Rekursive Funktionen

Christoph Kreitz & Nuria Brede

Institut für Informatik, Universität Potsdam, 14482 Potsdam

Zusammenfassung *Dieser Artikel gibt einen Überblick über die primitiv- und μ-rekursiven Funktionen. Er ist gedacht als Hintergrundmaterial zur Ergänzung der relativ knappen Abhandlung in der Vorlesung "Einführung in die Theoretische Informatik II".*

1 Rekursive Funktionen

Rekursive Funktionen sind möglicherweise der älteste konkrete Ansatz zur Erklärung von Berechenbarkeit (Dedekind 1888 [3]). Das Konzept entstand lange, bevor man über Maschinenmodelle nachdachte, und benutzt eine rein *mathematische* Vorgehensweise um zu beschreiben, welche Funktionen berechenbar sind.

Das Konzept der rekursiven Funktionen geht davon aus, daß der Begriff der Berechenbarkeit etwas intuitiv Einsichtiges sein ist. Bei einigen elementaren Funktionen würde niemand ernsthaft bezweifeln, daß sie berechenbar sein müssen. Hierzu zählt zum Beispiel die *Nachfolgerfunktion* (oft mit s bezeichnet), die bei Eingabe einer natürlichen Zahl x die nächstgrößere Zahl bestimmt, denn Zählen ist einer der intuitivsten Berechnungsmechanismen. Auch die sogenannten *Projektionsfunktionen*, die aus einer Gruppe von Werten $x_1, .., x_n$ einen spezifischen Wert x_k herausgreifen – wir bezeichen sie mit pr^n_k – sind offensichtlich berechenbar und das gleiche gilt für alle *Konstantenfunktionen*, also Funktionen, die unabhängig von der Eingabe $x_1, .., x_n$ immer denselben Wert k zurückgeben. Wir verwenden hierfür die Bezeichnung c^n_k.

Intuitiv einsichtig ist auch, daß man berechenbare Funktionen auf verschiedene Arten zusammensetzen kann und dabei wieder eine berechenbare Funktion erhalten muß. Dies gilt sicherlich für die einfache *Komposition* von Funktionen, also die Hintereinanderausführen mehrerer Berechnungen, bei der das Ergebnis einer Reihe von Funktionen $g_1, .., g_n$ als Eingabe für eine andere Funktion f dient, was man üblicherweise mit $f \circ (g_1, .., g_n)$ bezeichnet.

Auch die *Iteration* einer Funktion f, also ihre wiederholte Anwendung auf ihr eigenes Ergebnis, ist ist offensichtlich leicht zu berechnen. Im Prinzip entspricht dies einer vielfachen Komposition $f \circ f \circ ... \circ f$. Im Gegensatz dazu ist die Anzahl der Kompositionsschritte jedoch nicht fest vorgegeben, sondern abhängig von der Eingabe. In der Mathematik verwendet man daher zur Beschreibung eine einfache *Rekursion*, da dies präziser ist und (induktive) Beweise über Eigenschaften berechenbarer Funktionen vereinfacht. Die rekursive Betrachtung erlaubt es auch, die Iteration mehrstelliger Funktionen zu beschreiben, was mit der einfachen Schreibweise $f \circ f \circ ... \circ f$ nicht möglich ist. Bei der sogenannten *primitiven Rekursion* übernimmt eine Initialfunktion g die Berechnung des Startwertes für null Iterationen, während eine Schrittfunktion h die Rolle einer erweiterten Komposition übernimmt: sie bestimmt, wie das bisherige Funktionsergebnis, die Anzahl der bisher durchgeführten Iterationen und die restlichen Eingabewerte

im nächsten Iterationsschritt weiterverarbeitet werden. Wir verwenden im folgenden die Bezeichnung $Pr[g,h]$ für die so entstehende Funktion.

Primitive Rekursion und Komposition sind derart mächtige Mechanismen, daß es schwer ist, berechenbare Funktionen zu konstruieren, die nicht mit diesen Operationen ausgedrückt werden können. Erst einige Jahrzehnte später stellte sich heraus, daß eine weitere Operation, die sogenannte *Minimierung* nötig war, um alle berechenbaren Funktionen zu beschreiben. Minimierung, üblicherweise mit μf bezeichnet, sucht die erste Nullstelle der Funktion f. Im Prinzip ist auch dies wieder ein iterativer Prozess. Im Gegensatz zur primitiven Rekursion wird allerdings die Anzahl der Iterationsschritte zu Beginn nicht vorgegeben. Die Suche wird erst beendet, wenn sie erfolgreich war und kann daher möglicherweise nicht enden.

Es ist leicht einzusehen, daß die oben beschriebenen Funktionen und Operationen auf dem Papier schematisch, also durch einfaches Befolgen einer Rechenvorschrift, ausgerechnet werden können. Wichtig ist aber auch, wie diese Rechenvorschrift formalisiert werden kann, so daß keine Zweifel über die durchzuführende Berechnung entstehen können. Heutzutage würde man dafür eine Programmiersprache entwickeln, die dann auf einem abstrakten Maschinenmodell, meist einer Art von Neumann Maschine, ausgeführt wird. Diese Maschinenmodelle wurden aber erst 50 Jahre nach den rekursiven Funktionen entwickelt und sind – aus mathematischer Sicht – auch wenig elegant, da die Beschreibung der Bedeutung eines Programms aufwendig ist und Beweise meist sehr kompliziert macht.

Der Kalkül der rekursiven Funktionen wurde daher als *Funktionenkalkül* gestaltet, der zur Beschreibung von Berechenbarkeit nur die Funktionen selbst verwendet. Man legt fest, welche *Grundfunktionen* als berechenbar gelten und welche *Operationen* auf berechenbare Funktionen angewandt werden können, um neue berechenbare Funktionen zu erzeugen. Die Funktionsargumente tauchen in der *Beschreibung* der berechenbaren Funktionen überhaupt nicht auf, sondern erst, wenn diese Funktionen für konkrete Eingabewerte ausgerechnet werden sollen.[1] In diesem Sinne ist der Kalkül der rekursiven Funktionen eine *sehr abstrakte mathematische Programmiersprache*, die man als Vorläufer des informatiktypischen *Baukastensystems* ansehen könnte.

Die wesentliche Schwierigkeit für Informatiker liegt heutzutage in der *Abstraktion*, also der Beschreibung des Funktionsverhaltens, ohne dabei explizit auf die Funktionsargumente einzugehen. Wir wollen diese Vorgehensweise an einigen Beispielen erläutern.

Beispiel 1 (Abstrakte Funktionsbeschreibungen)

– Wir wollen eine Beschreibung der Funktion $+_2$ finden, welche bei Eingabe einer Zahl x den Wert $x+2$ berechnet, also 2 addiert. Dabei darf in der Beschreibung der Funktion das Argument x nicht genannt werden.
Wir wissen, daß $x+2$ dasselbe ist wie die zweifache Addition von 1, also $x+1+1$. Abstrahiert ist dies die doppelte Anwendung der Nachfolgerfunktion s, also $s(s(x))$. Damit ist die Funktion $+_2$ die Komposition von s und s und die gesuchte Beschreibung ist $+_2 \equiv s \circ s$.

[1] Diese Vorgehensweise ist aus der Analysis bekannt, in der der Ableitungsoperator d/dx ebenfalls nur auf Funktionssymbole ohne Argumente angewandt wird, um die erste Ableitung einer Funktion *als Funktion* zu generieren.

– Gesucht ist eine abstrakte Beschreibung der Addition zweier Zahlen x und y.
  Das Ergebnis läßt sich darstellen als y-fache Addition von 1: $x+y = x+\underbrace{1+...+1}_{y-mal}$.
  Diese Iteration muß nun rekursiv beschrieben werden.
  Der Initialwert für $y = 0$ ist eine null-fache Addition von 1 $x+0 = x$
  Im Schrittfall verwenden wir das Ergebnis der Addition von x und y,
  um die Addition von x und y+1 zu bestimmen $x+(y+1) = (x+y)+1$)

  Wir müssen nun diese Gleichungen als Funktionsanwendungen beschreiben. Der Initialwert ergibt sich durch Anwendung der Projektion pr_1^1 auf die Eingabe x. Der Schrittfall ergibt sich durch Anwendung von s auf das bisherige Funktionsergebnis $x+y$. Die Anzahl y der bisher durchgeführten Iterationen und der Eingabewert x werden nicht verwendet und müssen durch eine entsprechende Projektion ausgeblendet werden. Die Schrittfunktion hätte somit die Gestalt $s \circ pr_i^3$, wobei i die Position des bisherige Funktionsergebnisses unter den 3 Argumenten bezeichnet.

  Bei der üblichen Formalisierung der primitiven Rekursion erwartet man das Funktionsergebnis an der letzten Stelle aller Argumente. Damit wäre die gesuchte Beschreibung der Additionsfunktion $add = Pr[pr_1^1, s \circ pr_3^3]$
– Das kleinste gemeinsame Vielfache zweier Zahlen x und y läßt sich am einfachsten durch einen Suchprozeß beschreiben. Man sucht, beginnend bei der größeren der beiden Zahlen x und y, die kleinste Zahl z, die von x und y geteilt wird.

  Um diese Suche in der Denkweise der rekursiven Funktionen zu beschreiben, muß man zunächst einen Test auf Teilbarkeit als rekursive Funktion $t_{divides}$ darstellen (siehe Beispiel 16), welche bei Eingabe (x, z) genau dann den Wert 0 liefert, wenn x die Zahl y teilt. Addiert man dann die Ergebnisse von $t_{divides}(x, z)$ und $t_{divides}(y, z)$, so bekommt man genau dann das Resultat 0, wenn beide Tests erfolgreich waren.

  Auf ähnliche Weise ergänzt man noch einen Test, ob $x \leq z$ bzw. $y \leq z$ gilt (siehe Beispiel 9). Dies ergibt insgesamt eine rekursive Funktion f, die genau dann das Resultat 0 liefert, wenn z ein gemeinsames Vielfaches von x und y und mindestens so groß wie beide Zahlen ist, und die gesuchte Beschreibung des kleinsten gemeinsamen Vielfachen ist $kgV = \mu f$.

  Eine detaillierte Herleitung der Beschreibung von kgV werden wir in Beispiel 15 auf Seite 16 angeben.

Wir werden im folgenden eine präzise Definition der rekursiven Funktionen angeben und den Entwurf und die Analyse rekursiver Funktionen an einer Reihe von Beispielen illustrieren. Dabei werden wir auch einige abgeleitete Programmierschemata vorstellen, welche den Entwurf rekursiver Funktionen erheblich erleichtern. Wir beginnen mit der wichtigen Unterklasse der *primitiv-rekursiven Funktionen*, stellen anschließend eine Funktion vor, die intuitiv berechenbar, aber nicht primitiv-rekursiv ist, und betrachten schließlich die Erweiterung auf *(μ-)rekursive Funktionen*. Wir werden zeigen, daß die Klasse der rekursiven Funktionen identisch mit der der Turing-berechenbaren Funktionen auf natürlichen Zahlen ist. Somit können Turingmaschinen und rekursive Funktionen wahlweise dazu eingesetzt werden, um nachzuweisen, daß eine Funktion berechenbar ist.

2 Primitiv-rekursive Funktionen

Die Klasse der primitiv-rekursiven Funktionen ist die Menge aller Funktionen, die aus elementaren Grundfunktionen durch beliebig häufige Anwendung von Komposition und primitiver Rekursion aufgebaut werden können. Wir werden sie im folgenden öfter mit dem Kürzel $\mathcal{PR}$ bezeichnen. Als Grundfunktionen verwenden wir die Nachfolgerfunktion, Projektionen und Konstanten.

Definition 2 (Grundfunktionen).

- *Die Nachfolgerfunktion* $s{:}\mathbb{N}\to\mathbb{N}$ *ist definiert durch* $s(x) = x{+}1$ *für alle* $x \in \mathbb{N}$
- *Die Projektionsfunktionen* $pr^n_k{:}\mathbb{N}^n\to\mathbb{N}$ *sind definiert durch* $pr^n_k(x_1,..,x_n) = x_k$ *für alle* $x_1,..,x_n \in \mathbb{N}$, *wobei* $n \in \mathbb{N}$ *und* $k \in \{1..n\}$.
- *Die Konstantenfunktionen* $c^n_k{:}\mathbb{N}^n\to\mathbb{N}$ *sind definiert durch* $c^n_k(x_1,..,x_n) = k$ *für alle* $x_1,..,x_n \in \mathbb{N}$ *wobei* $n,k \in \mathbb{N}$.

$\mathcal{G} = \{s\} \cup \{pr^n_k \mid n \in \mathbb{N}, 1{\le}k{\le}n\} \cup \{c^n_k \mid n,k \in \mathbb{N}\}$ *ist die Menge der Grundfunktionen.*

Es gibt minimalistische Formulierungen der rekursiven Funktionen, in denen als Grundfunktionen nur die Nachfolgerfunktion, die Projektionen und die einstellige Nullfunktion $z \equiv c^1_0$ verwendet werden, da alle anderen Konstantenfunktionen hieraus durch Komposition erzeugt werden können. Andere Formulierungen verwenden anstelle von pr^n_k die Bezeichnungen id^n_k oder π^n_k.

Definition 3 (primitiv-rekursive Operationen).

- *Die Komposition* $f{\circ}(g_1,..,g_n){:}\mathbb{N}^k\to\mathbb{N}$ *der Funktionen* $f{:}\mathbb{N}^n\to\mathbb{N}$, $g_1,..g_n{:}\mathbb{N}^k\to\mathbb{N}$ *für beliebige* $n,k \in \mathbb{N}$ *ist die eindeutig bestimmte Funktion* h *mit der Eigenschaft*
  $$h(\hat{x}) = f(\,g_1(\hat{x}),..,g_n(\hat{x})\,)$$[2]
  für alle $\hat{x} \in \mathbb{N}^k$.
- *Die primitive Rekursion* $Pr[f,g]{:}\mathbb{N}^k\to\mathbb{N}$ *zweier Funktionen* $f{:}\mathbb{N}^{k-1}\to\mathbb{N}$ *und* $g{:}\mathbb{N}^{k+1}\to\mathbb{N}$ *für beliebige* $k{\ge}1$ *ist die eindeutig bestimmte Funktion* h *für die gilt*
  $$h(\hat{x},0) = f(\hat{x}) \quad \textit{und} \quad h(\hat{x},y{+}1) = g(\hat{x},y,\,h(\hat{x},y))$$
  für alle $\hat{x} \in \mathbb{N}^{k-1}$ *und* $y \in \mathbb{N}$.

In der Denkweise imperativer Programmiersprachen[3] entspricht die Komposition einer Folge von Anweisungen und die primitive Rekursion einer einfachen Zählschleife, die allerdings in umgekehrter Reihenfolge abgearbeitet wird. Um $h = f{\circ}(g_1,..,g_n)$ zu berechnen, würde man folgende Anweisungen verwenden.

