Grundbegriffe der mathematischen Logik

Hans Adler
Kurt Gödel Research Center
Universität Wien

Sommersemester 2012

Inhaltsverzeichnis

Kapitel 1

Primitive Rekursivität

Gegeben ein beliebiges Polynom $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ mit ganzzahligen Koeffizienten $a_0, a_1, \ldots, a_{n-1} \in \mathbb{Z}$, und gesucht ist eine ganze Zahl $x \in \mathbb{Z}$, so dass $p(x) = 0$ ist. Wenn nicht alle Koeffizienten 0 sind, dann hat diese Aufgabe höchstens n Lösungen. Man kann leicht eine obere Schranke für den Betrag einer Lösung des Problems angeben. Damit verbleiben nur noch endlich viele potentielle Lösungen, die man alle ausprobieren kann.

Man sollte meinen, dass es so ähnlich auch für Polynome in beliebig vielen Variablen geht. Hilberts zehntes Problem (1900)[1] besteht darin, einen Algorithmus zu finden, mit dem man für jedes solche Polynom herausfinden kann, ob es eine ganzzahlige Lösung gibt. Dieses schwerere Problem wurde nie gelöst. Stattdessen wurde 1970 gezeigt, dass es unlösbar ist.

Wie kann man zeigen, dass ein mathematisches Problem grundsätzlich nicht lösbar ist, wenn doch jede einzelne Instanz offensichtlich eine Lösung hat? Denn jedes einzelne Polynom $p(x, y, z, \ldots)$ hat entweder eine Nullstelle oder nicht. Daher hat es wenig Sinn, die Frage auf ein einzelnes Polynom einzuschränken.[2] Erst durch die Forderung nach einem gemeinsamen Algorithmus, der bei allen Eingaben das richtige Ergebnis liefert, wird das Problem schwierig.

Wir werden in dieser Vorlesung nicht zeigen, dass Hilberts zehntes Problem unlösbar ist. Aber wir werden (in einem späteren Kapitel) zeigen, wie man es so formalisieren konnte, dass seine Lösbarkeit überhaupt zu einer mathematischen Frage wurde, die man auch mit Nein beantworten kann. Wir werden uns dabei auf die natürlichen Zahlen beschränken.[3] Wir werden uns außerdem ein anderes Problem (das Halteproblem) anschauen, von dem wir relativ leicht zeigen können, dass es nicht algorithmisch lösbar ist. Zuvor beschäftigen wir uns aber in diesem Kapitel mit einem eng verwandten Thema, der *primitiven* Rekursivität.

[1] David Hilbert (1862–1943) formulierte im Jahr 1900 auf dem Internationalen Mathematiker-Kongress eine einflussreiche Liste von 23 mathematischen Problemen.

[2] Für jedes einzelne Polynom gibt es natürlich einen Algorithmus, der die jeweilige Instanz des Problems löst: Falls das Polynom eine Nullstelle hat, können wir einen Algorithmus nehmen, der unabhängig von der Eingabe immer *ja* sagt. Falls es keine Nullstelle hat, können wir einen Algorithmus nehmen, der immer *nein* sagt. Im Einzelfall kann es zwar sein, dass wir nicht wissen, welcher der beiden Algorithmen der richtige ist, aber wir wissen immer, dass einer der beiden richtig sein muss.

[3] In dieser Vorlesung gilt, wie in der Logik allgemein üblich, die Null immer als natürliche Zahl. Es ist also $\mathbb{N} = \{0, 1, 2, \ldots\}$.

1.1 Primitiv rekursive und LOOP-berechenbare Funktionen

Primitiv rekursive Funktionen

Bei der folgenden Definition handelt es sich um das erste Beispiel einer *induktiven Definition* in dieser Vorlesung. Wir werden noch einige weitere Beispiele sehen. Induktive Definitionen sind in der Logik sehr beliebt, weil sie induktive Beweise erlauben, eine Verallgemeinerung der vollständigen Induktion für natürliche Zahlen.

induktive Definition

primitiv rekursive Funktion

Definition 1.1 *Die Menge der* primitiv rekursiven Funktionen *ist die kleinste Menge* $F \subseteq \bigcup_{k\in\mathbb{N}} \mathbb{N}^{(\mathbb{N}^k)} = \bigcup_{k\in\mathbb{N}} \{f\colon \mathbb{N}^k \to \mathbb{N}\}$*, welche die folgenden Bedingungen erfüllt:*

- *F enthält die konstante Nullfunktion $0\colon \mathbb{N}^0 \to \mathbb{N}$.*[4]
- *F enthält die durch $S(n) = n + 1$ definierte Nachfolgerfunktion $S\colon \mathbb{N} \to \mathbb{N}$.*

Nachfolgerfunktion S

- *F enthält für alle $i, k \in \mathbb{N}$ mit $i \leq k$ die durch $\pi_i^k(x_1, \ldots, x_k) = x_i$ gegebene Projektionsfunktion $\pi_i^k\colon \mathbb{N}^k \to \mathbb{N}$.*
- *F ist abgeschlossen unter Zusammensetzung. Das heißt, wenn (für $\ell \in \mathbb{N}$) die Funktionen $f\colon \mathbb{N}^\ell \to \mathbb{N}$ und $g_1, \ldots, g_\ell\colon \mathbb{N}^k \to \mathbb{N}$ alle in F liegen, dann liegt auch die durch*

$$h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_\ell(x_1, \ldots, x_k))$$

  gegebene Funktion $h\colon \mathbb{N}^k \to \mathbb{N}$ in F.[5]

- *F ist abgeschlossen unter* primitiver Rekursion*. Das heißt, wenn die Funktionen $f\colon \mathbb{N}^k \to \mathbb{N}$ und $g\colon \mathbb{N}^{k+2} \to \mathbb{N}$ in F liegen, dann liegt auch die durch*

primitive Rekursion

$$\begin{aligned} h(\bar{x}, 0) &= f(\bar{x}) \\ h(\bar{x}, y+1) &= g(\bar{x}, y, h(\bar{x}, y)) \end{aligned}$$

  gegebene Funktion $h\colon \mathbb{N}^{k+1} \to \mathbb{N}$ in F.

Lemma 1.2 *Die primitiv rekursiven Funktionen sind wohldefiniert. Das heißt, es gibt tatsächlich eine kleinste Menge, welche die genannten Bedingungen erfüllt.*

Beweis Sei $\mathcal{F} = \left\{F \subseteq \bigcup_{k\in\mathbb{N}} \mathbb{N}^{(\mathbb{N}^k)} \;\middle|\; F \text{ erfüllt die genannten Bedingungen}\right\}$, und sei $F_0 = \bigcap \mathcal{F}$. Wie man leicht überprüft, erfüllt auch F_0 die genannten Bedingungen. Daher ist F_0 ein Element von $\mathcal{F}$, und zwar zwangsweise das kleinste. (F_0 ist also die Menge der primitiv rekursiven Funktionen.) ∎

Proposition 1.3 *Die folgenden Funktionen sind primitiv rekursiv:*

- $f_1\colon \mathbb{N}^2 \to \mathbb{N}$, $f_1(x, y) = x + y$.
- $f_2\colon \mathbb{N}^2 \to \mathbb{N}$, $f_2(x, y) = x \cdot y$.
- $f_3\colon \mathbb{N}^2 \to \mathbb{N}$, $f_3(x, y) = x^y$.
- $f_4\colon \mathbb{N} \to \mathbb{N}$, $f_4(x) = x! = 1 \cdot 2 \cdot \ldots \cdot x$.

Fakultätsfunktion

Beweis f_1 kann man durch primitive Rekursion wie folgt definieren:

$$\begin{aligned} f_1(x, 0) &= x \\ f_1(x, y+1) &= S(f_1(x, y)) \end{aligned}$$

[4] Zur Erinnerung: Für jede Menge X gilt $X^0 = \{()\}$, wenn wir $()$ für das unabhängig von X eindeutig bestimmte 0-Tupel schreiben. Eine nullstellige Funktion $f\colon \mathbb{N}^0 \to \mathbb{N}$ ist daher im Wesentlichen dasselbe wie eine Konstante, in diesem Fall die Konstante $0 \in \mathbb{N}$.

[5] Man könnte das auch kürzer so schreiben: Wenn $f\colon \mathbb{N}^\ell \to \mathbb{N}$ in F liegt und $\bar{g}\colon \mathbb{N}^k \to \mathbb{N}^\ell$ komponentenweise in F liegt (d.h. $\bar{f} = (f_1, \ldots, f_\ell)$ und jedes $f_i \in F$), dann liegt auch $h = f \circ \bar{g}\colon \mathbb{N}^k \to \mathbb{N}$ in F.

Danach können wir f_2 mit Hilfe von f_1 und schließlich auch f_3 mit Hilfe von f_2 auf dieselbe Weise definieren:

$$\begin{aligned} f_2(x,0) &= 0 & f_3(x,0) &= 1 \\ f_2(x,y+1) &= f_1(f_2(x,y),x) & f_3(x,y+1) &= f_2(f_3(x,y),x). \end{aligned}$$

Zuletzt noch f_4:

$$\begin{aligned} f_4(0) &= 1 \\ f_4(y+1) &= f_2(f_4(y),S(y)). \end{aligned}$$

Die letzte Zeile benutzt natürlich zusätzlich noch die Tatsache, dass auch die Zusammensetzung $f_2(x,S(y))$ primitiv rekursiv ist. ∎

LOOP-Programme

LOOP-Programm

Wir definieren informell eine primitive Programmiersprache für das Rechnen mit natürlichen Zahlen. Die Programme in dieser Sprache nennen wir *LOOP-Programme*. Der Einfachheit halber hat diese Sprache eine bequeme Eigenschaft, die vielen in der Praxis benutzten Programmiersprachen fehlt: Die durch ihre Variablen bezeichneten Register können beliebig große (natürliche) Zahlen aufnehmen.

