Funktionale Programmierung
Formen der Rekursion

Formen der Rekursion

1. Was ist nochmals Rekursion(?)
   - Informell
   - Rekursion als Gleichungssystem
   - Primitive Rekursion
   - Wertverlaufsrekursion
   - Allgemeine Rekursion

Was ist nochmals Rekursion(?)

Informell

Rekursive Definitionen zeichnen sich durch die Bezugnahme auf das zu definierende Objekt *in einer einfacheren Form* aus.

$$2^0 = 1$$

$$\underbrace{2^{n+1}}_{\text{definiere}} = 2 \cdot \underbrace{2^n}_{\text{Selbstbezug}}$$

Was ist nochmals Rekursion(?)

Informell

Eine rekursive Definition einer Funktion “ist” ein Algorithmus, der aus bereits vorhandenen Funktionswerten weitere Funktionswerte (der dadurch definierten Funktion) generiert.

$2^0 = 1$	Der Funktionswert an der Stelle 0 ist bekannt.
$2^{n+1} = 2 \cdot 2^n$	Durch das Verdoppeln eines bekannten Funktionswertes erhält man den “nächsten” Funktionswert.

Was ist nochmals Rekursion(?)

Auswertung

Die Auswertung einer rekursiv definierten Funktion erfolgt “rückwärts”:

$$
\begin{aligned}
2^3 &= 2^{2+1} \\
&= 2 \cdot 2^2 \\
&= 2 \cdot 2^{1+1} \\
&= 2 \cdot (2 \cdot 2^1) \\
&= 2 \cdot (2 \cdot 2^{0+1}) \\
&= 2 \cdot (2 \cdot (2 \cdot 2^0)) \\
&= 2 \cdot (2 \cdot (2 \cdot 1)) \quad \leftarrow \text{ bekannter Wert eingesetzt} \\
&= 2 \cdot (2 \cdot 2) \\
&= 2 \cdot 4 \\
&= 8
\end{aligned}
$$

Was ist nochmals Rekursion(?)

Rekursion als Gleichungssystem

Wesentliche Zutaten einer rekursiven Definition[1]:

- Bekannte Funktionswerte
- Funktion G zum Erweitern der Menge der bekannten Werte

$$2^0 = 1$$
$$2^{n+1} = 2 \cdot 2^n$$

bekannter Wert

$$\underbrace{G(x) = 2 \cdot x}_{\text{“body” des rek. Calls}}$$

[1] Wie wir sie bis jetzt kennen.

Dies entspricht im Allgemeinen einem Gleichungssystem von Funktionen:

$$f(0) = c$$
$$f(n+1) = G(f(n))$$

wobei

- c eine Konstante
- G eine Funktion (ein Algorithmus)

Was ist nochmals Rekursion(?)

Rekursion als Gleichungssystem

- Kann man etwas “als Lösung” eines Gleichungssystems “definieren”?

Was ist nochmals Rekursion(?)

Rekursion als Gleichungssystem

- Kann man etwas “als Lösung” eines Gleichungssystems “definieren”?
- Ja, wenn das Gleichungssystem eine eindeutige Lösung besitzt.

- Kann man etwas “als Lösung” eines Gleichungssystems “definieren”?
- Ja, wenn das Gleichungssystem eine eindeutige Lösung besitzt.
- Gleichungssysteme der Form

$$f(0) = c$$
$$f(n+1) = G(f(n))$$

haben immer eine eindeutige Lösung (Beweis Eindeutigkeit per Induktion, Existenz via “Fixpunkte”).

Rekursionsschemata

Vorher haben wir eine besonders einfache Art kennengelernt, wie man eine Rekursionsgleichung aufschreiben kann. Verschiedene, teilweise viel kompliziertere, “Arten” solcher Rekursionsgleichungen, werden als *Rekursionsschemas* bezeichnet. Wir werden nun ein paar sochler Schemata kennenlernen.