$\mathtt{y}_1 := g_1(\mathtt{x}_1,..,\mathtt{x}_k);\ \ ...;\ \ \mathtt{y}_n := g_n(\mathtt{x}_1,..,\mathtt{x}_k);\ \ h := f(\mathtt{y}_1,..,\mathtt{y}_n)$

Um $h = Pr[f,g]$ zu berechnen, würde man folgende Schleife verwenden.

---

[2] $\hat{x}$ ist abkürzend für ein Tupel $(x_1,...,x_m)$ von natürlichen Zahlen.

[3] In funktionalen Programmiersprachen können Komposition und primitive Rekursion wesentlich direkter ausgedrückt werden. $f{\circ}(g_1,..,g_n)$ entspricht dem komplexen Ausdruck

$\mathtt{fun\ (x_1,..,x_k)\ ->}\ f(\,g_1(\mathtt{x}_1,..,\mathtt{x}_k),..,g_n(\mathtt{x}_1,..,\mathtt{x}_k)\,)$

und $Pr[f,g]$ einer einfachen rekursive Funktionsdeklaration

$\mathtt{function\ h}(\hat{x},\mathtt{y}) = \mathtt{if\ y=0\ then}\ f(\hat{x})\ \mathtt{else}\ g(\hat{x},\mathtt{y-1},\mathtt{h}(\hat{x},\mathtt{y-1})\,)$

h := f(x$_1$,..,x$_k$); for i:=1 to y do h:=g(x$_1$,..,x$_k$,i-1,h)

Das Resultat h entspricht jeweils dem Funktionswert $h(x_1,..,x_k)$.

Definition 4 (primitiv-rekursive Funktionen).
Die Menge $\mathcal{PR}$ der primitiv-rekursiven Funktionen ist definiert als $\mathcal{PR} = \bigcup_{n=0}^{\infty} \mathcal{PR}_n$. Dabei sind die Klassen $\mathcal{PR}_i$ induktiv wie folgt definiert.

$$\mathcal{PR}_0 = \mathcal{G} \quad \textit{und}$$
$$\mathcal{PR}_{i+1} = \mathcal{PR}_i \cup \{f \circ (g_1,..,g_n) \mid n \in \mathbb{N}, f, g_1 ... g_n \in \mathcal{PR}_i\} \cup \{Pr[f,g] \mid f,g \in \mathcal{PR}_i\}$$

In anderen Worten, $\mathcal{PR}$ besteht aus den Grundfunktionen und allen Funktionen, die hieraus durch iterierte Komposition oder primitive Rekursion entstehen. $\mathcal{PR}_i$ beschreibt die Funktionen, bei deren Aufbau maximal i Iterationen benötigt. Die obige Definition präzisiert diesen intuitiven Gedanken, ist aber für Nicht-Mathematiker oft schwer zu handhaben. In vielen Fällen ist die folgende Charakterisierung handlicher.

Korollar 5

1. *Die Nachfolgerfunktion s ist primitiv-rekursiv.*
2. *Die Projektionsfunktionen pr_k^n sind primitiv-rekursiv für alle $n \in \mathbb{N}$ und $k \in \{1..n\}$.*
3. *Die Konstantenfunktionen c_k^n sind primitiv-rekursiv für alle $n, k \in \mathbb{N}$.*
4. *Die Komposition $f \circ (g_1,..,g_n)$:$\mathbb{N}^k \to \mathbb{N}$ der Funktionen f:$\mathbb{N}^n \to \mathbb{N}$, $g_1,..g_n$:$\mathbb{N}^k \to \mathbb{N}$ ist primitiv-rekursiv für alle $n, k \in \mathbb{N}$, wenn $f, g_1 ... g_n$ primitiv-rekursiv sind.*
5. *Die primitive Rekursion $Pr[f,g]$:$\mathbb{N}^k \to \mathbb{N}$ zweier Funktionen f:$\mathbb{N}^{k-1} \to \mathbb{N}$ und g:$\mathbb{N}^{k+1} \to \mathbb{N}$ ist primitiv-rekursiv für alle $k \in \mathbb{N}$, wenn f und g primitiv-rekursiv sind.*

Will man also zeigen, daß eine gegebene Funktion h primitiv-rekursiv ist, so muß man – wenn h nicht eine der Grundfunktionen ist – entweder zeigen, daß es primitiv-rekursive Funktionen $f, g_1 ... g_n$ gibt, so daß $h = f \circ (g_1,..,g_n)$ ist, oder zwei primitiv-rekursive Funktionen f und g finden, so daß $h = Pr[f,g]$ gilt. Zuweilen hilft es, für h einen *primitiv-rekursiven Ausdruck* zu konstruieren, also einen Ausdruck, der nur aus den Symbolen s pr_k^n und c_k^n und Anwendungen der Operatoren $\circ$ und Pr besteht.[4] Für den Nachweis, daß h primitiv-rekursiv ist, ist dies jedoch nicht erforderlich.

Wir wollen im folgenden die Analyse und Konstruktion primitiv-rekursiver Funktionen an einigen Beispielen erklären und zeigen, daß die wichtigsten arithmetischen Funktionen primitiv-rekursiv sind.

[4] Primitiv-rekursiven Ausdrücke sind aus heutiger Sicht eine Art von Programmiersprache in sehr kompakter Notation. Die Syntax dieser Programmiersprache besteht nur aus den Symbolen s, pr_k^n und c_k^n für die Grundfunktionen, und Ausdrücken der Form $f \circ (g_1,..,g_n)$ und $Pr[f,g]$, wobei $f, g, g_1, .., g_n$ primitiv-rekursive Ausdrücke sein müssen. Letztere müssen zusätzlich die in Definition 3 genannten Stelligkeitsbedingungen erfüllen. Dies entspricht den Typ-Bedingungen in modernen Programmiersprachen. Ausdrücke, welche diese Bedingungen nicht erfüllen, würden von einem Typechecker als unzulässig zurückgewiesen.
Zusätzlich erlaubt sind abkürzende Bezeichnungen für existierende primitiv-rekursive Ausdrücke wie z.B. p als Abkürzung für $Pr[c_0^0, pr_1^2]$ (Beispiel 8) oder add, sub, mul, etc. sowie die in Abschnitt 2.2 besprochenen primitiv-rekursiven Programmierschemata $Cond[t,f,g]$ als Abkürzung für $Pr[pr_1^2, pr_2^4] \circ (f,g,t)$ (Satz 1), wobei f, g und t primitiv-rekursive Ausdrücke mit entsprechenden Stelligkeiten sein müssen, oder Σf, Πf, $Mn[f]$, etc. Diese Abkürzungen entsprechen Unterprogrammen bzw. Makros in heutigen Programmiersprachen.

2.1 Analyse und Konstruktion primitiv-rekursiver Funktionen

Es ist relativ leicht, einen vorgegebenen primitiv-rekursiven Ausdruck auf bestimmten Argumenten auszurechnen. Man muß hierzu nur schrittweise die Interpretation der Funktions- und Operationssymbole einsetzen, bis ein Ergebnis dasteht. Wegen der großen Menge an Details sind Computer für diese Tätigkeit normalerweise besser geeignet als Menschen. Nichtsdestotrotz gewinnt man ein gewisses Verständnis der Begriffe, wenn man dies an einfachen Beispielen von Hand durchrechnet.

Beispiel 6 (Auswertung eines primitiv-rekursiven Ausdrucks)
Wir wollen den Ausdruck $Pr[pr^1_1,\ s{\circ}pr^3_3](5,2)$ *auswerten. Hierzu betrachten wir zunächst den äußeren Operator des Ausdrucks* $Pr[pr^1_1,\ s{\circ}pr^3_3]$*. Dieser ist* Pr*, d.h. der Ausdruck hat die Gestalt* $h = Pr[f,g]$*, wobei* $f = pr^1_1 \in \mathbb{N}{\to}\mathbb{N}$ *und* $g = s{\circ}pr^3_3 \in \mathbb{N}^3{\to}\mathbb{N}$*.*

Die Definition der primitiven Rekursion $Pr[f,g]$ *(Definition 3) unterscheidet zwei Fälle: entweder ist das letzte Argument 0 oder es ist Nachfolger einer Zahl* y*. In unserem Fall ist das letzte Argument eine 2, also der Nachfolger von 1, und wir beginnen mit der Auswertung von* $h(x,y+1) = g(x,y,h(x,y))$ *für* $x = 5$ *und* $y = 1$*, also von* $g(5,1,h(5,1))$ *bzw. von* $s{\circ}pr^3_3(5,1,h(5,1))$*.*

An dieser Stelle gibt es zwei Möglichkeiten des weiteren Vorgehens. Wir können zunächst $h(5,1)$ *weiter auswerten und anschließend den äußeren Ausdruck oder umgekehrt. In diesem Fall enthält der äußere Ausdruck eine Projektionsfunktion, deren Auswertung einen Ausruck üblicherweise erheblich vereinfacht. Deswegen lösen wir zunächst die Komposition* $s{\circ}pr^3_3$ *gemäß Definition 3 auf und erhalten den Ausdruck* $s(pr^3_3(5,1,h(5,1)))$*. Im nächsten Schritt wertet man nun die innere Projektionsfunktion* pr^3_3 *aus, also* $pr^3_3(5,1,h(5,1)) = h(5,1)$*, und erhält* $s(h(5,1))$*.*

Nun müssen wir erneut die primitive Rekursion $h = Pr[f,g]$ *erneut analysieren. Jetzt ist das letzte Argument eine 1, also der Nachfolger von , und wir beginnen mit der Auswertung von* $h(x,y+1) = g(x,y,h(x,y))$ *für* $x = 5$ *und* $y = 0$*, also von* $g(5,0,h(5,0))$ *bzw. von* $s{\circ}pr^3_3(5,0,h(5,0))$*. Nach Auflösen der Komposition und Auswertung der innere Projektionsfunktion ergibt sich* $h(5,1) = s(h(5,0))$*, also insgesamt* $h(5,2) = s(h(5,1)) = s(s(h(5,0)))$*.*

In der nun folgenden Analyse der primitive Rekursion $h = Pr[f,g]$ *stoßen wir auf den Basisfall* $h(x,0) = f(x)$ *für* $x = 5$*, und berechnen* $h(5,0) = f(5) = pr^1_1(5) = 5$*. Setzen wir dies in die obige Gleichung ein und werten weiter aus, so erhalten wir als Endergebnis* $h(5,2) = s(s(h(5,0))) = s(s(5)) = s(6) = \mathbf{7}$*.*
Die folgende Gleichungskette faßt das obige Argument in knapper Form zusammen.

$$
\begin{array}{ll}
Pr[pr^1_1,\ s{\circ}pr^3_3](5,2) & \\
= s{\circ}pr^3_3(5,1,\ Pr[pr^1_1,\ s{\circ}pr^3_3](5,1)) & = s(pr^3_3(5,1,\ Pr[pr^1_1,\ s{\circ}pr^3_3](5,1))) \\
= s(Pr[pr^1_1,\ s{\circ}pr^3_3](5,1)) & \\
= s(s{\circ}pr^3_3(5,0,\ Pr[pr^1_1,\ s{\circ}pr^3_3](5,0))) & = s(s(pr^3_3(5,0,\ Pr[pr^1_1,\ s{\circ}pr^3_3](5,0)))) \\
= s(s(Pr[pr^1_1,\ s{\circ}pr^3_3](5,0))) & = s(s(pr^1_1(5))) \\
= s(s(5)) & = s(6) \ = \ \mathbf{7}
\end{array}
$$

Diese Abarbeitungsreihenfolge ist nur eine von vielen Möglichkeiten. Hätten wir vorrangig die inneren Ausdrücke ausgewertet, so wäre folgende Gleichungskette entstanden.

$$
\begin{array}{lll}
Pr[pr_1^1,\ s{\circ}pr_3^3](5,2) & & \\
= s{\circ}pr_3^3(5,1,\ Pr[pr_1^1,\ s{\circ}pr_3^3](5,1)) & & \\
= s{\circ}pr_3^3(5,1,\ s{\circ}pr_3^3(5,0,\ Pr[pr_1^1,\ s{\circ}pr_3^3](5,0))) & = s{\circ}pr_3^3(5,1,\ s{\circ}pr_3^3(5,0,\ pr_1^1(5))) & \\
= s{\circ}pr_3^3(5,1,\ s{\circ}pr_3^3(5,0,\ 5)) & = s{\circ}pr_3^3(5,1,\ s(pr_3^3(5,0,\ 5))) & \\
= s{\circ}pr_3^3(5,1,\ s(5)) & & \\
= s{\circ}pr_3^3(5,1,\ 6) & = s(pr_3^3(5,1,\ 6)) & \\
= s(6) & = \mathbf{7} &
\end{array}
$$

Anstelle einer geschlossenen Gleichungskette hätte man auch die Rekursion durch eine Iteration ersetzen und die Werte $Pr[pr_1^1,\ s{\circ}pr_3^3](5,i)$ für $i = 0,1,2$ der Reihe nach berechnen können. Dies ist für Menschen normalerweise übersichtlicher, da sie keinen Abarbeitungsstack verwalten müssen. Einem Computer, der Ausdrücke schematisch verarbeiten muß, bleibt diese elegantere Vorgehensweise natürlich verwehrt.

Die Analyse eines vorgegebenen primitiv-rekursiven Ausdrucks beinhaltet neben seiner Auswertung für konkrete vor allem natürlich die Frage, welche mathematische Funktion durch diesen Ausdruck berechnet wird und den Nachweis, daß dies tatsächlich der Fall ist. Dabei liefert eine gründliche Analyse oft alle wesentlichen Argumente für den Beweis, so daß dieser nur noch die Kernpunkte in einer schlüssigen Reihenfolge zusammenfaßt.

Beispiel 7 (Analyse eines primitiv-rekursiven Ausdrucks)
Wir wollen den Ausdruck $f_1 = Pr[pr_1^1,\ s{\circ}pr_3^3]$ aus Beispiel 6 genauer analysieren. Hierzu betrachten wir zunächst einmal die Stelligkeit der Funktion f_1. Nach Definition 3 ergibt sich die Stelligkeit von f_1 aus der Stelligkeit der Funktionen pr_1^1 und $s{\circ}pr_3^3$. In Definition 2 wurde die Projektionsfunktion pr_1^1 als einstellige Funktion auf den natürlichen Zahlen definiert, d.h. es gilt $pr_1^1{:}\mathbb{N}{\to}\mathbb{N}$. pr_3^3 ist dreistellig und gemäß Definition 3 folgt auch $s{\circ}pr_3^3{:}\mathbb{N}^3{\to}\mathbb{N}$. Da die Stelligkeit von $Pr[f,g]$ zwischen der von f und g liegt, folgt damit, daß f_1 zweistellig sein muß, also $f_1{:}\mathbb{N}^2{\to}\mathbb{N}$.

