Datentyp, der mehr als eine Zahl halten kann. Wir werden in den nächsten Vorlesungen zwei neue Datentypen kennenlernen, die dafür geeignet sind: Tupel und Listen.