Mit wenigen zusätzlichen Parametern ergibt sich das sogenannte Schema der "primitiven Rekursion":

$$f(0, \vec{x}) = c(\vec{x})$$
$$f(n+1, \vec{x}) = G(f(n, \vec{x}), n, \vec{x})$$

wobei

- c eine Funktion (in den Variablen $\vec{x}$)
- G eine Funktion (ein Algorithmus)

Die zusätzlichen Parameter erlauben etwas mehr Freiheit, beispielsweise zum Definieren der allgemeinen Exponentialfunktion:

$$x^0 = 1 \qquad c(x) = 1$$
$$x^{n+1} = x \cdot x^n \qquad G(a, x) = x \cdot a$$

Manchmal, zum Beispiel bei der Definition der Fakultätsfunktion, erweist sich auch der Parameter n als nützlich:

$$0! = 1 \qquad c = 1$$
$$n+1! = (n+1) \cdot n! \qquad G(a, n) = (n+1) \cdot a$$

Manchmal wird in rekursiven Definitionen nicht bloss auf einen, sondern auf mehrere "Vorgänger" Bezug genommen:

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

Dies stellt einen Spezialfall der "Wertverlaufsrekursion" dar. In der Wertverlaufsrekursion kann nämlich auf den gesamten Wertverlauf (bis zu einem Punkt) der zu definierenden Funktion Bezug genommen werden.

$$f(n) = G(f \upharpoonright n)$$

oder mit Parametern:

$$f(n, \vec{x}) = G(f \upharpoonright n, n, \vec{x})$$

Bemerkung

Durch Codierung von endlichen Sequenzen als Zahlen, lässt sich jede Wertverlaufsrekursion auch als primitive Rekursion realisieren.

- Bis jetzt haben wir uns bei rekursiven Aufrufen stets auf einen (primitive Rekursion) oder mehrere (Wertverlaufsrekursion) Funktionswerte mit "kleineren" Argumenten gestützt (z.B. $2^4 = 2 \cdot 2^3$).
- Wir haben Rekursion entlang der "üblichen Ordnung" $<$ der natürlichen Zahlen angewendet.
- Rekursion kann grundsätzlich entlang jeder Relation angewendet werden. Den Preis, den man dafür bezahlt ist die Möglichkeit, dass die zu definierende Funktion nicht immer "wohldefiniert" ist und insbesondere manchmal keinen Rückgabewert liefert[2].

[2] Ist die Relation nicht wohlfundiert, kann nicht garantiert werden, dass die definierte Funktion total und/oder eindeutig ist oder überhaupt existiert.

Die Funktion $col : \mathbb{N} \to \mathbb{N}$ sei durch folgende Vorschrift gegeben:

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

Die n-te Collatz[3] - Sequenz C_n erhält man dadurch, dass man mit n beginnend so lange die Funktion col anwendet, bis der Wert 1 erreicht wird. Die Folge C_{17} ist demnach durch

$$17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

gegeben. Die Funktion $x \mapsto C_x$ ist eine rekursiv definierbare Funktion, von der nicht bekannt ist, ob zu jedem Input überhaupt ein Output existiert.

[3] `https://de.wikipedia.org/wiki/Collatz-Problem`

Aufgabe

Implementieren Sie die Funktion

```
col ::  Integer -> Integer,
```

die für eine Zahl die nächste Zahl in der (entsprechenden) Collatz Folge berechnet. Implementieren Sie die Funktion $x \mapsto C_x$ mit Rekursion.

Ein (in der Anwendung sehr wichtiger) Spezialfall der allgemeinen Rekursion ist die sogenannte strukturelle Rekursion. Sie dient dazu rekursive Typen zu "dekonstruieren". Ein typisches Beispiel ist

```
1 length' :: [a] -> Integer
2 length' [] = 0
3 length' (_:xs) = 1 + length' xs
```

Mit den verallgemeinerten "Folds" haben wir in den vergangenen Vorlesungen bereits gelernt, wie wir für jeden Summentyp ein passendes Schema für die strukturelle Rekursion zur Verfügung stellen können.

Vereinfacht gesagt, kann Alles was mit den üblichen Mitteln der Programmierung erreicht werden kann auch mittels Rekursion umgesetzt werden. Aufgrund dieser Tatsache sind Formalismen, die vollständig auf Iterationen verzichten, aber Rekursion in einer allgemeinen Form unterstützen, trotzdem Turing - vollständig.