Um einen Eindruck vom Funktionsverhalten zu bekommen, werten wir als nächstes f_1 auf einigen Beispielargumenten aus.

$$
\begin{array}{lll}
f_1(2,2) = Pr[pr_1^1, s{\circ}pr_3^3](2,2) & & \\
\quad = (s{\circ}pr_3^3)(2,1,\ f_1(2,1)) & = s(pr_3^3(2,1,\ f_1(2,1))) & \\
\quad = s(f_1(2,1)) & = s(Pr[pr_1^1, s{\circ}pr_3^3](2,1)) & \\
\quad = s((s{\circ}pr_3^3)(2,0,\ f_1(2,0))) & = s(s(pr_3^3(2,0,\ f_1(2,0)))) & \\
\quad = s(s(f_1(2,0))) & = s(s(Pr[pr_1^1, s{\circ}pr_3^3](2,0))) & \\
\quad = s(s(pr_1^1(2))) & & \\
\quad = s(s(2)) & = s(3) & = \mathbf{4} \\
f_1(6,1) = Pr[pr_1^1, s{\circ}pr_3^3](6,1) & & \\
\quad = (s{\circ}pr_3^3)(6,0,\ f_1(6,0)) & = s(pr_3^3(6,0,\ f_1(6,0))) & \\
\quad = s(f_1(6,0)) & = s(Pr[pr_1^1, s{\circ}pr_3^3](6,0)) & \\
\quad = s(pr_1^1(6)) & = s(6) & = \mathbf{7} \\
f_1(0,0) = Pr[pr_1^1, s{\circ}pr_3^3](0,0) & = pr_1^1(0) & = \mathbf{0}
\end{array}
$$

Das Ergebnis ist jeweils die Summe der beiden Eingaben, also liegt die Vermutung nahe, daß f_1 möglicherweise die Additionsfunktion darstellt. Um dies zu überprüfen,

analysieren wir das rekursiven Verhalten von f_1 für beliebige Eingaben. Für beliebige $x, y \in \mathbb{N}$ ist

$$\begin{aligned} f_1(x,0) &= pr_1^1(x) &&= x \\ f_1(x,y{+}1) &= (s{\circ}pr_3^3)(x,y,f_1(x,y)) = s(f_1(x,y)) &&= f_1(x,y){+}1 \end{aligned}$$

Bis auf die fehlende Infix-Notation ist dies genau die Rekursionsgleichung der Addition, die wir aus den Peanoaxiomen kennen

$$\begin{aligned} x{+}0 &= x \\ x{+}(y{+}1) &= (x{+}y){+}1 \end{aligned}$$

Damit ist bewiesen, daß $f_1(x,y) = x{+}y$ für beliebige $x, y \in \mathbb{N}$ gilt und daß f_1 die Additionsfunktion $add{:}\mathbb{N}^2{\to}\mathbb{N}$ ist, die durch $add(x,y) = x{+}y$ definiert ist. Außerdem haben wir bewiesen, daß die Additionsfunktion add primitiv-rekursiv ist.

Im obigen Beispiel haben wir bisher wir eigentlich nur gezeigt, daß die Funktion f_1 die gleiche Rekursionsgleichung besitzt wie die Additionsfunktion. Daß f_1 dann auch tatsächlich identisch mit der Funktion add ist, folgt hieraus mit dem sogenannten *Extensionalitätsprinzip*:

> *Zwei Funktionen sind gleich, wenn sie die gleiche Stelligkeit besitzen und auf allen Argmenten den gleichen Wert annehmen.*

Aufgrund dieses Prinzips müssen wir nur zeigen daß f_1 und add auf allen Paaren $(x,y) \in \mathbb{N}$ das gleiche Ergebnis liefern. Hierfür geben wir einen Induktionsbeweis an:

Es sei $x \in \mathbb{N}$ beliebig aber fest. Wir zeigen $f_1(x,y) = x{+}y$ für alle $y \in \mathbb{N}$.
Induktionsanfang $y{=}0$: Es ist $f_1(x,0) = x = x{+}0$
Induktionsannahme: Sei $f_1(x,y) = x{+}y$ für ein beliebiges y bewiesen
Induktionsschritt $y{+}1$: Es ist $f_1(x,y{+}1) = f_1(x,y){+}1 = (x{+}y){+}1 = x{+}(y{+}1)$

Man sieht, daß sich der Induktionsbeweis unmittelbar aus den Rekursionsgleichungen der primitiven Rekursion ergibt, da diese die Schlüsselargumente des Basis- und Schrittfalls der Induktion beschreiben. Aus diesem Grund werden wir im folgenden keine weiteren Induktionsbeweise zur Korrektheit primitiv-rekursiver Funktionen angeben, sondern nur noch die rekursiven Funktionsgleichungen analysieren und das folgende abgewandelte Extensionalitätsprinzip verwenden.

> *Zwei Funktionen auf den natürlichen Zahlen sind gleich, wenn sie die gleiche Stelligkeit besitzen und identische rekursive Funktionsgleichungen besitzen.*

Etwas schwieriger als die Analyse eines vorgegebenen primitiv-rekursiven Ausdrucks ist der Nachweis, daß eine bestimmte Funktion primitiv-rekursiv ist. Meist läuft diese Aufgabe darauf hinaus, einer Beschreibung der Funktion in durch einen primitiv-rekursiven Ausdruck zu finden, also im Endeffekt ein Programm in der Sprache $\mathcal{PR}$ schreiben. Im Prinzip reicht es aber zu zeigen, daß die Funktion sich durch Komposition und primitive Rekursion aus bekannten anderen primitiv-rekursiven Funktionen zusammensetzen läßt. Wenn man also bereits eine gewisse Bibliothek von primitiv-rekursiven Funktionen zur Verfügung hat, dann ist dieser Nachweis deutlich leichter – genauso, wie es in jeder anderen Programmiersprache leichter ist, Programme aus bekannten Bibliotheksfunktionen zusammen zu setzen als sie von Grund auf neu zu schreiben.

Erfahrungsgemäß macht die primitive Rekursion dabei die größten Schwierigkeiten. Man kennt die gesuchte Funktion h und muß nun zwei Funktionen f und g finden mit der Eigenschaft, daß h sich als $Pr[f,g]$ darstellen läßt. Dabei müssen vor allem die Rahmenbedingungen der Stelligkeiten und Anordnung der Argumente eingehalten werden. f muß eine Stelle weniger haben als h und g eine mehr. g muß als letztes Argument den Vorgängerwert der Rekursion verwenden, davor die Rekursionsvariable, also das letzte Argument von h, und davor die restlichen Argumente von h in genau der gleichen Reihenfolge. Wie in jeder Programmiersprache darf von derartigen formalen Rahmenbedingungen nicht abgewichen werden. Deswegen braucht man Projektionsfunktionen und Komposition, um die Argumente in der richtigen Anzahl und an der richtigen Stelle einsetzen zu können.

Um f und g zu finden, empfiehlt es sich, *zunächst eine Funktionsgleichung* für die vorgegebene Funktion *aufzustellen*, in der nur bereits bekannte Funktionen und die Argumente auftauchen. Dies kann eine Rekursionsgleichung sein, eine einfache Hintereinanderausführung mehrerer Funktionen, oder eines der primitiv-rekursiven Programmierschemata enthalten, die wir im Abschnitt 2.2 diskutieren werden. Gelingt dies nicht unmittelbar, so muß das Problem eventuell in kleinere Bestandteile zerlegt werden, für die dann eine Funktionsgleichung aufgestellt wird. Die Gesamtlösung ergibt sich dann durch Komposition der einzelnen Teillösungen.

Für den Nachweis, daß die vorgegebene Funktion h primitiv-rekursiv ist, reichen diese Funktionsgleichungen aus. Man weiß dann, daß h sich durch Komposition und primitive Rekursion aus anderen primitiv-rekursiven Funktionen ergibt und damit selbst primitiv-rekursiv ist. Will man zusätzlich den primitiv-rekursiven Ausdruck angeben, der h beschreibt, so muß man die Funktionsgleichungen in die Operatorschreibweise umwandeln, indem man schrittweise die Argumente nach außen schiebt und hierfür ggf. Komposition und Projektionen einsetzt.

Beispiel 8 (Programmierung mit primitiver Rekursion)
Wir wollen zeigen, daß die Vorgängerfunktion $p{:}\mathbb{N}\to\mathbb{N}$, *definiert durch* $p(n) = n\dot{-}1$, *primitiv-rekursiv ist. Dabei beschreibt* $\dot{-}$ *die Subtraktion auf den natürlichen Zahlen, die im Gegensatz zur Subtraktion auf ganzen Zahlen keine negativen Werte annehmen kann, d.h.* $x\dot{-}y = 0$, *falls* $x<y$. *Für* $x\geq y$ *ist* $x\dot{-}y$ *dasselbe wie* $x-y$.

Wir beginnen mit den Funktionsgleichungen für die Vorgängerfunktion. Wir wissen, daß $p(n) = 0$ *für* $n<1$ *ist und* $p(n) = n-1$ *für* $n\geq 1$. *Diese Fallunterscheidung ähnelt dem Gleichungsschema der primitiven Rekursion in Definition 3, bis darauf, daß die Tatsache, daß eine natürliche Zahl größer als Null ist, dort durch die Schreibweise* $y+1$ *ausgedrückt wird. Schreiben wir die Fallunterscheidung entsprechend um, so erhalten wir folgende Funktionsgleichungen.*

$$\begin{aligned} p(0) &= 0\dot{-}1 &&= 0 \\ p(y+1) &= (y+1)\dot{-}1 &&= y \end{aligned}$$

Wir müssen nun noch die beiden Fälle durch bekannte primitiv-rekursive Funktionen ausdrücken, die den Vorschriften der primitiven Rekursion genügen. Für den ersten Fall müssen wir also eine nullstellige Funktion $f{:}\mathbb{N}^0\to\mathbb{N}$ *mit* $p(0) = f()$ *finden und für den zweiten Fall eine zweistellige Funktion* $g{:}\mathbb{N}^2\to\mathbb{N}$ *mit* $p(y+1) = g(y, p(y))$.

Die einzige nullstellige Funktion, die den Wert $p(0) = 0$ als Ergebnis liefert ist die nullstellige Konstantenfunktion c_0^0. Wir wählen diese als unsere Funktion f. Die zweistellige Funktion g muß nach obiger Bedingung bei Eingabe der Werte $(y, p(y))$ das Resultat $p(y+1) = y$ liefern. Hierzu reicht es, das erste Argument y aus den zwei Eingaben herausgreifen. Wir verwenden hierzu die zweistellige Projektionsfunktion pr_1^2.

Insgesamt folgt also, daß die Vorgängerfunktion p sich durch primitive Rekursion aus den Funktionen $f = c_0^0$ und $g = pr_1^2$ ergibt, und damit ist bewiesen, daß die Vorgängerfunktion p primitiv-rekursiv ist. Aus der obigen Analyse ergibt sich auch direkt der primitiv-rekursive Ausdruck für die Vorgängerfunktion: $p = Pr[c_0^0, pr_1^2]$.

Im folgenden geben wir eine Reihe von Beispielen primitiv-rekursiver Funktionen. Wir beschränken uns dabei auf die Angabe der Funktionsgleichungen und des entstehenden primitiv-rekursive Ausdrucks und überlassen es dem Leser, Details zu ergänzen.

Beispiel 9 (Wichtige primitiv-rekursive Funktionen)

– *Die Subtraktionsfunktion sub:$\mathbb{N}^2 \to \mathbb{N}$ mit $sub(n, m) = n \dot{-} m$ ist primitiv-rekursiv.*
Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{array}{lll} sub(x,0) & = x & = pr_1^1(x) \\ sub(x,y{+}1) = x \dot{-} (y{+}1) & = (x \dot{-} y) \dot{-} 1 = p(x \dot{-} y) & = (p \circ pr_3^3)(x, y, sub(x,y)) \end{array}$$

Damit entsteht sub durch primitive Rekursion aus primitiv-rekursiven Funktionen und ist selbst primitiv-rekursiv. Der zugehörige Ausdruck ist $sub = Pr[pr_1^1, p \circ pr_3^3]$.

– *Die Multiplikationsfunktion mul:$\mathbb{N}^2 \to \mathbb{N}$ mit $mul(n, m) = n{*}m$ ist primitiv-rekursiv.*
Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{array}{lll} mul(x,0) & = 0 & = c_0^1(x) \\ mul(x,y{+}1) = x{*}(y{+}1) & = mul(x,y){+}x & = (add \circ (pr_1^3, pr_3^3))(x, y, mul(x,y)) \end{array}$$

Damit ist mul primitiv-rekursiv. Der zugehörige primitiv-rekursive Ausdruck ist $mul = Pr[c_0^1, (add \circ (pr_1^3, pr_3^3))]$.

– *Die Exponentialfunktion exp:$\mathbb{N}^2 \to \mathbb{N}$ mit $exp(n, m) = n^m$ ist primitiv-rekursiv.*
Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{array}{lll} exp(x,0) & = 1 & = c_1^1(x) \\ exp(x,y{+}1) = x^{y+1} & = exp(x,y){*}x & = (mul \circ (pr_1^3, pr_3^3))(x, y, exp(x,y)) \end{array}$$

Damit ist mul primitiv-rekursiv. Der zugehörige primitiv-rekursive Ausdruck ist $exp = Pr[c_1^1, (mul \circ (pr_1^3, pr_3^3))]$.

– *Die Fakultätsfunktion fak:$\mathbb{N} \to \mathbb{N}$ mit $fak(n) = n! = 1{*}2{*}...{*}n$ ist primitiv-rekursiv.*
Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{array}{lll} fak(0) & = 1 & = c_1^0() \\ fak(y{+}1) = (y{+}1)! & = (y{+}1){*}fak(y) & = (mul \circ (s \circ pr_1^2, pr_2^2))(y, fak(y)) \end{array}$$

Damit ist fak primitiv-rekursiv. Der zugehörige primitiv-rekursive Ausdruck ist $fak = Pr[c_1^0, (mul \circ (s \circ pr_1^2, pr_2^2))]$.

– *Die Vorzeichenfunktion* $sign$*:*$\mathbb{N}\to\mathbb{N}$ *mit* $sign(n) = \begin{cases} 0 \text{ falls } n{=}0 \\ 1 \text{ sonst} \end{cases}$ *ist primitiv-rekursiv.*
Für diese Funktion gibt es eine Reihe verschiedenartiger Bezeichnungen, wie zum Beispiel Signum-Funktion und Test auf 0 (t_0).[5] Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{aligned} sign(0) &= 0 = c_0^0() \\ sign(y{+}1) &= 1 = c_1^2(y, sign(y)) \end{aligned}$$

Damit ist $sign$ primitiv-rekursiv. Der zugehörige Ausdruck ist $sign = Pr[c_0^0, c_1^2]$.

– *Der Größenvergleichstest* $t_{\leq}$*:*$\mathbb{N}^2\to\mathbb{N}$ *mit* $t_{\leq}(n,m) = \begin{cases} 0 \text{ falls } n{\leq}m \\ 1 \text{ sonst} \end{cases}$ *ist primitiv-rekursiv.*
Für diese Funktion brauchen wir keine Rekursionsgleichung, denn $x{\leq}y$ gilt genau dann, wenn $x{-}y{\leq}0$ bzw. wenn $x \dot{-} y = 0$ ist. Damit ergibt sich $t_{\leq}$ durch Komposition aus Subtraktion und Vorzeichenfunktion und ist primitiv rekursiv. Der zugehörige Ausdruck ist $t_{\leq} = sign{\circ}sub$.