Jede Variable der Sprache hat einen Namen, der aus lateinischen Groß- und Kleinbuchstaben, Ziffern sowie dem Zeichen `_` bestehen darf. Er muss mit einem Kleinbuchstaben beginnen. Beispiele von gültigen Variablennamen: `x`, `a`, `abcdeXYZ`, `x_1`. Für das Rechnen und Kopieren von Zahlen zwischen den Registern, die durch die Variablen[namen] bezeichnet werden, haben wir die folgenden Befehle:

- `y = Zero()` — `Zero`
- `y = Val(x)` — `Val`
- `y = Inc(x)` — `Inc`
- `y = Dec(x)` — `Dec`

Der erste Befehl schreibt die Zahl 0 ins Register `y`. Der zweite Befehl kopiert die Zahl, die sich im Register `x` befindet, in das Register `y`. Der dritte Befehl nimmt den Wert aus Register `x`, erhöht diese Zahl um 1, und schreibt das Ergebnis in Register `y`. Der vierte Befehl schließlich nimmt den Wert aus Register `x`, verringert diese Zahl um 1, und schreibt das Ergebnis in Register `y`. (Falls der Wert bereits 0 ist, bleibt er 0.) An Stelle von `x` und `y` kann man beliebige Variablennamen schreiben. Das folgende Programm nimmt den Inhalt des Registers `input_1`, addiert 3 dazu, und schreibt das Ergebnis ins Register `output`. Dabei wird nur das Register `output` verändert, nicht aber das Register `input_1`.

```
output = Inc(input_1)
output = Inc(output)
output = Inc(output)
```

`loop ... times` Schleife

Dazu hat die Sprache der LOOP-Programme noch die folgende Kontrollstruktur, die eine primitive Form der FOR-Schleife darstellt, wie sie in vielen Programmiersprachen vorkommt.

```
loop n times {
  ...
}
```

Hierbei ist `n` wieder nur ein Beispiel und kann durch einen beliebigen anderen Variablennamen ersetzt werden. Die drei Punkte stehen für ein beliebiges LOOP-Programm, das wir in diesem Kontext kurz als den *Schleifenkörper* bezeichnen. Zum Beispiel schreibt das folgende LOOP-Programm die Summe der Werte von `x` und `y` ins Register `z`.

Schleifenkörper

```
z = Val(x)
loop y times {
  z = Inc(z)
}
```

Die Schleifen machen eigentlich den `Zero`-Befehl überflüssig, denn man kann `y = Zero()` durch das folgende Programm ersetzen.

```
loop y times {
  y = Dec(y)
}
```

Man kann aber andererseits auch den `Dec`-Befehl als überflüssig ansehen.

Die Anzahl der Durchgänge durch den Schleifenkörper wird noch vor dem ersten Durchgang endgültig festgelegt. Obwohl sich `y` *innerhalb* der LOOP-Schleife noch ändert, hat das keinen Einfluss mehr darauf, wie oft die LOOP-Schleife durchlaufen wird. (Das ist letztlich auch der Grund, warum LOOP-Programme garantiert immer anhalten, siehe Übungsaufgabe 1.4.) Es folgt aus der bisherigen Beschreibung, dass LOOP-Strukturen auch ineinander verschachtelt werden können wie im folgenden Beispiel.

```
z = Zero()
loop x times {
  loop y times {
    z = Inc(z)
  }
}
```

Wenn eine Schleifenanweisung sich innerhalb einer anderen Schleife befindet und deshalb mehrfach ausgeführt wird, dann wird jedesmal neu am Anfang festgelegt, wie oft ihr Schleifenkörper durchlaufen wird. Beispielsweise wird die innere Schleife im folgenden Programm zuerst einmal durchlaufen, dann zweimal, dann dreimal, bis hin zu `x`-mal.

```
y = Zero()
z = Zero()
loop x times {
  z = Inc(z)
  loop z times {
    y = Inc(y)
  }
}
```

Primitiv rekursive Funktionen sind LOOP-berechenbar

LOOP-berechenbare Funktion

Definition 1.4 *Sei $k \in \mathbb{N}$. Eine Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ heißt* LOOP-berechenbar*, falls es es ein LOOP-Programm gibt, welches die Funktion im folgenden Sinne berechnet. Falls das Programm zu Beginn in den Registern* `input_1`*,* `input_2`*, ...,* `input_`k *die Zahlen $x_1, x_2, \ldots, x_k$ stehen hat sowie in allen anderen Registern die Zahl 0, dann steht am Ende im Register* `output` *die Zahl $f(x_1, x_2, \ldots, x_k)$.*

Satz 1.5 *Jede primitiv rekursive Funktion lässt sich durch ein LOOP-Programm berechnen.*

induktiver Beweis

Hier nun unser erstes Beispiel für einen *induktiven Beweis*. Das Grundprinzip ist ganz einfach: Wenn X die kleinste Menge mit der Eigenschaft E ist und wir wollen zeigen, dass $X \subseteq Y$ ist, dann genügt es, zu beweisen, dass auch Y die Eigenschaft E hat. Induktive Beweise verallgemeinern die vollständige Induktion über die natürlichen Zahlen. Um das einzusehen: Nennen wir eine Teilmenge $X \subseteq \mathbb{Q}$ induktiv, falls gilt: (1) $0 \in X$ und (2) $n \in X \Rightarrow n + 1 \in X$. Dann ist $\mathbb{N}$ bekanntlich die kleinste induktive Menge. Um zu zeigen, dass $\mathbb{N} \subseteq Y$ ist, genügt es zu zeigen, dass Y ebenfalls induktiv ist, d.h., dass $0 \in Y$ und $n \in Y \Rightarrow n + 1 \in Y$ gilt.

Beweis [von Satz 1.5] Es genügt, zu zeigen, dass die Menge der durch LOOP-Programme berechenbaren Funktionen die Bedingungen in Definition 1.1 erfüllt. Ein LOOP-Programm, das die Nullfunktion berechnet:

```
output = Zero()
```

Ein LOOP-Programm, das die Nachfolgerfunktion berechnet:

```
output = Inc(input_1)
```

Ein LOOP-Programm, das für alle $k \geq 3$ die Projektionsfunktion π_3^k berechnet:

```
output = Val(input_3)
```

Um zu zeigen, dass die LOOP-berechenbaren Funktionen unter Zusammensetzung abgeschlossen sind, muss man sich überlegen, wie man LOOP-Programme zusammensetzen kann. Jede LOOP-berechenbare Funktion ist auch durch ein LOOP-Programm berechenbar, das die Inhalte der Register `input_1`, `input_2` etc. nicht ändert und in dem die Register `output_1`, `output2`, ... nicht vorkommen. Gegeben $f\colon \mathbb{N}^\ell \to \mathbb{N}$ und $g_1, \ldots, g_\ell\colon \mathbb{N}^k \to \mathbb{N}$, seien $P_1, \ldots, P_\ell$ LOOP-Programme, die $g_1, \ldots, g_\ell$ berechnen und die genannten zusätzlichen Bedingungen erfüllen. Dann erhält man ein LOOP-Programm, das die durch $h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_\ell(x_1, \ldots, x_k))$ gegebene Funktion berechnet, wie folgt:

```
...
output_1 = Val(output)
...
output_2 = Val(output)
...
output_3 = Val(output)
...
input_1 = Val(output_1)
input_2 = Val(output_2)
input_3 = Val(output_3)
...
```

Das Programm beginnt mit P_1, gefolgt von dem Befehl `output_1 = Val(output)`. Danach folgt P_2 und `output_2 = Val(output)`, usw. Dann werden mit `input_1 = Val(output_1)` usw. die Eingabewerte überschrieben, bevor zum Abschluss ein beliebiges LOOP-Programm kommt, das f berechnet.

Beim Beweis, dass die LOOP-berechenbaren Funktionen auch unter primitiver Rekursion abgeschlossen sind, spielt eine Schleife eine wesentliche Rolle:

```
...
input_(k+2) = Val(output)
y = Zero()
loop input_(k+1) times {
  input_(k+1) = Val(y)
  ...
  y = Inc(y)
  input_(k+2) = Val(output)
}
```

Wir beginnen mit einem LOOP-Programm (angedeutet durch drei Punkte), das f berechnet, ohne die Inhalte der Register `input_1`, `input_2` etc. zu verändern. Später kommt eine Schleife, innerhalb derer wir die Variable `y` von 0 bis `input_`$(k+1)-1$ hochzählen und in jedem Schritt das Zwischenergebnis $h(\bar{x}, y+1)$ mittels eines LOOP-Programms, das g berechnet, aus $\bar{x}$, dem aktuellen Wert der Zählervariablen `y` und dem letzten Zwischenergebnis berechnen. (Dieses wiederum durch drei Punkte angedeutete LOOP-Programm darf die Inhalte der Register `input_1`, `input_2` etc. und `y` nicht verändern.) ∎

Unsere LOOP-Programmiersprache wurde, von unwichtigen Details abgesehen, 1967 von Meyer und Ritchie[6] definiert, um genau die primitiv rekursiven Funktionen zu beschreiben. Im nächsten Abschnitt werden wir dann auch beweisen, dass ihnen das gelungen ist.