– *Der Gleichheitstest* $t_{=}$*:*$\mathbb{N}^2\to\mathbb{N}$ *mit* $t_{=}(n,m) = \begin{cases} 0 \text{ falls } n{=}m \\ 1 \text{ sonst} \end{cases}$ *ist primitiv-rekursiv.*
Zwei Zahlen n und m sind gleich, wenn sowohl $n{\leq}m$ als auch $m{\leq}n$ gilt, also wenn sowohl $t_{\leq}(n,m)$ als auch $t_{\leq}(m,n)$ den Wert 0 annimmt. Ansonsten sind n und m verschieden. Da sich die Tatsache, daß zwei Berechnungen den Wert 0 annehmen müssen, durch eine simple Addition der beiden Berechnungsergebnisse beschreiben läßt, können wir $t_{=}(n,m)$ als Summe der Werte $t_{\leq}(n,m)$ und $t_{\leq}(m,n)$ ausdrücken. $t_{\leq}(n,m){+}t_{\leq}(m,n)$ ergibt den Wert 0, wenn $n{=}m$ ist und den Wert 1, wenn $n{<}m$ oder $m{<}n$ ist. Da $n{<}m$ und $m{<}n$ nicht gleichzeitig der Fall sein kann, wird der Wert 2 niemals erreicht und die Summe muß daher nicht normiert werden.
Damit ergibt sich $t_{=}$ durch Komposition aus Addition, $t_{\leq}$ und Projektionen und ist primitiv rekursiv. Der zugehörige Ausdruck ist $t_{=} = add{\circ}(t_{\leq}, t_{\leq}{\circ}(pr_2^2, pr_1^2))$.

Wir werden später weitere wichtige Funktionen als primitiv-rekursiv nachweisen. Zuvor stellen wir jedoch einige Programmierschemata vor, die primitiv-rekursive Funktionen auf einfache Weise zu neuen primitiv-rekursiven Funktionen zusammensetzen können und die Menge der verwendbaren primitiv-rekursiven Operatoren erweiteren.

[5] In der Theorie der rekursiven Funktionen hat es sich eingebürgert, bei Testfunktionen im Erfolgsfall eine 0 zurückzugeben und sonst einen Wert, der größer als 0 ist. Dies liegt zum Teil daran, daß die Null das Grundelement der natürlichen Zahlen bildet und damit eine herausragende Rolle gegenüber allen anderen Zahlen besitzt. Außerdem ist es in vielen Anwendungen einfacher, nach Nullen zu suchen, als nach Einsen. Im Gegensatz dazu identifiziert man in der booleschen Algebra den Wert *wahr* meist mit der 1 und die 0 mit dem Wert *falsch*.

Dadurch ergibt sich eine unvermeidbare Diskrepanz zwischen den von der booleschen Algebra geprägten charakteristischen Funktionen, die wir im Kontext der Turing-Berechenbarkeit eingeführt hatten, und zahlentheoretisch geprägten Testfunktionen im Kontext der rekursiven Funktionen, die wir in diesem Artikel verwenden. Aus theoretischer Sicht ist dieser Unterschied unbedeutend, aber beim konkreten Ausprogrammieren einer Funktion muß natürlich darauf geachtet werden, die passenden Ausgabewerte zu generieren.

2.2 Primitiv-rekursive Programmierschemata

Eines der häufigsten Schemata in der Programmierung ist die *Fallunterscheidung*, auch *Konditional* genannt. Man führt einen Test durch und wählt abhängig vom Ergebnis eine von zwei möglichen Funktionen aus. Wir werden zeigen, daß dieses Schema immer eine primitiv-rekursive Funktion liefert, wenn der Test und die beteiligten Funktionen primitiv-rekursiv sind.

Definition 10 (Fallunterscheidung).
Das Konditional $Cond[t,f,g]{:}\mathbb{N}^k{\to}\mathbb{N}$ *der Funktionen* $t,f,g{:}\mathbb{N}^k{\to}\mathbb{N}$ *für beliebige* $k\in\mathbb{N}$ *ist die eindeutig bestimmte Funktion* h *mit der Eigenschaft* $h(\hat{x}) = \begin{cases} f(\hat{x}) \text{ falls } t(\hat{x})=0 \\ g(\hat{x}) \text{ sonst} \end{cases}$

Theorem 1. *Das Konditional* $Cond[t,f,g]{:}\mathbb{N}^k{\to}\mathbb{N}$ *der Funktionen* $t,f,g{:}\mathbb{N}^k{\to}\mathbb{N}$ *ist primitiv-rekursiv für alle* $k\in\mathbb{N}$*, wenn* t*,* f *und* g *primitiv rekursiv sind.*

Beweis: Sei $h = Cond[t,f,g]$. h nimmt den Wert von f an, wenn die Testfunktion t den Wert 0 annimmt und den Wert von g, wenn die Testfunktion einen Wert größer als 0 annimmt. Wir verwenden einen einfachen numerischen Trick, um dies mithilfe von Additionen und Multiplikationen zu definieren.
Wir wissen, daß $f(\hat{x}) = 1{*}f(\hat{x}) = 1{*}f(\hat{x}){+}0{*}g(\hat{x})$ und $g(\hat{x}) = 0{*}f(\hat{x}){+}1{*}g(\hat{x})$ gilt. Um also die Fallunterscheidung zu berechnen, müssen wir f mit 1 multiplizieren und g mit 0, wenn die Testfunktion den Wert 0 annimmt und f mit 0 multiplizieren und g mit 1, wenn die Testfunktion einen Wert größer als 0 annimmt. Das bedeutet, der Faktor für f muß den Wert der Testfunktion "umdrehen", was man erreichen kann, wenn man ihn von 1 abzieht, und der Faktor für g muß den Wert der Testfunktion "normieren", was man mit der Vorzeichenfunktion erreichen kann.
Damit erhalten wir $h(\hat{x}) = (1\dot{-}t(\hat{x})) * f(\hat{x}) + sign(t(\hat{x})) * g(\hat{x})$, d.h. h ergibt sich durch Komposition aus Addition, Multiplikation, Subtraktion und Vorzeichenfunktion und ist primitiv rekursiv. Der zugehörige Ausdruck ist für beliebige t, f und g

$$Cond[t,f,g] = add{\circ}(mul{\circ}(sub{\circ}(c_1^k, sign{\circ}t), f),\ mul{\circ}(sign{\circ}t, g))$$

Anstelle einer Beschreibung der Fallunterscheidung mit numerischen Mitteln kann man das Konditional auch als eine Komposition einer *Fallunterscheidungsfunktion* $cond{:}\mathbb{N}^3{\to}\mathbb{N}$ mit den Funktionen f, g und t darstellen und erstere direkt durch eine primitive Rekursion repräsentieren.
Wir definieren dazu $cond(x,y,z) = \begin{cases} x \text{ falls } z=0 \\ y \text{ sonst} \end{cases}$
Die zugehörigen primitiv-rekursiven Funktionsgleichungen lauten

$$\begin{aligned} cond(x,y,0) &= x = pr_1^2(x,y) \\ cond(x,y,z{+}1) &= y = pr_2^4(x,y,z,\ cond(x,y,z)) \end{aligned}$$

Damit ist $cond = Pr[pr_1^2, pr_2^4]$, also eine primitiv-rekursive Funktion, und für das Konditional ergibt sich insgesamt folgender primitiv-rekursiver Ausdruck

$$Cond[t,f,g] = Pr[pr_1^2, pr_2^4]{\circ}(f,g,t)$$

□

Viele mathematische Funktionen lassen sich natürlich durch Aufsummieren oder Aufmultiplizieren über eine andere Funktion beschreiben. So ist zum Beispiel die Fakultät $n!$ das Produkt $\prod_{i=1}^{n} i$ der Zahlen zwischen 1 und n und die Potenz x^n das n-fache Produkt $\prod_{i=1}^{n} x$ der Zahl x. Wir zeigen nun, daß die Summe und das Produkt über eine Funktion f primitiv-rekursiv ist, wenn dies für die beteiligte Funktion f gilt.

Definition 11 (Generelle Summe und generelles Produkt).

- *Die Generelle Summe $\Sigma f{:}\mathbb{N}^k{\to}\mathbb{N}$ der Funktion $f{:}\mathbb{N}^k{\to}\mathbb{N}$ für beliebige $k{\geq}1$ ist die eindeutig bestimmte Funktion h mit der Eigenschaft $h(\hat{x},y) = \Sigma_{i=0}^{y} f(\hat{x},i)$ für alle $\hat{x}\in\mathbb{N}^{k-1}$ und $y\in\mathbb{N}$.*
- *Das Generelle Produkt $\Pi f{:}\mathbb{N}^k{\to}\mathbb{N}$ der Funktion $f{:}\mathbb{N}^k{\to}\mathbb{N}$ für beliebige $k{\geq}1$ ist die eindeutig bestimmte Funktion h mit der Eigenschaft $h(\hat{x},y) = \Pi_{i=0}^{y} f(\hat{x},i)$ für alle $\hat{x}\in\mathbb{N}^{k-1}$ und $y\in\mathbb{N}$.*

Theorem 2. *Die Generelle Summe $\Sigma f{:}\mathbb{N}^k{\to}\mathbb{N}$ und das Generelle Produkt $\Pi f{:}\mathbb{N}^k{\to}\mathbb{N}$ sind primitiv-rekursiv für alle $k{\geq}1$, wenn f primitiv rekursiv ist.*

Beweis: Sei $h = \Sigma f$. Dann gilt

$$\begin{aligned} h(\hat{x},0) &= \textstyle\sum_{i=0}^{0} f(\hat{x},i) = f(\hat{x},0) \\ h(\hat{x},y{+}1) &= \textstyle\sum_{i=0}^{y+1} f(\hat{x},i) = \sum_{i=0}^{y} f(\hat{x},i)) + f(\hat{x},y{+}1) = h(\hat{x},y) + f(\hat{x},y{+}1) \end{aligned}$$

Damit entsteht h durch primitive Rekursion aus primitiv-rekursiven Funktionen und ist selbst primitiv-rekursiv. Der zugehörige Ausdruck lautet für $k{=}2$

$$\Sigma f = Pr[f{\circ}(pr_1^1, c_0^1),\ add{\circ}(pr_3^3, f{\circ}(pr_1^3, s{\circ}pr_2^3))].$$

Für andere k muß der Ausdruck entsprechend an die veränderte Anzahl der restlichen Argumente angepaßt werden. Für $k{=}1$ lautet der Ausdruck zum Beispiel

$$\Sigma f = Pr[f{\circ}c_0^0,\ add{\circ}(pr_2^2, f{\circ}(s{\circ}pr_1^2))].$$

Der Beweis dafür, daß Πf primitiv-rekursiv ist, verläuft analog. Die Gleichungen lauten nun

$$\begin{aligned} \textstyle\prod_{i=0}^{0} f(\hat{x},i) &= f(\hat{x},0) \\ \textit{und}\ \textstyle\prod_{i=0}^{y+1} f(\hat{x},i) &= \prod_{i=0}^{y} f(\hat{x},i)) * f(\hat{x},y{+}1) \end{aligned}$$

und der primitiv-rekursive Ausdruck für Πf und $k{=}2$ ist

$$\Pi f = Pr[f{\circ}(pr_1^1, c_0^1),\ mul{\circ}(pr_3^3, f{\circ}(pr_1^3, s{\circ}pr_2^3))]$$ □

Eine häufig verwendete einfache Form der primitiven Rekursion ist die *Iteration*, also deren vielfache Anwendung $\underbrace{f(f(\ldots f(x)))}_{k-mal}$ einer Funktion. Sie wird üblicherweise mit $f^k(x)$ bezeichnet.

Definition 12 (Iteration einer Funktion).
Die Iteration $f^*{:}\mathbb{N}^2{\to}\mathbb{N}$ der Funktion $f{:}\mathbb{N}{\to}\mathbb{N}$ ist die eindeutig bestimmte Funktion h mit der Eigenschaft $h(x,k) = f^k(x)$.

So ist z.B. die Addition eine Iteration der Nachfolgerfunktionen und die Subtraktion eine Iteration der Vorgängerfunktion. Die Beschreibung einer Funktion h als Iteration primitiv-rekursiver Funktionen reicht als Nachweis dafür, daß h primitiv-rekursiv ist.

Theorem 3. *Die Iteration $f^*:\mathbb{N}^2\to\mathbb{N}$ einer Funktion $f:\mathbb{N}\to\mathbb{N}$ ist primitiv-rekursiv, wenn f primitiv-rekursiv ist.*

Beweis: Sei $h = f^*$. Dann gilt

$$\begin{array}{lll} h(x,0) & = x & = pr_1^1(x) \\ h(x,y{+}1) & = f^{y+1}(x) = f(h(x,y)) & = f{\circ}pr_3^3(x,y,(h(x,y)) \end{array}$$

h entsteht also durch primitive Rekursion aus primitiv-rekursiven Funktionen und ist daher primitiv-rekursiv. Der zugehörige Ausdruck ist $f^* = Pr[pr_1^1, f{\circ}pr_3^3]$. □

Viele berechenbare Funktionen lassen sich erheblich leichter durch einen Suchprozeß beschreiben als durch eine primitive Rekursion. So ist das kleinste gemeinsame Vielfache zweier Zahlen x und y, wie in Beispiel 1 illustriert, unmittelbar durch eine Suche nach der kleinsten Zahl z, die Vielfaches von x und y ist, charakterisiert, während eine rekursive Beschreibung des kgV schwierig ist. Suchprozesse werden üblicherweise durch einen Startwert und eine Abbruchbedingung charakterisiert, wobei die Abbruchbedingung oft durch das Ergebnis 0 einer Testfunktion dargestellt wird. Damit wird also die kleinste Nullstelle einer berechenbaren Funktion f gesucht, ein Prozeß, den man oft auch als *Minimierung* bezeichnet. Wir werden nun zeigen, daß die Suche nach der kleinsten Nullstelle einer primitiv-rekursiven Funktion wieder eine primitiv-rekursive Funktion ist, sofern eine Obergrenze für die Suche angegeben wird. In diesem Fall spricht man von einer *beschränkten Minimierung*.

Definition 13 (Beschränkte Minimierung und Maximierung).