`if ... then`

Zum Abschluss dieses Abschnitts schauen wir uns noch eine Spracherweiterung an, die die Ausdruckskraft von LOOP-Programmen nicht erhöht, aber das Programmieren wesentlich erleichtert. Analog zu **`loop ... times`** könnten wir noch die folgende Art von Anweisung erlauben:

```
if x then {
  ...
}
```

Der Programmteil, für den die drei Punkte stehen, wird übersprungen, falls `x` den Wert Null hat, sonst aber genau einmal ausgeführt (und nicht `x` mal).

[6] Albert R. Meyer (geboren 1941) ist ein Informatiker, der grundlegende Arbeit in der Komplexitätstheorie geleistet hat. Dennis M. Ritchie (1941–2011) ist vor allem bekannt als der Entwickler der Programmiersprache C und einer der beiden ursprünglichen Entwickler des Betriebssystems UNIX.

Jedes LOOP-Programm in diesem erweiterten Sinne können wir mit dem folgenden Trick in ein normales LOOP-Programm übersetzen:

```
z = Zero()
loop x times {
  y = Inc(z)
}
loop y times {
  ...
}
```

Dabei müssen wir natürlich statt `y` und `z` Variablennamen verwenden, die anderswo im Programm nicht vorkommen.

Übungsaufgaben

Die Übungsaufgaben im Skript sollen Ihnen unabhängig von den offiziellen Übungen helfen, sich mit dem Stoff auseinanderzusetzen. Der Schwierigkeitsgrad variiert stark.

Übungsaufgabe 1.1 *Ein* Hüllenoperator *auf einer Menge X ist eine Abbildung $H\colon \mathcal{P}(X) \to \mathcal{P}(X)$, welche die folgenden Bedingungen erfüllt:*[7]

(extensiv) $H(A) \supseteq A$ *für alle* $A \subseteq X$*;*

(monoton) $H(A) \subseteq H(B)$ *für alle* $A \subseteq B \subseteq X$*;*

(idempotent) $H(H(A)) = H(A)$ *für alle* $A \subseteq X$*.*

Eine Menge $A \subseteq X$ heißt abgeschlossen unter einer Abbildung $H\colon \mathcal{P}(X) \to \mathcal{P}(X)$, falls $H(A) = A$ gilt.

Zeigen Sie, dass es zu jeder extensiven, monotonen Abbildung $G\colon \mathcal{P}(X) \to \mathcal{P}(X)$ einen Hüllenoperator auf X gibt, der dieselben abgeschlossenen Mengen hat. (Hinweis: Iterieren Sie G unendlich oft, um die Idempotenz zu erzwingen.)

Sei $X = \bigcup_{k\in\mathbb{N}}\{f\colon \mathbb{N}^k \to \mathbb{N}\}$. Geben Sie einen Hüllenoperator H auf X an, so dass die primitiv rekursiven Funktionen gerade die kleinste für H abgeschlossene Menge bilden, also von der Form $H(\emptyset)$ sind.

Übungsaufgabe 1.2 *Alle Polynome $p\colon \mathbb{N}^k \to \mathbb{N}$ mit Koeffizienten in den natürlichen Zahlen sind primitiv rekursiv.*

Übungsaufgabe 1.3 *Die folgenden Funktionen sind LOOP-berechenbar:* Monusfunktion

- $f\colon \mathbb{N} \to \mathbb{N}$, $f(x) = x \dot{-} 1 = \max(x - 1, 0)$.
- $f\colon \mathbb{N}^2 \to \mathbb{N}$, $f(x, y) = x \dot{-} y = \max(x - y, 0)$.

(Die Operation $\dot{-}$, eine Variante von Minus, wird manchmal als Monus *bezeichnet.)*

Übungsaufgabe 1.4 *Jedes LOOP-Programm endet nach endlich vielen Schritten. (Das ist nicht selbstverständlich. Aus unseren späteren Resultaten folgt sogar: Wenn eine Programmiersprache diese Eigenschaft hat, dann gibt es eine durch Computer berechenbare Funktion, die nicht in dieser Sprache berechenbar ist.)*

Übungsaufgabe 1.5 *Die LOOP-Sprache enthält mehr Redundanz, als man vielleicht auf den ersten Blick annehmen würde:*

- *Man kann problemlos ohne den* `Zero`*-Befehl auskommen.*
- *Man kann zusätzlich auch ohne den* `Val`*-Befehl auskommen, sofern man bereit ist, in Programmen zusätzliche Variablen einzuführen. (Von den 4 einfachen Befehlen genügen also* `Inc` *und* `Dec`*.)*
- *Oder man kann den* `Dec`*-Befehl und den* `Val`*-Befehl weglassen. (Von den vier einfachen Befehlen genügen also* `Inc` *und* `Zero`*.)*
- *Könnte man sogar nur mit* `Inc` *(und natürlich* **`loop`***) auskommen? Inwiefern ist diese Frage nicht gut gestellt?*[8]
- *Warum kann es ohne den* `Inc`*-Befehl nicht gehen?*

Übungsaufgabe 1.6 *Zeigen Sie jeweils auf zwei verschiedene Arten:*

- *Die durch $f(x, y) = x^y$ gegebene Funktion $f\colon \mathbb{N}^2 \to \mathbb{N}$ ist LOOP-berechenbar.*
- *Die Fakultätsfunktion ist LOOP-berechenbar.*

Übungsaufgabe 1.7 *Es gibt eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$, die nicht LOOP-berechenbar ist. (Hinweis: Wenn Ihnen das zu schwer vorkommt, zeigen Sie, es sogar überabzählbar viele Funktionen $f\colon \mathbb{N} \to \mathbb{N}$ gibt, die nicht LOOP-berechenbar sind.)*

[7] $\mathcal{P}(X) = \{A \mid A \subseteq X\}$ ist die Potenzmenge von X.

[8] Tipp: Denken Sie an die Anfangswerte von Variablen. Suchen Sie notfalls im Internet nach Informationen über `malloc` und `calloc`, zwei Standard-Funktionen der Programmiersprache C. Auch wenn Sie überhaupt nichts von C verstehen, müssten Sie doch den entscheidenden Hinweis erkennen.

1.2 Äquivalenz von primitiver Rekursivität und LOOP-Berechenbarkeit

Cantorsche Paarungsfunktion und simultane primitive Rekursion

Proposition 1.6 *Die folgenden Funktionen sind primitiv rekursiv:*

Monusfunktion $\dot{-}$
not
sgn
lt
gt
eq

$$f\colon \mathbb{N} \to \mathbb{N}, \qquad f(x) = x \dot{-} 1 = \max(x-1, 0)$$
$$f\colon \mathbb{N}^2 \to \mathbb{N}, \qquad f(x,y) = x \dot{-} y = \max(x-y, 0)$$
$$\mathrm{not}(x) = \begin{cases} 0 & \text{falls } x > 0 \\ 1 & \text{sonst} \end{cases} \qquad \mathrm{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{sonst} \end{cases}$$
$$\mathrm{lt}(x,y) = \chi_< = \begin{cases} 1 & \text{falls } x < y \\ 0 & \text{sonst} \end{cases} \qquad \mathrm{gt}(x,y) = \chi_> = \begin{cases} 1 & \text{falls } x > y \\ 0 & \text{sonst} \end{cases}$$
$$\mathrm{eq}(x,y) = \chi_= = \begin{cases} 1 & \text{falls } x = y \\ 0 & \text{sonst} \end{cases}$$
$$\min(x,y) \qquad \max(x,y)$$

Beweis $y \dot{-} 1$ lässt sich wie folgt durch primitive Rekursion definieren: $0 \dot{-} 1 = 0$ und $(y+1) \dot{-} 1 = y$. Also ist $y \dot{-} 1$ primitiv rekursiv. Nun lässt sich auch $x \dot{-} y$ durch primitive Rekursion definieren: $x \dot{-} 0 = x$ und $x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$.

$$\mathrm{not}(x) = 1 \dot{-} x \qquad \mathrm{sgn}(x) = \mathrm{not}(\mathrm{not}(x))$$
$$\mathrm{lt}(x,y) = \mathrm{sgn}(y \dot{-} x) \qquad \mathrm{gt}(x,y) = \mathrm{sgn}(x \dot{-} y)$$
$$\mathrm{eq}(x,y) = \mathrm{gt}(x+1, y) \cdot \mathrm{gt}(y+1, x)$$
$$\min(x,y) = x \cdot \mathrm{gt}(y,x) + y \cdot \mathrm{gt}(x,y) + x \cdot \mathrm{eq}(x,y) \qquad \max(x,y) = x + y - \min(x,y).$$

Wie man sieht, erhalten wie die weiteren Funktionen einfach durch Zusammensetzung. ∎

Fallunterscheidung

Proposition 1.7 *Wenn $f_1, f_2, g\colon \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch die durch*

$$h(\bar{x}) = \begin{cases} f_1(\bar{x}) & \text{falls } g(\bar{x}) > 0 \\ f_2(\bar{x}) & \text{sonst} \end{cases}$$

definierte Funktion $h\colon \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv.

Beweis $h(\bar{x}) = f_1(\bar{x}) \cdot \mathrm{sgn}(g(\bar{x})) + f_2(\bar{x}) \cdot \mathrm{not}(g(\bar{x}))$. ∎

Manchmal ist eine Funktion indirekt definiert, zum Beispiel als das Inverse einer anderen, bijektiven Funktion. Das folgende Resultat gibt uns in solchen Fällen einen einfachen Weg, um zu zeigen, dass das Inverse primitiv rekursiv ist. Voraussetzung hierfür ist, dass die ursprüngliche Funktion selbst primitiv rekursiv ist und dass wir die Funktionswerte der inversen Funktion nach oben abschätzen können – durch eine Funktion, von der wir bereits wissen, dass sie primitiv rekursiv ist.

Proposition 1.8 *Wenn $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ und $g\colon \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch die durch*

$$h(\bar{x}) = \min\{y \le g(\bar{x}) \mid y = g(\bar{x}) \text{ oder } f(\bar{x}, y) > 0\}$$

definierte Funktion $h\colon \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv.

Beweis Wir zeigen zunächst, dass die durch $h'(\bar{x}, z) = \min\{y \le z \mid y = z \text{ oder } f(\bar{x}, y) > 0\}$ definierte Funktion $h'\colon \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv ist. Es ist $h'(\bar{x}, 0) = 0$ sowie

$$h'(\bar{x}, y+1) = \begin{cases} h'(\bar{x}, y) & \text{falls } h'(\bar{x}, y) < y \\ y & \text{falls } h'(\bar{x}, y) = y \text{ und } f(\bar{x}, y) > 0 \\ y+1 & \text{sonst.} \end{cases}$$

Mittels primitiver Rekursion kann man daher sehen, dass h' tatsächlich primitiv rekursiv ist. Wegen $h(\bar{x}) = h'(\bar{x}, g(\bar{x}))$ ist h ebenfalls primitiv rekursiv. ∎

Wenn man aus irgendeinem Grund weiß, dass die Menge $\{y \in \mathbb{N} \mid f(\bar{x}) > 0\}$ für alle $\bar{x}$ nicht leer ist, dann kann man auch die Funktion $h(\bar{x}) = \min\{y \in \mathbb{N} \mid f(\bar{x}) > 0$ betrachten. Allerdings muss diese nicht unbedingt primitiv rekursiv sein, selbst wenn f es ist. Wenn man die primitiv rekursiven Funktionen unter dieser zusätzlichen Operation (der sogenannten μ-Rekursion) abschließt, erhält man die rekursiven Funktionen, von denen Kapitel 3 handelt.

μ-Rekursion

Korollar 1.9 *Sei $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv. Durch*

$$h(\bar{x}) = \mu y\big(f(\bar{x}, y) > 0\big) = \min\{y \in \mathbb{N} \mid f(\bar{x}, y) > 0\}$$

wird genau dann eine primitiv rekursive Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ definiert, wenn es eine primitiv rekursive Funktion $g\colon \mathbb{N}^k \to \mathbb{N}$ gibt, so dass für alle $\bar{x}$ gilt: $h(\bar{x}) \leq g(\bar{x})$.

Korollar 1.10 *Die Funktion* $f(x,y) = \begin{cases} \lceil \frac{x}{y} \rceil & \text{falls } y > 0 \\ 0 & \text{sonst} \end{cases}$ *ist primitiv rekursiv.*

Beweis Für $y > 0$ ist $\lceil \frac{x}{y} \rceil = \min\{z \leq x \mid z = x \text{ oder } y \cdot z \geq x\}$. ∎

Satz 1.11 (Cantorsche Paarungsfunktion)[9] *Die Funktion*

Cantorsche Paarungsfunktion J, L, R

$$\begin{aligned} J&\colon \mathbb{N}^2 \to \mathbb{N} \\ (x,y) &\mapsto J(x,y) = \frac{(x+y)(x+y+1)}{2} + y \end{aligned}$$

ist eine primitiv rekursive Bijektion zwischen $\mathbb{N}^2$ und $\mathbb{N}$. Die Komponenten $L, R\colon \mathbb{N} \to \mathbb{N}$ der Umkehrfunktion sind ebenfalls primitiv rekursiv. Außerdem ist $x, y \leq J(x,y)$ für alle $x, y \in \mathbb{N}$.

Beweis J ist wohldefiniert, weil $(x+y)(x+y+1)$ immer gerade ist. Man überlegt sich leicht, dass man durch Einschränkung von J für jedes $n \in \mathbb{N}$ eine Bijektion

$$J_n\colon \left\{(x,y) \in \mathbb{N}^2 \;\middle|\; x + y = n\right\} \to \left\{\frac{n(n+1)}{2}, \frac{n(n+1)}{2} + 1, \ldots, \frac{n(n+1)}{2} + n\right\}$$

erhält. Da $\mathbb{N}^2$ in die linken Seiten und $\mathbb{N}$ in die rechten Seiten partitioniert wird, folgt, dass J eine Bijektion ist. Da J aus primitiv rekursiven Funktionen zusammengesetzt ist (Addition, Multiplikation, Division durch 2, Projektionen und die Konstante 1), ist J auch primitiv rekursiv. Die Ungleichung $y \leq J(x,y)$ ist offensichtlich. Es gilt aber auch $x \leq \frac{x(x+1)}{2} \leq J(x,y)$, wobei man die Fälle $x = 0$ und $x \geq 1$ unterscheidet, um die erste Ungleichung einzusehen.

Seien $L\colon \mathbb{N} \to \mathbb{N}$ und $R\colon \mathbb{N} \to \mathbb{N}$ die beiden eindeutig bestimmten Funktionen, so dass $J(L(z), R(z)) = z$ für alle $z \in \mathbb{N}$. Wir müssen noch zeigen, dass L und R primitiv rekursiv sind. Zunächst einmal ist die Funktion

$$R'(z,x) = \min\{y \leq z \mid y = z \text{ oder } J(x,y) = z\}$$

sicher primitiv rekursiv. Weil $R(z) \leq z$ ist, ist $R'(z,x) = R(z)$ falls $x = L(z)$ ist. Daraus folgt zusammen mit $L(z) \leq z$, dass

$$L(z) = \min\{x \leq z \mid z = J(x, R(z,x))\}.$$

Dieser Darstellung sieht man die primitive Rekursivität von L an. Analog zeigt man, dass auch R primitiv rekursiv ist. ∎

Simultane primitive Rekursion

Satz 1.12 *Die primitiv rekursiven Funktionen sind abgeschlossen unter simultaner primitiv rekursiver Definition von mehreren Funktionen. Genauer:*

[9] Georg Cantor (1845–1918) gehörte zu den Begründern der damals noch „Mannigfaltigkeitslehre“ genannten Mengenlehre. Er beschrieb diese Paarungsfunktion 1878 explizit.

1. *Wenn die Funktionen* $f_1, f_2 \colon \mathbb{N}^k \to \mathbb{N}$ *und* $g_1, g_2 \colon \mathbb{N}^{k+1+2} \to \mathbb{N}$ *primitiv rekursiv sind, dann sind auch die durch*

$$\begin{aligned} h_1(\bar{x}, 0) &= f_1(\bar{x}) & h_2(\bar{x}, 0) &= f_2(\bar{x}) \\ h_1(\bar{x}, y+1) &= g_1(\bar{x}, y, h_1(\bar{x}, y), h_2(\bar{x}, y)) & h_2(\bar{x}, y+1) &= g_2(\bar{x}, y, h_1(\bar{x}, y), h_2(\bar{x}, y)) \end{aligned}$$

definierten Funktionen h_1 *und* $h_2 \colon \mathbb{N}^{k+1} \to \mathbb{N}$ *primitiv rekursiv.*

2. *Allgemeiner, wenn die Komponenten der Funktionen* $\bar{f} \colon \mathbb{N}^k \to \mathbb{N}^m$ *und* $\bar{g} \colon \mathbb{N}^{k+1+m} \to \mathbb{N}^m$ *primitiv rekursiv sind, dann sind auch die Komponenten der durch*

$$\begin{aligned} \bar{h}(\bar{x}, 0) &= \bar{f}(\bar{x}) \\ \bar{h}(\bar{x}, y+1) &= \bar{g}(\bar{x}, y, \bar{h}(\bar{x}, y)) \end{aligned}$$

definierten Funktion $\bar{h} \colon \mathbb{N}^{k+1} \to \mathbb{N}^m$ *primitiv rekursiv.*

Beweis Wir zeigen nur 1, weil der allgemeinere Fall 2 im Wesentlichen genauso geht. Im Fall $m = 2$ wenden wir die Cantorsche Paarungsfunktion J und ihre Umkehrfunktionen L und R an. Die Funktion $h(\bar{x}) = J(h_1(\bar{x}), h_2(\bar{x}))$ ist primitiv rekursiv, weil wir sie wie folgt durch primitive Rekursion erhalten:

$$\begin{aligned} h(\bar{x}, 0) &= J\big(f_1(\bar{x}), f_2(\bar{x})\big) \\ h(\bar{x}, y+1) &= J\big(g_1(\bar{x}, y, L(h(\bar{x}, y)), R(h(\bar{x}, y))), \\ &\qquad g_2(\bar{x}, y, L(h(\bar{x}, y)), R(h(\bar{x}, y)))\big) \end{aligned}$$

Also sind auch $h_1(\bar{x}, y) = L(h(\bar{x}, y))$ und $h_2(\bar{x}, y) = R(h(\bar{x}, y))$ primitiv rekursiv. ∎

LOOP-Programme mit Orakeln

In diesem Abschnitt werden wir die vier Befehle `Zero`, `Val`, `Inc` und `Dec` verallgemeinern, um das Einsetzen eines Programms in ein anderes besser beschreiben zu können.

Ein *Orakelname* darf wie ein Variablenname aus lateinischen Groß- und Kleinbuchstaben, Ziffern sowie dem Zeichen `_` bestehen. Allerdings muss er mit einem Großbuchstaben beginnen. Eine *Orakelsignatur* ist eine Menge von Orakelnamen zusammen mit einer natürlichen Zahl für jeden Orakelnamen, welche die Stelligkeit des Orakels angibt.[10]

Orakelname

Orakelsignatur

Ein *LOOP-Orakelprogramm* einer gegebenen Orakelsignatur ist wie ein LOOP-Programm in Abschnitt 1.1 zusammengesetzt aus LOOP-Anweisungen sowie Anweisungen der Gestalt

LOOP-Orakelprogramm

- `y = F()`, falls `F` ein nullstelliges Element der Signatur ist,
- `y = F(x_1)`, falls `F` ein einstelliges Element der Signatur ist,
- `y = F(x_1,x_2)`, falls `F` ein zweistelliges Element der Signatur ist,
- `y = F(x_1,x_2,x_3)`, falls `F` ein dreistelliges Element der Signatur ist,
- usw.