- *Die beschränkte Minimierung $Mn[f]:\mathbb{N}^k\to\mathbb{N}$ der Funktion $f:\mathbb{N}^k\to\mathbb{N}$ für beliebige $k{\geq}1$ ist die eindeutig bestimmte Funktion h mit der Eigenschaft*
$$h(\hat{x},y) = \begin{cases} min\{z{\leq}y \mid f(\hat{x},z){=}0\} & \textit{falls dies existiert} \\ y{+}1 & \textit{sonst} \end{cases}$$
- *Die beschränkte Maximierung $Max[f]:\mathbb{N}^k\to\mathbb{N}$ der Funktion $f:\mathbb{N}^k\to\mathbb{N}$ für beliebige $k{\geq}1$ ist die eindeutig bestimmte Funktion h mit der Eigenschaft*
$$h(\hat{x},y) = \begin{cases} max\{z{\leq}y \mid f(\hat{x},z){=}0\} & \textit{falls dies existiert} \\ y{+}1 & \textit{sonst} \end{cases}$$

Theorem 4. *Die beschränkte Minimierung $Mn[f]:\mathbb{N}^k\to\mathbb{N}$ und die beschränkte Maximierung $Max[f]:\mathbb{N}^k\to\mathbb{N}$ sind primitiv-rekursiv für alle $k{\geq}1$, wenn dies für f gilt.*

Beweis: Sei $h = Mn[f]$. Dann gilt

$$h(\hat{x},0) = \begin{cases} 0 \text{ falls } f(\hat{x},0) = 0 \\ 1 \text{ sonst} \end{cases}$$

$$h(\hat{x},y{+}1) = \begin{cases} h(\hat{x},y) & \text{falls } h(\hat{x},y){\leq}y \\ y{+}1 & \text{falls } h(\hat{x},y) = y{+}1 \text{ und } f(\hat{x},y{+}1) = 0 \\ y{+}2 & \text{sonst} \end{cases}$$

Damit entsteht h durch primitive Rekursion aus primitiv-rekursiven Funktionen und ist selbst primitiv-rekursiv.

Der zugehörige Ausdruck ist relativ komplex und wegen der notwendigen Projektionen abhängig von der Anzahl der Argumente. Normalerweise würde man ihn nicht explizit konstruieren. Der Vollständigkeit halber werden wir ihn beispielhaft für $k{=}2$ herleiten. Wegen der obigen Funktionsgleichungen wissen wir, daß der Ausdruck für $Mn[f]$ die Gestalt $Pr[f',g]$ hat, wobei

$$f'(\hat{x}) \quad = sign(f(\hat{x},0))$$

$$g(\hat{x},y,z) = \begin{cases} z & \text{falls } z{\leq}y \\ y{+}1 & \text{falls } z = y{+}1 \text{ und } f(\hat{x},y{+}1) = 0 \\ y{+}2 & \text{sonst} \end{cases}$$

Der Teilausdruck für f' ist leicht zu konstruieren: $f' = sign{\circ}f{\circ}(pr^1_1, c^1_0)$.

Der Teilausdruck für g hat die Form $g = Cond[t_1, f_1, Cond[t_2, f_2, g_2]$, besteht also aus einem verschachtelten Konditional. Dabei vergleicht der erste Test das dritte und das zweite Argument und im Erfolgsfall ist das Ergebnis das dritte Argument, d.h. $t_1 = t_{\leq}{\circ}(pr^3_3, pr^3_2)$ und $f_1 = pr^3_3$. Der zweite Test vergleicht das dritte Argument und den Nachfolger des zweiten und testet ob $f(\hat{x}, y{+}1)$ den Wert 0 ergibt. Addiert man die beiden Testteile, so erhält man eine 0 genau dann, wenn beide Tests erfolgreich sind. Damit ergibt sich als Teilausdruck ür den zweiten Test $t_2 = add{\circ}(\, t_{=}{\circ}(pr^3_3, s{\circ}pr^3_2),\ f{\circ}(pr^1_1, s{\circ}pr^3_2)\,)$ und für die beiden Ergebnisfunktionen $f_2 = s{\circ}pr^3_2$ sowie $g_2 = s{\circ}s{\circ}pr^3_2$. Setzt man all diese Teilausdrücke zusammen, so ergibt sich folgender Ausdruck

$$\begin{aligned} Mn[f] = Pr[&sign{\circ}f{\circ}(pr^1_1, c^1_0), \\ &Cond[t_{\leq}{\circ}(pr^1_1, pr^1_1), pr^3_3, \\ &\quad Cond[add{\circ}(\, t_{=}{\circ}(pr^3_3, s{\circ}pr^3_2),\ f{\circ}(pr^1_1, s{\circ}pr^3_2)\,), \\ &\qquad s{\circ}pr^3_2, s{\circ}s{\circ}pr^3_2]\ ]] \end{aligned}$$

Der Beweis dafür, daß $Max[f]$ primitiv-rekursiv ist, könnte analog geführt werden. Es ist aber einfacher, sich die beschränkte Minimierung zunutze zu machen und die Suchreihenfolge umzudrehen. Sucht man nach dem größten $z{\leq}y$ mit $f(\hat{x},z){=}0$, so kann man auch nach dem kleinsten $z'{\leq}y$ suchen, für das die Funktion f' mit $f'(\hat{x},z') = f(\hat{x}, y{-}z')$ den Wert 0 annimt. Im Erfolgsfall ist dann $z = y{-}z'$.

Damit ist $Max[f](\hat{x},y) = \begin{cases} y \dot{-} Mn[f'](\hat{x},y) & \text{falls } y{\leq}Mn[f'](\hat{x},y) \\ y{+}1 & \text{sonst} \end{cases}$

Daraus folgt, daß $Max[f]$ primitiv-rekursiv ist. Die explizite Konstruktion des zugehörigen Ausdrucks sei dem Leser überlassen. □

Die beschränkte Minimierung und Maximierung wird selten in ihrer Grundversion verwendet, da hierbei eine Funktion entsteht, welche die obere Grenze als expliziten Parameter benötigt. In den allermeisten Fällen wird die obere Grenze aus den anderen Argumenten berechnet oder als Konstante vorgegeben. Wir verwendent die Notationen $Mn_g[f]$ bzw. $Mn_y[f]$ für diese beiden Spezialfälle, wobei g eine primitiv-rekursive Funktion und y eine natürliche Zahl ist. Damit ist

$$Mn_g[f](\hat{x}) = \begin{cases} min\{z{\leq}g(\hat{x}) \mid f(\hat{x},z){=}0\} & \text{falls dies existiert} \\ g(\hat{x}){+}1 & \text{sonst} \end{cases}$$

und

$$Mn_y[f](\hat{x}) = \begin{cases} min\{z{\leq}y \mid f(\hat{x},y){=}0\} & \text{falls dies existiert} \\ y{+}1 & \text{sonst} \end{cases}$$

Zu beachten ist dabei, daß die Stelligkeit von $Mn_g[f]$ und $Mn_y[f]$ um 1 geringer ist als die von f bzw. $Mn[f]$, da die Obergrenze der Suche durch den Parameter g bzw. y bestimmt wird. Beide Operationen können leicht mithilfe der allgemeinen beschränkten Minimierung beschrieben werden, denn es ist $Mn_g[f](\hat{x}) = Mn[f](\hat{x}, g(\hat{x}))$ und $Mn_y[f](\hat{x}) = Mn[f](\hat{x}, y)$. Damit ist $Mn_g[f]$ primitiv rekursiv, wenn f und ggf. g primitiv rekursiv sind. $Mn_y[f]$ ist primitiv rekursiv für alle $y \in \mathbb{N}$, wenn f primitiv rekursiv ist. Die zugehörigen primitiv-rekursiven Ausdrücke sind (für k=2)

$Mn_g[f] = Mn[f] \circ (pr^1_1, g)$ und $Mn_y[f] = Mn[f] \circ (pr^1_1, c^1_y)$

Theorem 4 zeigt, daß begrenzte Suchschleifen nicht mächtiger sind als einfache Zählschleifen, da jede beschränkte Minimierung durch einen primitiv-rekursiven Ausdruck ersetzt werden kann. Nichtsdestotrotz ist das "Programmieren" mit Suchschleifen oft erheblich eleganter, wie die folgenden Beispiele zeigen.

Beispiel 14 (Integer-Quadratwurzel)

Die Integer-Quadratwurzelfunktion $sqrt$:$\mathbb{N} \to \mathbb{N}$ ist definiert durch $sqrt(x) = \lfloor \sqrt{x} \rfloor$. Sie bestimmt den ganzzahligen Anteil der Quadratwurzel von n, also die eindeutig bestimmte Zahl $z \in \mathbb{N}$ mit der Eigenschaft $z^2 \leq x < (z+1)^2$.
Ein Lösungsansatz mit primitiver Rekursion benötigt die Funktionsgleichungen

$$\begin{aligned} sqrt(0) &= 0 \\ sqrt(x+1) &= \begin{cases} sqrt(x)+1 & \textit{falls } (sqrt(x)+1)^2 \leq x+1 \\ sqrt(x) & \textit{sonst} \end{cases} \end{aligned}$$

die nun in einen primitiv-rekursiven Ausdruck übersetzt werden müssen. Es ergibt sich
$sqrt = Pr[c^0_0,\ Cond[t_{\leq} \circ (sqr \circ s \circ pr^2_2, s \circ pr^2_1),\ s \circ pr^2_1, pr^2_1]\,]$,
wobei $sqr = mul \circ (pr^1_1, pr^1_1)$ die Quadratfunktion ist.

Ein Lösungsansatz mit begrenzter Suche benötigt nur die Erkenntnis, daß $sqrt(x)$ das größte z mit $z^2 \leq x$ ist, wobei als Obergrenze für die Suche die Zahl x selbst dienen kann. Damit ist die f:$\mathbb{N}^2 \to \mathbb{N}$ mit $f(x,z) = t_{\leq}(z^2, x)$ die Funktion, deren maximale Nullstelle gesucht werden muß: $sqrt = Max_{pr^1_1}[t_{\leq} \circ (sqr \circ pr^2_2, pr^2_1)]$

Dieser Ausdruck ist nicht nur einfacher, sondern auch schneller zu verarbeiten. Für eine effiziente Verarbeitung dürfen wir allerdings nicht den primitiv-rekursiven Ausdruck für die Minimierung einsetzen, sondern benötigen den "echten" Minimierungsoperator, den wir später in Definition 24 auf Seite 25 vorstellen werden.

Beispiel 15 (Kleinstes gemeinsames Vielfaches)

Die Funktion kgV:$\mathbb{N}^2 \to \mathbb{N}$ bestimmt bei Eingabe zweier Zahlen x und y die kleinste Zahl z mit der Eigenschaft, daß z von x und y geteilt wird.

Ein Lösungsansatz mit begrenzter Suche besteht aus nichts anderem als einer Formalisierung dieser Definition. Gesucht wird die kleinste Zahl z, welche folgende vier Bedingungen erfüllt: x teilt z und y teilt z und $x \leq z$ und $y \leq z$. Dabei sollen die letzten beiden Bediingungen verhindern, daß die Zahl 0 als kleinstes gemeinsames Vielfaches akzeptiert wird, wenn eine der beiden Zahlen größer als 0 ist, denn 0 wird von jeder Zahl geteilt. Als Obergrenze für die Suche kann $x * y$ gewählt werden. Da Teilbarkeit und Größenvergleich nach Beispiel 16 bzw. 9 primitiv rekursiv sind, ist damit gezeigt, daß kgV ebenfalls primitiv rekursiv ist.[6]

Für die Konstruktion des primitiv-rekursiven Ausdrucks verwendet man die Tests $t_{divides}$ und $t_{\leq}$ aus Beispiel 16 bzw. 9. $f{:}\mathbb{N}^3{\to}\mathbb{N}$ mit $f(x,y,z) = t_{divides}(x,z) + t_{divides}(y,z) + t_{\leq}(x,z) + t_{\leq}(y,z)$ ist die Funktion, deren minimale Nullstelle gesucht werden muß. Der Ausdruck ist damit strukturell sehr einfach und wird nur durch die erforderlichen Projektionen etwas komplexer.

$$kgV = Mn_{mul}[add{\circ}(add{\circ}(t_{divides}{\circ}(pr^3_1, pr^3_3), t_{divides}{\circ}(pr^2_1, pr^3_3)),\\ add{\circ}(t_{\leq}{\circ}(pr^3_1, pr^3_3), t_{\leq}{\circ}(pr^2_1, pr^3_3)))]$$

Ein Lösungsansatz mit primitiver Rekursion würde erklären müssen, wie $kgV(x, y{+}1))$ nur aus x, y und $kgV(x,y))$ berechnet werden kann. Zwischen $kgV(x, y{+}1))$ und $kgV(x,y))$ besteht jedoch kein unmittelbarer Zusammenhang, so daß dieser Ansatz nur auf Umwegen zum Erfolg führten kann.

Begrenzte Suche, primitive Rekursion und Komposition ermöglichen es, nahezu jeden Mechanismus zu simulieren, der in der Beschreibung arithmetischer Funktionen vorkommt. Die folgende Beispiele stellen einige wichtige primitiv-rekursive Funktionen vor, die zum Teil mit einfachen Kompositionen und zum Teil nur mit Minimierung "programmiert" werden können. Die Beweise und die Konstruktion der entsprechenden primitiv-rekursiven Ausdrücke überlassen wir dem Leser als Übungsaufgabe.

Beispiel 16 (Weitere wichtige primitiv-rekursive Funktionen)
Die folgenden Funktionen sind primitiv rekursiv.

- *Die absolute Differenz* $absdiff{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $absdiff\,(n,m) = |n{-}m|$
- *Das Maximum* $max{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $max(n,m) = \begin{cases} n & \text{falls } n{\geq}m \\ m & \text{sonst} \end{cases}$
- *Das Minimum* $min{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $min(n,m) = \begin{cases} n & \text{falls } n{\leq}m \\ m & \text{sonst} \end{cases}$
- *Die Division* $div{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $div(n,m) = n \div m$
- *Der Divisionsrest* $mod{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $mod(n,m) = n \bmod m$
- *Der Teilbarkeitstest* $t_{divides}{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $t_{divides}(n,m) = \begin{cases} 0 & \text{falls } n \text{ teilt } m \\ 1 & \text{sonst} \end{cases}$
- *Der Primzahltest* $t_{prim}{:}\mathbb{N}{\to}\mathbb{N}$ *mit* $t_{prim}(n) = \begin{cases} 0 & \text{falls } n \text{ Primzahl} \\ 1 & \text{sonst} \end{cases}$
- *Der duale Integer-Logarithmus:* $ld{:}\mathbb{N}{\to}\mathbb{N}$ *mit* $ld(n) = \lfloor \log_2 n \rfloor$
- *Der größte gemeinsamer Teiler* $ggT{:}\mathbb{N}^2{\to}\mathbb{N}$ *mit* $ggT(n,m) = max\{z \mid z \text{ teilt } x \wedge z \text{ teilt } y\}$

[6] Vorverweise auf später als primitiv-rekursiv nachzuweisende Funktionen enthalten prinzipiell die Gefahr eines zirkulären Arguments, da die Funktion kgV aus Beispiel 15 bei der Programmierung der Teilbarkeitsfunktion $t_{divides}$ in Beispiel 16 verwendet werden könnte. In dieser konkreten Situation ist dies aber nicht der Fall. Wir haben die Funktion kgV lediglich vorgezogen, um das Prinzip der beschränkten Minimierung zu illustrieren.