Die gewöhnlichen LOOP-Programme sind also genau die LOOP-Orakelprogramme der Signatur bestehend aus dem nullstelligen Orakelnamen `Zero` und den einstelligen Orakelnamen `Val`, `Inc` und `Dec`. Um ein Orakelprogramm auszuführen, braucht man zusätzlich noch eine Interpretation zu jedem Orakelnamen der Orakelsignatur. Die Interpretation eines k-stelligen Orakelnamens ist eine k-stellige Funktion $f \colon \mathbb{N}^k \to \mathbb{N}$. Diese kann beispielsweise durch ein weiteres LOOP-Programm oder auch LOOP-Orakelprogramm gegeben sein. (Mit dieser Definition ist es allerdings nicht möglich, dass LOOP-Orakelprogramme sich rekursiv selbst aufrufen!)

Satz 1.13 *Die folgenden Mengen von Funktionen sind identisch.*

1. *Alle primitiv rekursiven Funktionen.*

[10] Im Interesse der Lesbarkeit werden wir die Funktion, welche die Stelligkeiten angibt, aus der Notation unterdrücken und Signaturen in der folgenden Art beschreiben: „Betrachte die Signatur {`Zero`, `Val`, `Inc`, `Dec`}, wobei `Zero` nullstellig ist und alle anderen Elemente einstellig sind."

2. *Alle Funktionen, die sich durch ein LOOP-Programm berechnen lassen.*

3. *Alle Funktionen, die sich durch ein LOOP-Orakelprogramm berechnen lassen, wobei jeder Orakelname durch eine primitiv rekursive Funktion interpretiert wird.*

Beweis Nach Satz 1.5 lassen sich alle primitiv rekursiven Funktionen durch ein LOOP-Programm berechnen. Alle durch ein LOOP-Programm berechenbaren Funktionen liegen in der durch 3 beschriebenen Menge, weil die Standardinterpretationen von `Zero`, `Inc`, `Dec`, `Val` primitiv rekursiv sind. Und nach dem folgenden Lemma sind alle in der durch 3 beschriebenen Menge liegenden Funktionen primitiv rekursiv. ∎

Lemma 1.14 *Wenn man ein LOOP-Orakelprogramm ausführt und dabei jeden Orakelnamen durch eine primitiv rekursive Funktion interpretiert, dann ergibt sich der Endwert jeder Variablen durch eine primitiv rekursive Funktion aus den Anfangswerten aller Variablen.*

Beweis Offenbar ist jedes beliebige LOOP-Programm entweder leer, ein einzelner einfacher Befehl, ergibt sich durch Hintereinanderreihen von zwei echt kleineren (daher nichtleeren) LOOP-Programmen, oder besteht aus einer loop-Schleife, deren Schleifenkörper von einem echt kleineren LOOP-Programm gebildet wird. Das erlaubt es uns, die Behauptung durch Induktion zu beweisen.

Die Behauptung ist klar, falls das Programm leer ist (die Variablen ändern sich gar nicht), aus einem einzigen einfachen Befehl besteht (die Funktionen, die Orakel interpretieren, sind laut Voraussetzung primitiv rekursiv) oder durch Hintereinanderreihen aus zwei kleineren Programmen besteht, für welche die Behauptung wahr ist (die neuen Variablenwerte ergeben sich aus den alten durch Funktionen, die aus primitiv rekursiven zusammengesetzt sind).

Der einzige interessante Fall ist, wenn das Programm aus einer äußeren **`loop`**-Schleife mit einem beliebigen LOOP-Programm als Schleifenkörper im Innern besteht. Für den Schleifenkörper gilt nach der Induktionsvoraussetzung, dass die Variablenwerte am Ende sich durch primitiv rekursive Funktionen aus den Variablenwerten am Anfang berechnen lassen. Die Funktionen, die uns in Abhängigkeit von den Anfangswerten (einschließlich n) die Werte nach n Durchgängen durch die Schleife geben, erhalten wir durch simultane primitive Rekursion wie in Satz 1.12. Sie sind daher ebenfalls primitiv rekursiv. ∎

Beispiel

Satz 1.13 macht es relativ einfach, zu zeigen, dass eine gegebene Funktion primitiv rekursiv ist: Dazu genügt es, ein LOOP-Programm anzugeben, das die Funktion berechnet und dabei allenfalls primitiv rekursive Orakel benutzt. Um zu zeigen, dass die Interpretation eines Orakels primitiv rekursiv ist, können wir wiederum (müssen aber nicht) den Satz selbst anwenden. Als Beispiel leiten wir aus der Cantorschen Paarungsfunktion J eine Bijektion zwischen $\mathbb{N}$ und $\mathbb{N}^* = \bigcup_{k\in\mathbb{N}} \mathbb{N}^k$ ab und zeigen, dass alle damit zusammenhängenden Funktionen primitiv rekursiv sind.

Definition 1.15 *Für jedes $k \in \mathbb{N} \setminus \{0\}$ definieren wir eine Bijektion $J_k \colon \mathbb{N}^k \to \mathbb{N}$ wie folgt: $J_1(x_0) = x_0$, $J_2(x_0, x_1) = J(x_0, x_1)$, $J_3(x_0, x_1, x_2) = J(x_0, J_2(x_1, x_2))$ usw. Die allgemeine Regel ist also $J_{k+1}(x_0, \ldots, x_k) = J(x_0, J_k(x_1, \ldots, x_k))$.*

Darauf aufbauend, definieren wir die Bijektion $\langle\cdot\rangle : \mathbb{N}^* \to \mathbb{N}$ durch

$$\langle x_0, \ldots, x_{k-1}\rangle = \begin{cases} 0 & \text{falls } k = 0 \\ J(k-1, J_k(x_0, \ldots, x_{k-1})) + 1 & \text{sonst.} \end{cases}$$

Proposition 1.16 *Die folgenden Funktionen sind alle primitiv rekursiv:*

- *Für jedes $k \in \mathbb{N}$ die Einschränkung von $\langle\cdot\rangle$ auf $\mathbb{N}^k$.*
- $\mathrm{Lg} \colon \mathbb{N} \to \mathbb{N}$, *so dass* $\mathrm{Lg}(x)$ *dasjenige $k \in \mathbb{N}$ ist, so dass x von der Form $\langle x_0, \ldots, x_{k-1}\rangle$ ist.*
- $\mathrm{Component} \colon \mathbb{N}^2 \to \mathbb{N}$, *so dass* $\mathrm{Component}(\langle x_0, \ldots, x_{k-1}\rangle, i) = x_i$ *falls $i \leq k-1$ und* $\mathrm{Component}(x, i) = 0$ *falls $i > k-1$.*
- $\mathrm{Replace} \colon \mathbb{N}^3 \to \mathbb{N}$, *so dass*

$$\mathrm{Replace}(\langle x_0, \ldots, x_{k-1}\rangle, i, y) = \begin{cases} \langle x_0, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{k-1}\rangle & i < k \\ \langle x_0, \ldots, x_{k-1}, 0, \ldots, 0, y\rangle & \text{sonst.} \end{cases}$$

Dabei wird das Tupel im Fall $i \geq k$ mit sovielen Nullen aufgefüllt, dass es die Länge $i+1$ hat.

Beweis Dass die Einschränkungen von $\langle\cdot\rangle$ auf $\mathbb{N}^k$ primitiv rekursiv sind, beweist man einfach durch Induktion nach k. Noch einfacher sieht man, dass Lg primitiv rekursiv ist. Für Component geben wir ein LOOP-Programm mit Orakeln an.

```
output = Zero()
maxindex = L(input_1)

index_is_small = Lt(input_2, maxindex)
if index_is_small then {
  x = R(input_1)
  loop input_2 times {
    x = R(x)
  }
  output = L(x)
}

index_is_last = Eq(input_2, maxindex)
if index_is_last then {
  x = Val(input_1)
  loop input_2 times {
    x = R(x)
  }
  output = R(x)
}
```

Zum einfacheren Verständnis benutzt das Programm als Abkürzung die Kontrollstruktur **`if ... then`**, von der wir früher gesehen haben, wie man sie übersetzen kann. Das zweistellige Orakel `Lt` wird interpretiert durch die primitiv rekursive Funktion $\mathrm{lt} = \chi_<$ und das zweistellige Orakel `Eq` durch die primitiv rekursive Funktion $\mathrm{eq} = \chi_=$. Die einstelligen Orakel `L` und `R` werden interpretiert durch die primitiv rekursiven Funktionen L und R, die zusammen das Inverse der Cantorschen Paarungsfunktion J bilden.

Die Funktion Replace wird durch das folgende LOOP-Orakelprogramm berechnet.

```
  zero = Zero()
  x = input_1
  i = input_2
  y = input_3
  n = Lg(x)
  output = Zero()
  if n then {
    last_index = Inc(zero)
    loop n times {
      n = Dec(n)
      z = Component(x,n)
      switch_value = Eq(n,i)
      if switch_value then {
        z = Val(y)
      }
      output = J(z,output)
      if last_index then {
        output = Val(z)
        last_index = Zero()
      }
    }
    n = Lg(x)
    n = Dec(n)
    output = J(n,output)
    output = Inc(output)
  }
```

Dabei ist `Eq` durch $\chi_=$ zu interpretieren, `J` durch die Cantorsche Paarungsfunktion J und `Lg` sowie `Component` durch die Funktionen Lg und Component. ∎

Übungsaufgaben

Übungsaufgabe 1.8 *Die Funktion* $f(x,y) = \begin{cases} \lfloor \frac{x}{y} \rfloor & \text{falls } y > 0 \\ 0 & \text{sonst} \end{cases}$ *ist primitiv rekursiv.*

Übungsaufgabe 1.9 *Gegeben ein* $k \in \mathbb{N}$ *betrachten Sie die primitiv rekursive Bijektion* $J_k \colon \mathbb{N}^k \to \mathbb{N}$ *aus Definition 1.15 und zeigen Sie, dass auch die Komponenten der Umkehrfunktion primitiv rekursiv sind.*

Übungsaufgabe 1.10 *Die Definition der LOOP-Programmiersprache ist für mathematische Verhältnisse nicht sehr präzise. In der Informatik ist es üblich, Programmiersprachen in der EBNF (erweiterten Backus-Naur-Form) zu definieren, was aber leider auch nicht viel genauer ist. Der Grund ist wohl, dass der hohe Aufwand einer präzisen Definition in keinem vernünftigen Verhältnis zum Nutzen steht. Hier ist eine EBNF für LOOP-Orakelprogramme beliebiger Orakelsignatur.*

⟨capital-letter⟩ = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
⟨small-letter⟩ = a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
⟨character⟩ = ⟨capital-letter⟩|⟨small-letter⟩|0|1|2|3|4|5|6|7|8|9|_
⟨variable⟩ = ⟨small-letter⟩ **{** ⟨character⟩ **}**
⟨oracle⟩ = ⟨capital-letter⟩ **{** ⟨character⟩ **}**
⟨right-value-0⟩ = ⟨oracle⟩ ()
⟨right-value-1⟩ = ⟨oracle⟩ (**{** ⟨variable⟩ , **}** ⟨variable⟩)
⟨right-value⟩ = ⟨right-value-0⟩ | ⟨right-value-1⟩
⟨assignment⟩ = ⟨variable⟩ = ⟨right-value⟩
⟨loop⟩ = loop ⟨variable⟩ times { ⟨program⟩ }
⟨statement⟩ = ⟨assignment⟩ | ⟨loop⟩
⟨program⟩ = **{** ⟨statement⟩ **}**

Falls Sie mit EBNF nicht vertraut sind: Links stehen jeweils für „nichtterminale Symbole“, d.h., Bezeichner für syntaktische Einheiten wie Programm oder Variable, die dann jeweils von der Form sein müssen, wie rechts angegeben. Als besondere Symbole gibt es u.A. **{** *. . .* **}** *(hier fett gedruckt) für beliebige (auch nullmalige) Wiederholung und | für „oder“. Wie üblich sind hier die Stelligkeit von Funktionen (bei uns: Orakel) und die Bedeutung von Leerzeichen, neuer Zeile usw. nicht berücksichtigt.*

Die EBNF entspricht ungefähr unseren induktiven Definitionen, die wir im nächsten Kapitel auch dazu verwenden werden, „Sprachen“, d.h. Mengen von Strings, zu definieren. (Man kann eine EBNF für eine Programmiersprache übrigens dazu verwenden, automatisch ein Gerüst für einen Compiler einschließlich Fehlerbehandlung zu erzeugen, bei dem man dann vor allem noch die eigentliche Codeerzeugung implementieren muss.)

Zeigen Sie am Beispiel der Standard-Orakelsignatur {`Zero`, `Val`, `Inc`, `Dec`}*, wie man bei vorgegebener Signatur in* ⟨*right-value*⟩ *auch die Stelligkeit genau vorgeben kann.*

Ergänzen Sie die vorgegebene EBNF, so dass auch der Befehl **`if ... then { ... }`** *erlaubt ist. Ergänzen Sie sie weiter, so dass man hinter* **`loop`** *und* **`if`** *sowie auf der rechten Seite von Gleichungen beliebige Terme verwenden kann.*

1.3 Hyperoperatoren und Ackermannfunktion

In den Zwanzigerjahren des 20. Jahrhunderts fragte David Hilbert, ob die primitiv rekursiven Funktionen den intuitiven Begriff der Berechenbarkeit genau beschreiben. Schon kurze Zeit später fanden jedoch seine Schüler Wilhelm Ackermann und Gabriel Sudan Funktionen, deren Werte man mit Papier und Bleistift im Prinzip alle ausrechnen kann, die jedoch nicht primitiv rekursiv sind. Bei der ursprünglichen, dreistelligen Ackermannfunktion handelt es sich um eine Variante der Hyperoperatoren, wie wir sie in diesem Abschnitt betrachten werden.[11]

Jede der grundlegenden mathematischen Operationen Addition, Multiplikation und Exponentiation lässt sich aus der jeweils vorhergehenden nach demselben Prinzip rekursiv definieren, wobei wir als Vorgänger der Addition die Nachfolgerfunktion S nehmen (was ein bisschen künstlich ist).

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

Wie man sieht, sind die Definitionen im Fall $y = 0$ (rechts) jeweils unterschiedlich, aber im Fall $y' = y + 1$ (links) kann man bei genauerem Hinschauen ein einheitliches Prinzip erkennen. Die *Hyperoperatoren* erhalten wir, indem wir die Folge nach demselben Prinzip weiter fortsetzen, wobei wir für jeden der neuen Operatoren im Fall $y = 0$ genauso vorgehen, wir für die Exponentation. Für die Hyperoperatoren wurden viele verschiedene Notationen vorgeschlagen, aber die von Donald Knuth[12] eingeführte suggestive Schreibweise $x \uparrow^n y$ gehört sicherlich zu den bekanntesten und praktischsten. Es ist $x \uparrow^0 y = x \cdot y$ und $x \uparrow^1 y = x^y$, und die allgemeine Definition geht wie folgt.

Hyperoperatoren $\uparrow$

Definition 1.17 (Hyperoperatoren in Knuthscher Pfeilnotation)

$$\begin{aligned} x \uparrow^0 y &= x \cdot y & \qquad x \uparrow^{n+1} 0 &= 1 \\ & & \qquad x \uparrow^{n+1} (y+1) &= x \uparrow^n (x \uparrow^{n+1} y) \end{aligned}$$

Die Hyperoperatoren wachsen sehr schnell – aber im Fall $x = 2$ erst ab $y \geq 3$.

Lemma 1.18 *Für alle $n \in \mathbb{N}$ gilt:*

1. $2 \uparrow^n 1 = 2$
2. $2 \uparrow^n 2 = 4$
3. $2 \uparrow^{n+1} 3 = 2 \uparrow^n 4$.

Beweis Zu 1: $2 \uparrow^{n+1} 1 = 2 \uparrow^n (2 \uparrow^{n+1} 0) = 2 \uparrow^n 1 = \ldots = 2 \uparrow^0 1 = 2 \cdot 1 = 2$. Mit 1 können wir 2 beweisen: $2 \uparrow^{n+1} 2 = 2 \uparrow^n (2 \uparrow^{n+1} 1) = 2 \uparrow^n 2 = \ldots = 2 \uparrow^0 2 = 2 \cdot 2 = 4$. Aus 2 folgt nun direkt 3: $2 \uparrow^{n+1} 3 = 2 \uparrow^n (2 \uparrow^{n+1} 2) = 2 \uparrow^n 4$. ∎

Man kann die Schreibweise $x \uparrow^n y$ auch noch auf die Fälle $n = -1$ und $n = -2$ fortsetzen, und zwar setzt man $x \uparrow^{-1} y = x + y$ und $x \uparrow^{-2} y = x + 1$. Dabei ist dann aber zu beachten, dass die Gleichung $x \uparrow^n 0 = 1$ für $n \leq 0$ nicht gilt! Ebenso gilt das obige Lemma für $n < 0$ nicht. In der Tat ist $2 \uparrow^{-2} 1 = 2 \uparrow^{-1} 1 = 2 \uparrow^{-2} 2 = 3$. $\uparrow^{-1}$, $\uparrow^{-2}$

Bemerkung 1.19 $\uparrow^0$ *ist die Multiplikation.* $\uparrow = \uparrow^1$ *ist die Exponentiation* $x \uparrow y = x^y$. $\uparrow\uparrow = \uparrow^2$ *ist der* Potenzturm*, das heißt*

$$x \uparrow\uparrow y = \begin{cases} 1 & \text{falls } y = 0 \\ x^{\cdot^{\cdot^{\cdot^{x^x}}}} & \text{sonst } (y \text{ Kopien von } x). \end{cases}$$

[11] Wilhelm Ackermann (1896–1962) und Gabriel Sudan (1899–1977) waren Schüler von Hilbert. Ackermann wurde nach seiner Eheschließung von Hilbert bewusst in seiner Hochschulkarriere nicht mehr unterstützt und wurde Gymnasiallehrer, blieb aber in der Forschung aktiv. Die heute oft als Ackermannfunktion bezeichnete zweistellige Funktion geht auf die ungarische Logikerin Rózsa Péter (1905–1977) zurück und ist deshalb auch als Ackermann-Péter-Funktion bekannt.