2.3 Berechnungen auf Zahlenpaaren und -listen

In einigen der bisherigen Beispiele und Konstruktionen stellte sich heraus, daß eine unbestimmte Anzahl von Argumenten bei der Konstruktion primitiv-rekursiver Ausdrücke ein Hindernis darstellte und man für jede mögliche Anzahl einen separaten Ausdruck konstruieren mußte. Der wesentliche Unterschied zwischen den einzelnen Ausdrücken war dabei die Stelligkeit und Anzahl der Projektionsfunktionen, die verwendet werden mußten um zum Beispiel auf das letzte und vorletzte Argument zuzugreifen. Die Grundstruktur des Ausdrucks war dagegen immer die gleiche. In späteren Beweisen werden wir zudem noch rekursive Funktionen benötigen, die auf Listen von Zahlen, also einer wechselnden Anzahl von Argumenten arbeiten.

Beide Probleme sind leicht zu lösen, wenn man es schafft, mehrere Argumente durch ein einziges zu codieren, also ein Tupel oder eine Liste natürlicher Zahlen bijektiv und berechenbar durch eine einzige natürliche Zahl zu repräsentieren. Dies würde zudem ermöglichen, Funktionen mit mehreren Eingaben und mehreren Ausgaben auf einheitliche Art zu beschreiben und hierfür einstellige Funktionen zu verwenden.

Wenn man, wie heute üblich, alle Daten eines Programms binär codiert, dann ist es nicht verwunderlich, daß man mehrere Zahlen durch eine einzige Zahl darstellen kann. Man muß hierzu nur die einzelnen Bitketten aneinanderhängen und das Ergebnis als Zahl interpretieren. Hierbei muß man allerdings einige Details berücksichtigen.

Interessanter und mathematisch eleganter als ein Umweg über Bitketten ist aber, die Codierung von Zahlentupeln und -listen auf rein arithmetischer Basis durchzuführen. Die wohl bekannteste Form ist Gödel-Codierung von Zahlenfolgen durch Primzahlpotenzen, bei der die i-te Zahl einer Liste als Potenz der i-ten Primzahl verwendet und alle Primzahlpotenzen aufmultipliziert werden. Diese Codierung, mit der z.B. das Tupel $(4, 2, 1, 1)$ durch die Zahl $2^4 * 3^2 * 5 * 7 = 5040$ repräsentiert wird, ist zwar mathematisch sehr elegant, aber in bezug auf den Berechnungsaufwand relativ komplex.

Daneben gibt es aber auch eine sehr einfache Codierung von Zahlenpaaren, die leicht auf größere Tupel und Listen fortgesetzt werden kann. Die Idee ist dabei, Paaren von Zahlen (x, y) einen Wert zuzuweisen, indem man schrittweise die Wertetabelle der Funktion mit Zahlen auffüllt. Man beginnt am Nullpunkt mit dem Ergebnis 0 und füllt dann, wie in Tabelle 2.3 illustriert, jeweils diagonale Linien mit konstantem $x+y$-Wert der Reihe nach mit immer größer werdenden Zahlen auf. Die so entstehende Funktion weist jedem Zahlenpaare eine eindeutige Nummer zu und wird im folgenden als *Standard-Tupelfunktion* bezeichnet.

⋮	⋮	⋮	⋮	⋮	⋮	⋮	...
5	**20**						...
4	**14**	**19**					...
3	**9**	**13**	**18**				...
2	**5**	**8**	**12**	**17**			...
1	**2**	**4**	**7**	**11**	**16**		...
0	**0**	**1**	**3**	**6**	**10**	**15**	...
y/x	0	1	2	3	4	5	...

Tabelle 1. Ergebnistabelle der Standard-Tupelfunktion

Da jede Diagonale mit dem konstantem $x{+}y$-Wert genau $x{+}y{+}1$ Elemente enthält, läßt sich die Standard-Tupelfunktion auch gut numerisch beschreiben. An der Stelle $x{=}0$ hat sie den Wert $(\sum_{i=1}^{x+y} i)$ und für größere y erhöht sich dieser Wert genau um y. Diese Beobachtung führt zu folgender Definition.

Definition 17 (Standard-Tupelfunktion).
Die Standard-Tupelfunktion $\langle\rangle{:}\mathbb{N}^2{\to}\mathbb{N}$ *ist (in Infix-Notation) definiert durch*

$$\langle x,y\rangle = (\sum_{i=1}^{x+y} i) + y = (x{+}y)(x{+}y{+}1){\div}2 + y$$

Gelegentlich wird die Standard-Tupelfunktion auch mit π bezeichnet, was sich wesentlich leichter aussprechen läßt als $\langle\rangle$. Sie hat eine Reihe wichtiger Eigenschaften.

Theorem 5.

1. $\langle\rangle{:}\mathbb{N}^2{\to}\mathbb{N}$ *ist primitiv-rekursiv*
2. $\langle\rangle{:}\mathbb{N}^2{\to}\mathbb{N}$ *ist bijektiv*
3. *Die Umkehrfunktionen* $\pi_i{:}\mathbb{N}{\to}\mathbb{N}$ *von* $\langle\rangle$*, definiert durch* $\pi_i = pr_i^2 \circ \langle\rangle^{-1}$ *für* $i \in \{1,2\}$ *sind primitiv-rekursiv*

Beweis:

1. $\langle\rangle$ kann durch eine Komposition von Addition, Multiplikation und Division beschrieben werden und ist damit primitiv rekursiv. Der zugehörige primitiv-rekursive Ausdruck ergibt sich unmittelbar aus der obigen Gleichung.
2. Wir müssen zeigen, daß $\langle\rangle$ injektiv und surjektiv ist.

   Um zu zeigen, daß $\langle\rangle$ injektiv ist, nehmen wir an, daß $(x,y){\neq}(x',y')$ ist und zeigen $\langle x,y\rangle{\neq}\langle x',y'\rangle$.
   Liegen (x,y) und (x',y') auf derselben Diagonale, dann gilt $x{+}y = x'{+}y'$, aber $x{\neq}x'$ und $y{\neq}y'$. In diesem Fall ist
   $\langle x,y\rangle = (x{+}y)(x{+}y{+}1){\div}2 + y = (x'{+}y')(x'{+}y'{+}1){\div}2 + y$
   $\neq (x'{+}y')(x'{+}y'{+}1){\div}2 + y' = \langle x',y'\rangle$.
   Liegen (x,y) und (x',y') nicht auf derselben Diagonale, dann ist o.B.d.A. $x{+}y < x'{+}y'$, also $x{+}y{+}1{\leq}x'{+}y'$. Es folgt
   $\langle x',y'\rangle = (x'{+}y')(x'{+}y'{+}1){\div}2 + y' \geq (x{+}y{+}1)(x{+}y{+}2){\div}2 + y'$
   $= (x{+}y{+}1)(x{+}y){\div}2 + x{+}y{+}1 + y'$
   $> (x{+}y{+}1)(x{+}y){\div}2 + y = \langle x,y\rangle$

   Um zu zeigen, daß $\langle\rangle$ surjektiv ist, zeigen wir per Induktion, daß für alle $n \in \mathbb{N}$ ein Paar (x,y) existiert mit $\langle x,y\rangle = n$.
   Für $n = 0$ wählen wir $x = y = 0$, denn es gilt $\langle 0,0\rangle = 0$.
   Sei $\langle x,y\rangle = n$. Wir konstruieren (x',y') mit $\langle x',y'\rangle = n{+}1$.
   - Falls $x = 0$ ist, dann ist
     $n{+}1 = \langle x,y\rangle{+}1 = y(y{+}1){\div}2 + y{+}1 = (y{+}2)(y{+}1){\div}2 + 0 = \langle y{+}1, 0\rangle$
   - Sonst ist $n{+}1 = \langle x,y\rangle{+}1 = (x{+}y)(x{+}y{+}1){\div}2 + y{+}1 = \langle x{-}1, y{+}1\rangle$

   Damit existiert in jedem Fall ein (x',y') mit $\langle x',y'\rangle = n{+}1$.

3. Zur Konstruktion der Umkehrfunktionen bestimmen wir zunächst den $x{+}y$-Wert der Diagonalen, auf der sich der Eingabewert n befindet. Aus der Gleichung für $\langle x,y\rangle$ lassen sich dann zunächst y und daraus x leicht bestimmen.
Es sei $g(n) = min\{z \mid (z{+}1)(z{+}2){\div}2 > n\}$. Dann ist $g{:}\mathbb{N}{\to}\mathbb{N}$ durch eine Minimierung mit Obergrenze n konstruierbar und somit primitiv rekursiv. Per Konstruktion ist $\langle g(n),0\rangle{\leq}n$ und $\langle g(n){+}1,0\rangle{>}n$. Damit beschreibt $g(n)$ den $x{+}y$-Wert der Diagonalen von n und für das Paar (x,y) mit $\langle x,y\rangle = n$ folgt $\pi_2(n) = y = n - g(n)(g(n){+}1){\div}2$ und $\pi_1(n) = x = g(n) - y$.
Damit ist gezeigt, daß beide Umkehrfunktionen primitiv rekursiv sind.

   Der Vollständigkeit halber geben wir zusätzlich die zugehörigen Ausdrücke an
$g = Mn_{pr_1^1}[t_{\leq}{\circ}(s{\circ}pr_1^2, div{\circ}(mul{\circ}(s{\circ}pr_2^2, s{\circ}s{\circ}pr_2^2), c_2^1))]$
$\pi_2 = sub{\circ}(pr_1^1, div{\circ}(mul{\circ}(g, s{\circ}g), c_2^1)$
$\pi_1 = sub{\circ}(g, \pi_2)$.

□

Die Standard-Tupelfunktion $\langle\rangle$ kann iterativ auf Tupel aus $\mathbb{N}^k$ für beliebige k und auf endliche Listen von Zahlen aus $\mathbb{N}^*$ fortgesetzt werden.

Definition 18 (Codierungen von Tupeln und Listen).
Die Standard k-Tupelfunktionen $\langle\rangle^k{:}\mathbb{N}^k{\to}\mathbb{N}$ *sind induktiv definiert durch*

$$\begin{aligned}\langle x\rangle^1 &= x\\ \langle x_1,..,x_{k+1}\rangle^{k+1} &= \langle\langle x_1,..,x_k\rangle^k, x_{k+1}\rangle\end{aligned}$$

Die Listencodierung $\langle\rangle^*{:}\mathbb{N}^*{\to}\mathbb{N}$ *ist definiert durch*

$$\langle x_1..x_k\rangle^* = \langle k, \langle x_1,..,x_k\rangle^k\rangle$$

Gelegentlich werden diese Funktionen mit π^k und π^* bezeichnet und es gilt $\langle\rangle = \langle\rangle^2$. Die meisten Eigenschaften der Standard-Tupelfunktion gelten auch für $\langle\rangle^k$ und $\langle\rangle^*$.

Theorem 6.

1. *Die Funktionen* $\langle\rangle^k{:}\mathbb{N}^k{\to}\mathbb{N}$ *und* $\langle\rangle^*{:}\mathbb{N}^*{\to}\mathbb{N}$ *sind für alle* $k{\in}\mathbb{N}$ *primitiv-rekursiv.*
2. *Die Funktionen* $\langle\rangle^k$ *und* $\langle\rangle^*$ *sind für alle* $k{\in}\mathbb{N}$ *bijektiv.*
3. *Die Umkehrfunktionen* $\pi_i^k{:}\mathbb{N}{\to}\mathbb{N}$ *mit* $\pi_i^k = pr_i^k{\circ}(\langle\rangle^k)^{-1}$ *sind für alle* $k{\in}\mathbb{N}$ *und* $i{\leq}k$ *primitiv-rekursiv.*

Dieses Theorem kann durch Induktion über die Größe der Tupel k bewiesen werden. Eine wichtige Konsequenz ist, daß sich jede mehrstellige Funktion $f{:}\mathbb{N}^k{\to}\mathbb{N}$ durch eine einstellige Funktion simulieren läßt.

Theorem 7. *Zu jeder primitiv-rekursiven Funktion* $f{:}\mathbb{N}^k{\to}\mathbb{N}$ *gibt es eine primitiv-rekursive einstellige Funktion* $f'{:}\mathbb{N}{\to}\mathbb{N}$ *mit der Eigenschaft* $f = f'{\circ}\langle\rangle^k$.

Beweis: Wir wählen $f' = f{\circ}(\pi_1^k,..,\pi_k^k)$. Dann ist f' primitiv rekursiv und es gilt für alle $x_1,..,x_k{\in}\mathbb{N}^k$ $f'{\circ}\langle\rangle^k(x_1,..,x_k) = f{\circ}(\pi_1^k,..,\pi_k^k)\langle x_1,..,x_k\rangle^k = f(x_1,..,x_k)$

□

Die Tupelfunktionen liefern auch eine Möglichkeit, Funktionen mit mehrstelligen Ein- *und Ausgaben* primitiv-rekursiv zu beschreiben. Wir werden von dieser Möglichkeit später gelegentlich Gebrauch machen.

Definition 19 (Mehrstellige primitiv-rekursive Funktionen).

- *Eine Funktion* $f{:}\mathbb{N}^k \to \mathbb{N}^j$ *ist primitiv-rekursiv, wenn die zugehörige einstellige Funktion* f' *mit* $f' \circ \langle\rangle^k = \langle\rangle^j \circ f$ *primitiv-rekursiv ist.*
- *Eine Funktion* $g{:}\mathbb{N}^* \to \mathbb{N}^*$ *ist primitiv-rekursiv, wenn die zugehörige einstellige Funktion* g' *mit* $f' \circ \langle\rangle^* = \langle\rangle^* \circ g$ *primitiv-rekursiv ist.*

Aufgrund dieser Definition können wir alle berechenbare Funktionen auf Zahlen, Tupeln und Listen als einstellige Funktionen ansehen, da die Unterschiede durch die Standard-Tupelfunktionen ausgeglichen werden können. Der Einfachheit halber werden wir berechenbare Funktionen im folgenden öfter durch Gleichungen der Form $f\langle x_1, .., x_k\rangle = \langle y_1, .., y_j\rangle$ beschreiben und dabei sowohl die Funktionsklammern als auch die Stelligkeit der Standard-Tupelfunktionen auslassen, sofern aus dem Kontext klar ist, was gemeint ist.

3 Die Ackermannfunktion

Primitiv-rekursive Funktionen sind eine sehr mächtige Beschreibungsform für berechenbare Funktionen, denn es ist hat sich herausgestellt, daß es ausgesprochen schwierig ist, Funktionen zu identifizieren, die intuitiv berechenbar aber nicht primitiv-rekursiv sind. Daher war man lange Zeit der Ansicht, daß primitiv-rekursive Funktionen als Konzept ausreichen, um den intuitiven Begriff der Berechenbarkeit genau zu charakterisieren. Noch im Jahre 1926 stellte David Hilbert die Vermutung auf, daß alle berechenbaren Funktionen primitiv-rekursiv seien.

Im gleichen Jahr gelang es jedoch seinen Schülern Wilhelm Ackermann und Gabriel Sudan unabhängig voneinander, Funktionen zu konstruieren,[7] die zwar total – also auf allen Eingaben definiert – und offensichtlich berechenbar, jedoch nicht primitiv-rekursiv waren. Die bekanntere der beiden Funktionen ist die 1928 publizierte *Ackermannfunktion* [1], obwohl Sudans Ergebnis allgemeiner war [2].