[12] Donald Knuth (geboren 1938) ist ein amerikanischer Informatiker. Als Autor des Textsatzsystems TEX und Verfasser von *The Art of Computer Programming*, dem wohl einzigen über 40 Jahre alten Informatik-Lehrbuch, das immer noch nicht veraltet ist, genießt er eine Art Kultstatus in der Informatik.

x	0	1	2	3	4	5	6	7	8	9	10
$A_0(x)$	1	2	3	4	5	6	7	8	9	10	11
$A_1(x)$	2	3	4	5	6	7	8	9	10	11	12
$A_2(x)$	3	5	7	9	11	13	15	17	19	21	23
$A_3(x)$	5	13	29	61	125	253	509	1021	2045	4093	8189
$A_4(x)$	13	65533	(19729 Stellen)	...	...	...	...	...	...	...	...
$A_5(x)$	65533	...	...	...	...	...	...	...	...	...	...
$A_6(x)$	...	...	...	...	...	...	...	...	...	...	...

Abbildung 1.1: Ackermann-Péter-Funktion

Ziel dieses Abschnitts ist der Beweis des folgenden Satzes. Weil es natürlich im Prinzip[13] ein Computerprogramm gibt, das für beliebige Eingaben x, y, n den Wert $x \uparrow^n y$ berechnet, zeigt der Satz, dass nicht alle im anschaulichen Sinne berechenbaren Funktionen primitiv rekursiv sind. Das motiviert die Definition der rekursiven Funktionen später in Kapitel 3.

Satz 1.20 *Zwar ist jede der Funktionen $\bullet \uparrow^n \bullet\colon \mathbb{N}^2 \to \mathbb{N}$, $(x, y) \mapsto x \uparrow^n y$ primitiv rekursiv, aber die dreistellige Funktion $\bullet \uparrow^\bullet \bullet\colon \mathbb{N}^3 \to \mathbb{N}$, $(x, y, n) \mapsto x \uparrow^n y$ ist nicht primitiv rekursiv.*

Die primitive Rekursivität der zweistelligen Funktionen $\uparrow^n$ ist leicht einzusehen. Um den schwierigen Teil von Satz 1.20 zu beweisen, arbeiten wir mit einer etwas einfacheren, eng mit $\uparrow$ verwandten Funktion, der Ackermann-Péter-Funktion.

Ackermann-Péter-Funktion $A(n, x)$

Definition 1.21 *Die Ackermann-Péter-Funktion $A\colon \mathbb{N}^2 \to \mathbb{N}$, meist nicht ganz zutreffend als Ackermannfunktion[14] bezeichnet, ist durch die folgenden Gleichungen definiert.*

$$A_0(x) = x + 1 \qquad A_{n+1}(0) = A_n(1)$$
$$A_{n+1}(x+1) = A_n\big(A_{n+1}(x)\big).$$

Wir schreiben $A(n, x)$ oder $A_n(x)$ je nachdem, ob wir uns A als eine einzige zweistellige Funktion oder als eine Folge von einstelligen Funktionen A_n vorstellen, von denen dann jede durch die vorhergehende mittels $A_{n+1}(x) = A_n^{x+1}(1)$ gegeben ist. (A_n^{x+1} steht für die $x+1$-fache Iteration der Funktion A_n.)

Proposition 1.22 *Die Ackermann-Péter-Funktion ist wohldefiniert, und jede der einstelligen Funktionen A_n ist primitiv rekursiv. Es gilt außerdem:*

$$A_n(x) = \begin{cases} x + 1 & \text{falls } n = 0 \\ x + 2 & \text{falls } n = 1 \\ \big(2 \uparrow^{n-2} (x+3)\big) - 3 & \text{sonst.} \end{cases}$$

Insbesondere ist $A_2(x) = 2x + 3$ und $A_3(x) = 2^{x+3} - 3$. Mit der Konvention $x \uparrow^{-1} y = x + y$ ist auch $A_1(x) = 2 \uparrow^{1-2} (x+3) - 3$, so dass die zweite Zeile der obigen Fallunterscheidung überflüssig ist. Aber A_0 ist ein Sonderfall: $A_0(x) = x + 1$, aber $2 \uparrow^{0-2} (x+3) - 3 = (2 + 1) - 3 = 0$.

Beweis Man zeigt durch Induktion nach n, dass jede der Funktionen A_n wohldefiniert und primitiv rekursiv ist. Seien A'_n die Funktionen aus der Proposition. Man überprüft die Rekursionsgleichungen problemlos für die angegebenen Lösungen für A'_0, A'_1 und A'_2. Für $n \geq 2$ folgen die Rekursionsgleichungen aus denen von $\uparrow$:

$$\begin{aligned} A'_{n+1}(0) &= (2 \uparrow^{n-1} 3) - 3 \\ &= (2 \uparrow^{n-2} 4) - 3 \\ &= A'_n(1) \end{aligned} \qquad \begin{aligned} A'_{n-1}(x+1) &= \big(2 \uparrow^{n-1} (x+4)\big) - 3 \\ &= 2 \uparrow^{n-2} \big([2 \uparrow^{n-1} (x+3)] - 3 + 3\big) - 3 \\ &= A'_n(A'_{n+1}(x)). \end{aligned}$$

(Für $A'_{n+1}(0)$ brauchen wir außerdem noch Lemma 1.18.) Da die Funktionen A'_n die Rekursionsgleichungen erfüllen, welche die A_n eindeutig definieren, ist $A_n = A'_n$. ∎

Wir beweisen jetzt einige grundlegende Tatsachen über das Wachstum von A.

[13] In der Praxis geht das nur deshalb nicht, weil die zu berechnenden Werte sehr schnell astronomisch werden.
[14] In der Literatur wird manchmal auch die einstellige Funktion $A(n, n)$ Ackermannfunktion genannt.

Lemma 1.23 *Die zweistellige Funktion $A(n,x) = A_n(x)$ wächst streng monoton in beiden Argumenten und stärker im ersten als im zweiten. Genauer: Es gilt $A_{n+1}(x) \geq A_n(x+1) \geq A_n(x)+1$ für alle $x, n \in \mathbb{N}$.*

Beweis Wir bemerken zunächst noch, dass offensichtlich $A_n(x) \geq 1$ für alle $n, x \in \mathbb{N}$. Wir werden simultan durch Induktion nach n zeigen:

1. $A_n(x+1) \geq A_n(x)+1$ für alle $x \in \mathbb{N}$ (und folglich $A_n(x) \geq x+1$ für alle $x \in \mathbb{N}$),
2. $A_{n+1}(x) \geq A_n(x+1)$ für alle $x \in \mathbb{N}$.

Zu 1: Offensichtlich für $n = 0$. Im Fall $n > 0$ ist $A_n(x+1) = A_{n-1}(A_n(x)) \geq A_n(x)+1$ nach Induktionsvoraussetzung. Zu 2: Wegen 1 ist $A_n(1) \geq A_n(0)+1 \geq 2$. Mit 1 folgt $A_n^{x+1}(1) = A_n^x(A_n(1)) \geq A_n^x(2) \geq A_n^x(1)+1$, für alle $x \in \mathbb{N}$. Folglich ist $A_n^x(1) \geq A_n^0(1)+x = x+1$. Wiederum mit 1 folgt $A_{n+1}(x) = A_n(A_n^x(1)) \geq A_n(x+1)$. ∎

Lemma 1.24 *Für alle $n \geq 1$ ist $A_n(2x) < A_{n+1}(x)$.*

Beweis Für $x = 0$ ist das klar. Sei also $x > 0$. Wegen $A_2(x) = 2x+3$ ist $2x < A_2(x-1)$. Es folgt $A_n(2x) < A_n(A_2(x-1)) \leq A_n(A_{n+1}(x-1)) = A_{n+1}(x)$. ∎

Satz 1.25 *Für jede primitiv rekursive Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ gibt es ein $n \in \mathbb{N}$, so dass für alle $\bar{x} \in \mathbb{N}^k$ gilt:*

$$f(\bar{x}) < A_n(\Sigma\bar{x}),$$

wobei $\bar{x} = (x_1, \ldots, x_k)$ und $\Sigma\bar{x} = x_1 + \cdots + x_k$ ist. $\Sigma\bar{x}$

Beweis Der Beweis geht durch Induktion über die primitiv rekursiven Funktionen: Wir zeigen, dass die Menge der Funktionen, die sich wie beschrieben abschätzen lassen, die Grundfunktionen 0, S und π_i^k enthält und unter Zusammensetzung und primitiver Rekursion abgeschlossen ist.

Nullfunktion Es ist $0 < 1 = A_0(0) = A_0(\Sigma())$, wobei $() \in \mathbb{N}^0$ das leere Tupel ist.

Nachfolgerfunktion Es ist $S(x) = x+1 < x+2 = A_1(x)$.

Projektionsfunktionen Es ist $\pi_i^k(\bar{x}) = x_i \leq \Sigma\bar{x} < \Sigma\bar{x}+1 = A_0(\Sigma\bar{x})$.