Später wurden noch weitere Funktionen nach ähnlichem Bildungsprinzip entwickelt, die z.T. auch als Ackermannfunktionen bezeichnet werden.[8] Eine vereinfachte Darstellung der ursprünglichen Ackermannfunktion stammt von der ungarischen Mathematikerin Rózsa Péter [4]. Wir folgen hier ihrer Darstellung.

[7] Interessant ist, daß bisher alle total berechenbaren Funktionen, die nicht primitiv-rekursiv sind, auf konstruierten Beispielen beruhen, die kein echtes natürliches Gegenstück in der Mathematik besitzen. Es besteht daher nach wie vor die Vermutung, daß alle berechenbaren "mathematischen" Funktionen primitiv-rekursiv sein könnten. Dies ist aber eine eher philosophische Fragestellung, da auch künstliche Konstruktionen mit einem gewissen Bekanntheitsgrad nach einiger Zeit in die konventionelle Mathematik Einzug halten.

[8] Funktionen dieser Art werden heute gerne als Benchmarks für Programmiersprachen/Compiler genutzt, um rekursive Prozeduraufrufe zu testen, da die zur Berechnung notwendige Schachtelungstiefe schnell zu einem Stackoverflow führen kann. Dieser Ansatz geht auf Sundblad [5] zurück.

Definition 20 (Ackermann-Funktionen).
Die Ackermann-Funktionen A_n*:*$\mathbb{N}\to\mathbb{N}$ *sind induktiv definiert durch*

$$\begin{aligned} A_0(x) &= \begin{cases} 1 & \text{falls } x{=}0 \\ 2 & \text{falls } x{=}1 \\ x{+}2 & \text{sonst} \end{cases} \\ A_{n+1}(0) &= 1 \\ A_{n+1}(x{+}1) &= A_n(A_{n+1}(x)) \end{aligned}$$

Die große Ackermannfunktion A*:*$\mathbb{N}\to\mathbb{N}$ *ist definiert durch* $A(x) = A_x(x)$.

Mit einem einfachen Induktionsbeweis lässt sich zeigen, dass jede einzelne der Ackermann-Funktionen A_n primitiv rekursiv ist. Die Funktion A_0 ist offensichtlich primitiv rekursiv und A_{n+1} entsteht durch primitive Rekursion aus A_n.

Theorem 8. *Die Ackermannfunktionen* A_n*:*$\mathbb{N}\to\mathbb{N}$ *sind primitiv rekursiv für alle* $n\in\mathbb{N}$.

Damit ist intuitiv leicht einzusehen, daß die große Ackermannfunktion A total ist und berechenbar sein muß. Daß sie selbst nicht primitiv-rekursiv sein kann, liegt an ihrem extremen Wachstumsverhalten, das man schon bei kleinen Eingaben erkennen kann, wie die folgenden Berechnungen zeigen.

$$A_0(x) = x{+}2 \ \ (x{\geq}2)$$
$$A_1(x) = 2x \ \ (x{\geq}1)$$
$$A_2(x) = 2^x$$
$$A_3(x) = \underbrace{2^{(2^{(2^{\cdots^2})})}}_{x-mal}$$

$$A_4(0) = 1$$
$$A_4(1) = 2$$
$$A_4(2) = 2^2 = 4$$
$$A_4(3) = 2^{2^{2^2}} = 65536$$
$$A_4(4) = \underbrace{2^{(2^{(2^{\cdots^2})})}}_{65536-mal}$$
$$A_4(5) = \underbrace{2^{(2^{(2^{\cdots^2})})}}_{A_4(4)-mal}$$

$$A(0) = 1$$
$$A(1) = 2$$
$$A(2) = 2^2 = 4$$
$$A(3) = 2^{2^2} = 16$$
$$A(4) = \underbrace{2^{(2^{(2^{\cdots^2})})}}_{65536-mal}$$
$$A(5) = \underbrace{A_4(A_4(....A_4(1))))}_{65536-mal} = \underbrace{A_4(A_4(....A_4(4))))}_{65533-mal}$$

3.1 Die Ackermannfunktion ist nicht primitiv-rekursiv

Die Ackermannfunktionen A_n sind, wie gezeigt, trotz ihres starken Wachstumsverhaltens immer noch primitiv rekursiv. Das noch erheblich stärkere Wachstums der großen Ackermannfunktion A entsteht dadurch, daß diese Funktion diagonal durch alle Ackermannfunktionen A_n hindurchgeht und somit neben größer werdenden Argumenten auch immer stärker wachsende Funktionen zur Berechnung des Funktionswertes verwendet. Dadurch wird das Wachstumsverhaltens der großen Ackermannfunktion so extrem, daß keine primitiv-rekursive Funktion mithalten kann – *die große Ackermann-Funktion wächst stärker als jede primitiv-rekursive Funktion*.

Ein präziser Beweis dieser Behauptung ist aufwendig, da eine Reihe von Induktionsbeweisen und Lemmata über das Monotonieverhalten der Ackermann-Funktionen benötigt werden. Wir geben daher ein einfaches intuitives Argument.

Eine gutes Klassifikation der Komplexität von Funktionen ist ihre (rekursive) *Schachtelungstiefe*, also die Anzahl der ineinander verschachtelten Rekursionen. Jede primitiv-rekursive Funktion hat eine feste Schachtelungstiefe und wir werden zeigen, daß für die große Ackermannfunktion keine feste Schachtelungstiefe angegeben werden kann.

Definition 21 (Schachtelungstiefe primitiv-rekursiver Funktionen).
Die Schachtelungstiefe einer primitiv-rekursiven Funktion f ist induktiv definiert.

- *Jede Grundfunktion aus $\mathcal{G}$ hat die Schachtelungstiefe* 0.
- *Die Komposition von Funktionen mit maximaler Schachtelungstiefe n hat wiederum die Schachtelungstiefe n.*
- *Die primitive Rekursion über Funktionen der maximalen Schachtelungstiefe n besitzt die hat die Schachtelungstiefe $n+1$.*

Beispiel 22 (Schachtelungstiefe wichtiger Funktionen)

- *Die Addition konstanter Werte ist beschreibbar als $\underbrace{s \circ s \circ .. \circ s}_{k-mal}$.*
  Sie hat die Schachtelungstiefe 0.
- *Addition add, Vorgänger p und Signum-Funktion $sign$ haben Schachtelungstiefe* 1.
- *Multiplikation mul und Substraktion sub haben Schachtelungstiefe* 2.
- *Exponentiation exp und Fakultätsfunktion fak haben Schachtelungstiefe* 3.

Betrachten wir nun das Verhalten der Funktionen A_n, so fällt auf, dass ihre Schachtelungstiefe mit n wächst. Die Schachtelungstiefe von A_0 ist 1, wenn man das Konditional und den zugehörigen Ausdrucks aus dem Beweis von Theorem 1 verwendet, und die Schachtelungstiefe von A_{n+1} ist jeweils um 1 größer als die von A_n.

Da die große Ackermannfunktion A diagonal durch die Ackermannfunktionen hindurchgeht, bedeutet dies, daß zur Berechnung des Wertes $A(x)$ ein Ausdruck der Schachtelungstiefe $x+1$ benötigt wird, also die notwendige Schachtelungstiefe mit der Eingabe wächst. Damit kann keine feste Schachtelungstiefe für die große Ackermannfunktion angegeben werden und deshalb kann A kann nicht primitiv-rekursiv sein.

Theorem 9. *Die große Ackermannfunktion ist nicht primitiv-rekursiv ($A \notin \mathcal{PR}$).*

3.2 Wie kann man die Ackermannfunktion berechnen?

Die klare rekursive Beschreibung der große Ackermannfunktion macht es leicht, einen intuitiven Berechnungsmechanismus für diese Funktion anzugeben. Bei Verwendung moderner funktionaler Programmiersprachen ist es sogar nahezu trivial, ein Programm für die große Ackermannfunktion anzugeben. In ML-artiger Notation würde solch ein Programm wie folgt aussehen.

```
function A'(n, x) =
        if n=0 then if x=0 then 1 else if x=1 then 2 else x+2
               else if x=0 then 1 else A'(n-1,A'(n,x-1))
let A(x) = A'(x,x)
```

Die Berechnung von $A(x)$ führt dann zu einer schrittweisen Abarbeitung der Rekursion, wobei die Verarbeitung von Rekursionsstacks durch den Compiler geregelt wird.

Die große Ackermannfunktion kann auch im Kalkül der rekursiven Funktionen programmiert werden. Hierzu muß man das explizit beschreiben, was in modernen Programmiersprachen der Compiler übernimmt, nämlich die Verarbeitung des Berechnungsstacks der Ackermannfunktion. Wir wollen dies zunächst an einem Berechnungsbeispiel für $A(2)$ erläutern.

Beispiel 23 (Berechnung der Ackermannfunktion)

$$\begin{aligned}
A(2) &= A_2(2) \\
&= A_1(A_2(1)) \\
&= A_1(A_1(A_2(0))) \\
&= A_1(A_1(1)) \\
&= A_1(A_0(A_1(0))) \\
&= A_1(A_0(1)) \\
&= A_1(2) \\
&= A_0(A_1(1)) \\
&= A_0(A_0(A_1(0))) \\
&= A_0(A_0(1)) \\
&= A_0(2) \\
&= 4
\end{aligned}
\qquad
\begin{aligned}
A(2) &\mapsto \langle 2,2\rangle^* \\
&\mapsto \langle 1,2,1\rangle^* \\
&\mapsto \langle 1,1,2,0\rangle^* \\
&\mapsto \langle 1,1,1\rangle^* \\
&\mapsto \langle 1,0,1,0\rangle^* \\
&\mapsto \langle 1,0,1\rangle^* \\
&\mapsto \langle 1,2\rangle^* \\
&\mapsto \langle 0,1,1\rangle^* \\
&\mapsto \langle 0,0,1,0\rangle^* \\
&\mapsto \langle 0,0,1\rangle^* \\
&\mapsto \langle 0,2\rangle^* \\
&\mapsto \langle 4\rangle^*
\end{aligned}$$

Die rekursive Abarbeitung der Definition der Ackermannfunktionen kann vereinfacht durch die Liste der vorkommenden Zahlen beschrieben werden, die wiederum mithilfe der Standard-Tupelfunktion als natürliche Zahl dargestellt wird. Anstelle von $A_2(2)$ schreiben wir also $\langle 2,2\rangle^*$, anstelle von $A_1(A_2(1))$ schreiben wir $\langle 1,2,1\rangle^*$ usw. Damit wird die Berechnung der Ackermannfunktion, wie auf der rechten Seite illustriert, durch eine Folge von Zahlen codiert, wobei die Übergänge zwischen diesen Zahlen der rekursiven Definition der Ackermannfunktionen entsprechen. Die Berechnung endet, wenn der Stack nur noch ein einziges Element, das Ergebnis, enthält.

Wir wollen nun zeigen, daß die Abarbeitungsfunktion δ eines Berechnungsstacks für die Ackermannfunktionen als primitiv-rekursive Funktion beschrieben werden kann. Nach Definition 20 muß diese Funktion folgende Eigenschaften besitzen.

$$\begin{array}{lll}
\delta\langle w\,n\,0\rangle^* & = \langle w\,1\rangle^* & A_n(0) = 1 \\
\delta\langle w\,0\,1\rangle^* & = \langle w\,2\rangle^* & A_0(1) = 2 \\
\delta\langle w\,0\,(x{+}2)\rangle^* & = \langle w\,(x{+}4)\rangle^* & A_0(x{+}2) = x{+}4 \\
\delta\langle w\,(n{+}1)(x{+}1)\rangle^* & = \langle w\,n\,(n{+}1)\,x\rangle^* & A_{n+1}(x{+}1) = A_n(A_{n+1}(x))
\end{array}$$

Dabei steht w für eine beliebige Liste natürlicher Zahlen, während x und n einache Zahlen sind. Durch die Verwendung der Standardtupelfunktion wird der gesamte Stack als natürlicher Zahl dargestellt und daher kann $\delta{:}\mathbb{N}{\to}\mathbb{N}$ eine feste Stelligkeit haben, obwohl die Größe des Stacks sich ständig ändert. Da die Standardtupelfunktion bijektiv ist und primitiv-rekursive Umkehrfunktionen besitzt, läßt sich δ mittels Fallunterscheidung und der Verwendung einfacher primitiv-rekursiver Funktionen programmieren und ist somit selbst primitiv rekursiv.

Die Berechnung der großen Ackermannfunktion kann nun durch eine Iteration der Abarbeitungsfunktion δ beschrieben werden. Es ist $A(x) = \pi_2^2(\delta^k\langle x\,x\rangle^*$, also der eigentliche Inhalt des Stacks $\delta^k\langle x\,x\rangle^*$, wobei k die kleinste Zahl ist für die der Stack nur aus einer einzigen Zahl besteht, also für die $\pi_1^2(\delta^k\langle x\,x\rangle^*){=}1$ gilt.

Da die Iteration primitiv-rekursiver Funktionen primitiv-rekursiv ist, ist auch die Berechnung von $\delta^k\langle x\,x\rangle^*$ primitiv rekursiv. Der einzige Teil der Berechnung, der nicht mit primitiv-rekursiven Mitteln beschrieben werden kann ist die Bestimmung der Zahl k, also der Anzahl der Schritte bis zur Terminierung. Dies muß mit einem Suchprozeß erreicht werden, der aber im Gegensatz zur beschränkten Minimierung ohne vorgegebene Schrittzahlgrenze durchgeführt werden muß. Zwar läßt sich durch Induktion beweisen, daß diese Suche terminieren muß, da in jedem δ-Schritt entweder die erste bearbeitete Zahl im Stack oder die Anzahl der Zahlen kleiner wird, aber diese Erkenntnis reicht nicht aus, um die Schrittzahlgrenze im Voraus angeben zu können. Man muß die Suche also ohne Angabe einer Schrittzahlbegrenzung durchführen und dieser Prozeß ist, wie in Theorem 9 gezeigt, nicht mehr primitiv rekursiv. Diese Erkenntnis führt zu einer Erweiterung des Berechnungsmodells der rekursiver Funktionen, die wir im folgenden Kapitel besprechen werden.

4 μ-rekursive Funktionen

Die Hinzunahme einer unbegrenzten Minimierung stellt eine echte Erweiterung des Kalküls der primitiv-rekursiven Funktionen dar. Eine unbegrenze Suche kann möglicherweise nicht zum Erfolg führen und liefert dann kein Ergebnis. Die hierdurch beschriebene Funktion ist also an dieser Stelle undefiniert und somit eine *partielle Funktion*. Anders als die primitiv-rekursiven Funktionen muß der so entstehende Kalkül die Möglichkeit undefinierter Funktionsergebnisse mit einkalkulieren. Dies bedeutet, daß das Verhalten der in Definition 3 beschriebenen Operatoren Komposition und primitive Rekursion für partielle Funktionen neu erklärt werden muß.

Die Komposition $f\circ(g_1,..,g_n)$:$\mathbb{N}^k\to\mathbb{N}$ ist definiert für eine Eingabe $\hat{x}\in\mathbb{N}^k$, wenn jedes g_i hierauf definiert ist und $(\,g_1(\hat{x}),..,g_n(\hat{x})\,)$ zum Definitionsbereich von f gehört. Ansonsten ist sie undefiniert.

Eine durch primitive Rekursion $Pr[f,g]$ definierte Funktion h:$\mathbb{N}^k\to\mathbb{N}$ ist definiert auf $(\hat{x},0)$, wenn f auf $\hat{x}$ definiert ist und genau dann auf $(\hat{x},y{+}1)$ definiert, wenn $h(\hat{x},y)$ definiert ist und $(\hat{x},y,\,h(\hat{x},y))$ zum Definitionsbereich von g gehört. Man beachte, daß damit $h(\hat{x},y)$ nur dann definiert ist, wenn $h(\hat{x},z)$ für alle $z{<}y$ definiert ist.

Für die unbegrenzte Minimierung führt die Betrachtung partieller Funktionen zu folgender Definition.

Definition 24 (μ-Operator). *Die Minimierung μf:$\mathbb{N}^k\to\mathbb{N}$ einer Funktion f:$\mathbb{N}^{k+1}\to\mathbb{N}$ ist die eindeutig bestimmte Funktion h für die für alle $\hat{x}\in\mathbb{N}^k$ gilt*

$$h(\hat{x}) = \begin{cases} min\{y \mid f(\hat{x},y){=}0\} & \text{falls dies existiert und alle } f(\hat{x},i) \text{ für } i{<}y \text{ definiert sind} \\ \bot & \text{sonst} \end{cases}$$

In der Denkweise imperativer Programmiersprachen entspricht die Minimierung einer unbegrenzte Suchschleife. Wenn die Suche terminiert, dann liefert sie die Anzahl der Durchläufe bis zum Abbruch dieser Schleife. Um $h = \mu f$ zu berechnen, würde man folgende Anweisungen verwenden.

```
z:=0; while f(x1,..,xk,z)≠0 do z:=z+1; h:=z
```

Im Gegensatz hierzu würde die beschränkte Minimierung wie folgt implementiert.

```
z:=0; while f(x1,..,xk,z)≠0 and z≤y do z:=z+1; h:=z
```

Anstelle von $h = \mu f$ schreibt man gelegentlich auch $h(x) = \mu_z[f(x,z)=0]$, wobei an der Stelle von $f(x,z)$ ein beliebiger Ausdruck über den Variablen x und z stehen darf.

Funktionen die aus den Grundfunktionen durch Komposition, primitive Rekursion oder Minimierung entstehen, heißen *μ-rekursiv* oder kurz *rekursiv*.

Definition 25 ($(\mu$-)rekursive Funktionen).
Die Menge $\mathcal{R}$ der (μ-)rekursiven Funktionen ist definiert als $\mathcal{R} = \bigcup_{n=0}^{\infty} \mathcal{R}_n$. Dabei sind die Klassen $\mathcal{PR}_i$ induktiv wie folgt definiert.

$$\begin{aligned}&\mathcal{R}_0 = \mathcal{G} \quad \textit{und}\\ &\mathcal{R}_{i+1} = \mathcal{R}_i \cup \{f \circ (g_1, .., g_n) \mid n \in \mathbb{N}, f, g_1 ... g_n \in \mathcal{R}_i\}\\ &\qquad\qquad \cup \{Pr[f,g] \mid f,g \in \mathcal{R}_i\} \cup \{\mu f \mid f \in \mathcal{R}_i\}\end{aligned}$$

Ähnlich wie die formale Definition der Klasse $\mathcal{PR}$ in Definition 4 ist die obige Definition oft schwer zu handhaben. Meist ist die folgende Charakterisierung handlicher.

Korollar 26

1. *Die Nachfolgerfunktion s ist μ-rekursiv.*
2. *Die Projektionsfunktionen pr_k^n sind μ-rekursiv für alle $n \in \mathbb{N}$ und $k \in \{1..n\}$.*
3. *Die Konstantenfunktionen c_k^n sind μ-rekursiv für alle $n, k \in \mathbb{N}$.*
4. *Die Komposition $f \circ (g_1, .., g_n)$:$\mathbb{N}^k \to \mathbb{N}$ der Funktionen f:$\mathbb{N}^n \to \mathbb{N}$, $g_1, ..g_n$:$\mathbb{N}^k \to \mathbb{N}$ ist μ-rekursiv für alle $n, k \in \mathbb{N}$, wenn $f, g_1 ... g_n$ μ-rekursiv sind.*
5. *Die primitive Rekursion $Pr[f,g]$:$\mathbb{N}^k \to \mathbb{N}$ zweier Funktionen f:$\mathbb{N}^{k-1} \to \mathbb{N}$ und g:$\mathbb{N}^{k+1} \to \mathbb{N}$ ist μ-rekursiv für alle $k \in \mathbb{N}$, wenn f und g μ-rekursiv sind.*
6. *Die Minimierung μf:$\mathbb{N}^k \to \mathbb{N}$ einer Funktion f:$\mathbb{N}^{k+1} \to \mathbb{N}$ ist μ-rekursiv für alle $k \in \mathbb{N}$, wenn f μ-rekursiv ist.*

Durch die Hinzunahme des μ-Operators ändert sich eigentlich nur wenig am Konzept der rekursiven Funktionen. Die Analyse und Konstruktion μ-rekursiver Funktionen ist ähnlich zu derjenigen der primitiv-rekursiven Funktionen, für die bereits eine begrenzte Minimierung zur Verfügung stand. Es entfällt nun die Notwendigkeit, eine Obergrenze für die Suche im Voraus anzugeben. Dafür muß man allerdings mit möglicherweise partiellen Funktionen und undefinierten Funktionswerten umgehen. Wir wollen dies an einigen Beispielen erklären.

Beispiel 27 (Analyse μ-rekursiver Funktionen)

– *Wir wollen den Ausdruck $f_2 = \mu\, c_1^2$ analysieren. Es ist*

$$\begin{aligned}f_2(x) &= \begin{cases} min\{y \mid c_1^2(x,y)=0\} & \textit{falls } y \textit{ existiert und alle } c_1^2(x,i), i<y \textit{ definiert} \\ \bot & \textit{sonst}\end{cases}\\ &= \begin{cases} min\{y \mid 1 = 0\} & \textit{falls dies existiert} \\ \bot & \textit{sonst}\end{cases}\\ &= \bot\end{aligned}$$

Damit ist f_2 die nirgends definierte Funktion.

– *Wir betrachten* $f_3 = \mu\, add$. *Es ist*

$$f_3(x) = \begin{cases} min\{y \mid add(x,y){=}0\} & \textit{falls } y \textit{ existiert } (add \textit{ ist total}) \\ \bot & \textit{sonst} \end{cases}$$

$$= \begin{cases} 0 & \textit{falls } x = 0 \\ \bot & \textit{sonst} \end{cases}$$

Damit ist f_3 *nur für* $x{=}0$ *definiert.*

– *Wir betrachten* $f_4 = \mu\, h$, *wobei* $h(x,y) = \begin{cases} 0 & \textit{falls } x{=}y \\ \bot & \textit{sonst} \end{cases}$. *Dann ist*

$$f_4(x) = \begin{cases} min\{y \mid h(x,y){=}0\} & \textit{falls } y \textit{ existiert und alle } h(x,i), i{<}y \textit{ definiert} \\ \bot & \textit{sonst} \end{cases}$$

$$= \begin{cases} 0 & \textit{falls } x = 0 \\ \bot & \textit{sonst} \end{cases}$$

Hier gilt zwar $h(x,y){=}0$ *für* $x{=}y$, *aber* $h(x,y)$ *ist für* $0{<}y{<}x$ *undefiniert.*

μ-rekursive Funktionen haben gegenüber den primitiv-rekursiven Funktionen den Nachteil, daß sie möglicherweise partiell, also nicht für alle Eingaben definiert sind. Primitiv-rekursive Funktionen sind dagegen *total*, terminieren also immer. Eine wichtige Unterklasse der μ-rekursiven Funktionen sind daher die Funktione, die total, aber nicht notwendigerweise primitiv rekursiv sind.

Definition 28 (total-rekursive Funktionen).
Die Menge $\mathcal{TR}$ *der total-rekursiven Funktionen ist definiert als*

$$\mathcal{TR} = \{f \in \mathcal{R} \mid f \textit{ total}\}.$$

Es ist leicht zu sehen, daß $\mathcal{PR} \subseteq \mathcal{TR} \subseteq \mathcal{R}$ gilt, da alle primitiv-rekursiven Operatoren und Grundfunktionen auch bei der Definition der μ-rekursiven Funktionen verwendet werden und alle primitiv-rekursiven Funktionen total sind. Daß die drei Klassen nicht gleich sind, kann man an zwei einfachen Beispielen sehen.

– $\mathcal{PR} \neq \mathcal{TR}$, da die große Ackermannfunktion A total rekursiv, aber nicht primitiv-rekursiv ist.
– $\mathcal{TR} \neq \mathcal{R}$, da es rekursive Funktionen wie z.B. $f_3 = \mu\, add$ gibt, die nicht total sind.

Damit ist $\mathcal{PR}$ eine echte Unterklasse von $\mathcal{TR}$ und dies eine echte Unterklasse von $\mathcal{R}$.

Die Klasse der total-rekursiven Funktionen ist die Klasse, die aus Anwendersicht die interessanteste wäre, da sie alle berechenbare Funktionen enthält, aber keine nicht-terminierenden Berechnungen berücksichtigen muß. Leider ist es nicht möglich, einen Kalkül oder eine Programmiersprache zu konstruieren, die genau die Klasse $\mathcal{TR}$ erzeugt, denn die Menge der Programme für Funktionen aus $\mathcal{TR}$ ist nicht aufzählbar. Dies werden wir in einer der späteren Abhandlungen zeigen.

Rekursive Funktionen sind ein extrem mächtiger Mechanismus zur Beschreibung berechenbarer Funktionen. Es hat sich herausgestellt, daß sie genauso mächtig sind wie die Turing-berechenbaren Funktionen. Ein formaler Beweis für diese Behauptung ist aufwendig, da für jedes Modell eine Simulation durch das andere angegeben werden muß. Wir geben daher (vorläufig) nur eine kurze Skizze des Beweises

Um zu zeigen, daß rekursive Funktionen Turing-berechenbar sind ($\mathcal{R} \subseteq \mathcal{T}$), muß man zeigen, daß die rekursiven Grundfunktionen und Operationen Turing-berechenbar sind. Turingmaschinenprogramme für die Grundfunktionen – Nachfolger, Projektion und Konstantenfunktionen sind leicht zu konstruieren. Auch haben wir gezeigt, daß es leicht ist, Komposition, Primitive Rekursion und μ-Operator durch Anweisungssequenzen in imperativen Programmiersprachen zu repräsentieren. Da alle Computerprogramme auf Turing-Maschinen simuliert werden können, ist es möglich Turingmaschinen für die rekursiven Operationen zu konstruieren. Damit kann man für jede rekursive Funktion f eine Turingmaschine konstruieren, indem man die Teilmaschinen entsprechend des rekursiven Ausdrucks für f zusammensetzt.

Um zu zeigen, daß Turing-berechenbare Funktionen rekursiv sind ($\mathcal{T} \subseteq \mathcal{R}$), geht man ähnlich vor wie bei der Berechnung der Ackermannfunktion durch rekursive Funktionen in Abschnitt 3.2. Wir codieren Konfigurationen, also Tupel von Worten als Zahlentupel, indem wir jedes Symbol durch seine Nummer im Alphabet repräsentieren und anschließend die Zahlenliste mit der Standard-Tupelfunktion in eine Zahl umwandeln. Anschließend simulieren wir die Konfigurationsübergänge, die sich aus dem Progamm der Turingmaschine ergeben durch eine primitiv-rekursive Überführungsfunktion δ. Diese wird dann iteriert bis eine Endkonfiguration erreicht ist. Hierzu verwenden wir eine primitiv-rekursive Testfunktion f, welche identifizert ob δ^k eine Endkonfigurationen liefert und die unbeschränkte Minimierung für die Suche nach der ersten Nullstelle von f. Wenn diese Suche terminiert, dann liefert die Endkonfiguration das Ergebnis der Berechnung.

Theorem 10. *$\mathcal{R} = \mathcal{T}$, d.h. die Klasse der rekursiven Funktionen ist identisch mit der Menge der Turing-berechenbaren Funktionen.*

Theorem 10 und sein Beweis haben zwei wichtige Aspekte. Zum einen ist gezeigt, daß beide Modelle – Turingmaschine und rekursive Funktionen – sich gegenseitig simulieren können. Deswegen ist es möglich, in Beweisen μ-rekursive Funktionen als "Unterprogramme" von Turingmaschinen einzusetzen, wenn dies einfacher ist als die direkte Konstruktion. Es reicht ja zu wissen, daß eine Umwandlung möglich ist, ohne daß man sie explizit ausführen muß. Zum anderen wurde im Beweis von $\mathcal{T} \subseteq \mathcal{R}$ nur eine einzige Minimierung eingesetzt, um die Anzahl der Schritte bis zur Terminierung zu bestimmen, während alle anderen Komponenten primitiv rekursiv sind.

Dies bedeutet, daß rekursive Funktionen im Prinzip nicht mehr als eine einzige Minimierung benötigen und ansonsten mit primitiv-rekursiven Konstrukten auskommen können. In anderen Worten: jede berechenbare Funktion kann mit einer einzigen unbegrenzten Schleife programmiert werden. Diese Erkenntnis ist unter dem Namen *Kleene Normalform Theorem* bekannt geworden.

Theorem 11 (Kleene Normalform Theorem). *Für jede berechenbare Funktion h kann man primitiv-rekursive Funktionen f und g konstruieren, so daß $h(x) = g(x, \mu f(x))$.*

Beweis: Wir betrachten die Simulation der Turingmaschine für h durch μ-rekursive Funktionen. Die Funktion g mit $g(x, k)$ berechnet die Iteration $\delta^k(x)$ der Konfigurationsübergänge. f ist die Funktion, welche die Terminierung von $\delta^k(x)$ charakterisiert und damit berechnet μf berechnet die Anzahl der Schritte bis zur Terminierung. Also liefert $g(x, \mu f(x))$ das Ergebnis zum Zeit der Terminierung. □

Literatur

1. Wilhelm Ackermann. Zum hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99, 1928. 21
2. Christian Calude and Solomon Marcus. The first example of a recursive function which is not primitive recursive. *Historia Mathematica*, 6:380–384, 1979. 21
3. Richard Dedekind. *Was sind und was sollen die Zahlen?* 1888. 1
4. Rózsa Péter. Die beschränkt-rekursiven funktionen und die ackermannsche Majorisierungsmethode. *Publicationes Mathematicae Debrecen*, 4:362–375, 1956. 21
5. Yngve Sundblad. The ackermann function. a theoretical, computational, and formula manipulative study. *BIT - numerical mathematics*, 11:107–119, 1971. 21

Stand 19. April 2013