Zusammensetzung Wenn sich $f\colon \mathbb{N}^\ell \to \mathbb{N}$ und alle Komponenten von $\bar{g}\colon \mathbb{N}^k \to \mathbb{N}^\ell$ so abschätzen lassen, dann gibt es wegen der Monotonie von A auch ein gemeinsames $n \geq 1$, so dass jede dieser Funktionen sich durch A_n abschätzen lässt:

$$f(\bar{y}) < A_n(\Sigma\bar{y}) \qquad\qquad g_1(\bar{x}) < A_n(\Sigma\bar{x})$$
$$\vdots$$
$$g_\ell(\bar{x}) < A_n(\Sigma\bar{x}).$$

Folglich ist mit $N > n + \log_2(k+1)$:

$$\begin{aligned}
h(\bar{x}) = f(\bar{g}(\bar{x})) &< A_n(\Sigma\bar{g}(\bar{x})) && \text{(Abschätzung von } f\text{)}\\
&< A_n(k \cdot A_n(\Sigma\bar{x})) && \text{(Abschätzung der } g_i\text{)}\\
&< A_N(A_n(\Sigma\bar{x})) && \text{(Lemma 1.24)}\\
&< A_N(A_{N+1}(\Sigma\bar{x})) && \text{(Lemma 1.23)}\\
&= A_{N+1}(\Sigma\bar{x}+1) && \text{(Definition von } A\text{)}\\
&\leq A_{N+2}(\Sigma\bar{x}). && \text{(Lemma 1.23)}
\end{aligned}$$

Primitive Rekursion Gegeben $f\colon \mathbb{N}^k \to \mathbb{N}$ und $g\colon \mathbb{N}^{k+2} \to \mathbb{N}$, die sich durch A_n nach oben abschätzen lassen, und $h\colon \mathbb{N}^{k+1} \to \mathbb{N}$ die aus f und g durch primitive Rekursion definierte Funktion:

$$h(\bar{x}, 0) = f(\bar{x}) \qquad\qquad f(\bar{x}) < A_n(\Sigma\bar{x})$$
$$h(\bar{x}, y+1) = g(\bar{x}, y, h(\bar{x}, y)) \qquad\qquad g(\bar{x}, y, z) < A_n(\Sigma\bar{x} + y + z).$$

Wir beweisen nun durch vollständige Induktion nach y, dass h durch A_{n+2} nach oben abgeschätzt wird, dass also $h(\bar{x}, y) < A_{n+2}(\Sigma x + y)$ gilt. Die Induktionsbasis $y = 0$ ist klar: $h(\bar{x}, 0) = f(\bar{x}) < A_n(\Sigma\bar{x}) < A_{n+2}(\Sigma\bar{x} + 0)$. Der Induktionsschritt von y nach $y + 1$ geht so:

$$\begin{aligned}
h(\bar{x}, y{+}1) &= g(\bar{x}, y, h(\bar{x}, y)) \\
&< A_n(\Sigma\bar{x} + y + h(\bar{x}, y)) && \text{(Abschätzung von } g\text{)} \\
&< A_n(\Sigma\bar{x} + y + A_{n+2}(\Sigma\bar{x} + y)) && \text{(Induktionsvoraussetzung)} \\
&< A_n(2 \cdot A_{n+2}(\Sigma\bar{x} + y)) \\
&< A_{n+1}(A_{n+2}(\Sigma\bar{x} + y)) && \text{(Lemma 1.24)} \\
&= A_{n+2}(\Sigma\bar{x} + y + 1) && \text{(Definition von } A\text{).}
\end{aligned}$$

Die Menge der wie behauptet abschätzbaren Funktionen erfüllt also alle Bedingungen aus Definition 1.1 und enthält folglich die Menge der primitiv rekursiven Funktionen. ∎

Satz 1.20 können wir jetzt als einfache Folgerung aus Satz 1.25 beweisen.

Beweis [von Satz 1.20] Die Multiplikation $\uparrow^0$ ist bekanntlich primitiv rekursiv. Jede der Funktionen $\uparrow^{n+1}$ ist nun aber explizit durch primitive Rekursion aus $\uparrow^n$ definiert.

Wenn die dreistellige Funktion $\uparrow$ primitiv rekursiv wäre, dann wäre auch die Funktion $A_*(n) = A_n(n) = 2 \uparrow^n (n + 3) - 3$ primitiv rekursiv, und daher gäbe es nach Satz 1.25 ein $N \in \mathbb{N}$, so dass $A_*(n) < A_N(n)$ für alle $n \in \mathbb{N}$. Dann wäre aber $A_N(N) = A_*(N) < A_N(N)$, ein Widerspruch. ∎

Übungsaufgaben

Übungsaufgabe 1.11 *Die ursprüngliche Ackermannfunktion ist durch die folgenden Rekursionsgleichungen definiert.*

$$A(x, y, 0) = x + y \qquad A(x, 0, n + 1) = \begin{cases} n & n \leq 1 \\ x & n \geq 2 \end{cases}$$

$$A(x, y + 1, n + 1) = A(x, A(x, y, n + 1), n).$$

Zeigen Sie, dass diese Funktion wohldefiniert ist, und dass gilt:

$$A(x, y, n) = \begin{cases} x \uparrow^{n-1} y & \text{falls } n \leq 2 \\ x \uparrow^{n-1} (y + 1) & \text{falls } n \geq 3. \end{cases}$$

Übungsaufgabe 1.12 *Betrachten wir das folgende (erweiterte) LOOP-Orakelprogramm in der Orakelsignatur* {`Zero`, `Dec`, `Multiply`, `Hyper`}.

```
zero = Zero()
x = Val(input_1)
y = Val(input_2)
n = Val(input_3)
output = Multiply(x,y)
if n then {
  output = Inc(zero)
  if y then {
    y_minus_1 = Dec(y)
    n_minus_1 = Dec(n)
    z = Hyper(x,y_minus_1,n)
    output = Hyper(x,z,n_minus_1)
  }
}
```

Geben Sie Interpretationen der Orakelnamen an, so dass das Programm die Funktion $H(x, y, n) = x \uparrow^n y$ berechnet. Nehmen Sie dann Stellung zu dem folgenden Argument: „H lässt sich durch ein LOOP-Orakelprogramm berechnen, wobei jeder Orakelname seinerseits durch ein LOOP-Orakelprogramm berechnet wird. Nach Satz 1.13 ist H daher primitiv rekursiv.“

Übungsaufgabe 1.13 *An dieser Aufgabe können Sie testen, wie gut Sie Ihnen unbekannte LOOP-Programme lesen können. Im Interesse der Lesbarkeit werden diese Programme stillschweigend von den früher beschriebenen Vereinfachungen Gebrauch machen. Darüber hinaus sollen Sie zeigen, dass es eine primitiv rekursive Funktion $A' \colon \mathbb{N}^3 \to \mathbb{N}$ gibt, so dass gilt:*

- *Für alle $n, x \in \mathbb{N}$ existiert ein $f \in \mathbb{N}$, so dass $A'(n, x, f) = A(n, x)$.*

- *Für alle* $n, x, f \in \mathbb{N}$ *ist entweder* $A'(n,x,f) = A(n,x)$ *oder* $A'(n,x,f) = 0$.

Da immer $A(n,x) > 0$ *ist, kann man* $A(n,x)$ *also dadurch berechnen, dass man* f *von 0 ausgehend schrittweise erhöht und* $A'(n,x,f)$ *solange berechnet, bis man ein von 0 verschiedenes Ergebnis erhält. Zeigen oder widerlegen Sie, dass daraus folgt, dass* A *primitiv rekursiv ist.*

Betrachten wir zunächst das folgende LOOP-Orakelprogramm in der Orakelsignatur $\mathcal{S} = \{$`Val`, `Zero`, `Inc`, `Dec`, `Eq`, `J`, `L`, `R`, `Component`, `Lg`$\}$.

```
n = Val(input_1)
x = Val(input_2)
a = Val(input_3)
function = Val(input_4)
output = Zero()
if Not(n) then {
  if Eq(p,Inc(x)) then {
    output = Inc(ouput)
  }
}
if n then {
  if Not(x) then {
    if Eq(a,Component(function,J(n,x))) then {
      output = Inc(output)
    }
  }
  if x then {
    r = Component(function,J(n,Dec(x)))
    if r then {
      if Eq(p,Component(function,J(Dec(n),r))) then {
        output = Inc(output)
      }
    }
  }
}
if Not(a) then {
  output = Zero()
}
```

Wenn wir alle Orakelnamen auf die offensichtliche Weise interpretieren (siehe hierzu insbesondere Satz 1.11 und Proposition 1.16), dann erhalten wir eine primitiv rekursive Funktion $\mathrm{Check}_1(n,x,a,f)$*, welche die Zahl* f *als Einschränkung von* A *auf eine endliche Teilmenge von* $\mathbb{N}^2$ *interpretiert (nicht definierte Werte werden durch* $f(n,x) = 0$ *dargestellt) und auf der Basis dieser Funktion zu überprüfen versucht, dass* $A(n,x) = a$ *ist. Das folgende Programm in der Orakelsignatur* $\mathcal{S} \cup \{$`Check_1`$\}$ *berechnet dann darauf aufbauend eine Funktion* $\mathrm{Check}_2(f)$*, die überprüft, dass* f *tatsächlich eine Einschränkung von* A *ist, und zwar derart, dass für jeden Funktionswert auch alle anderen, die zu seiner Berechnung nötig sind, vorhanden sind.*

```
function = input_1
output = Inc(Zero())
i = Zero()
loop Lg(function) times{
  a = Component(function,i)
  if a then {
    if Not(Justified(n,x,a,function)) then {
      output = Zero()
    }
  }
  i = Inc(i)
}
```

Damit können wir A' *in der Orakelsignatur* $\mathcal{S} \cup \{$`Check_2`$\}$ *wie folgt programmieren.*

```
n = Val(input_1)
x = Val(input_2)
f = Val(input_3)
output = Zero()
if Check_2(f) then {
  output = Component(f,J(n,x))
}
```

Überzeugen Sie sich, dass die so programmierte Funktion $A'(n, x, f)$ auch tatsächlich die eingangs genannten Eigenschaften hat.

Übungsaufgabe 1.14 *Geben Sie eine Tabelle der exakten Werte von $2 \uparrow^{n-2} x$ an für alle Paare $(n, x) \in \mathbb{N} \times \{0, 1, \ldots, 10\}$, für die das praktikabel ist. Erweitern Sie die Tabelle, indem Sie, soweit praktikabel, für große Funktionswerte die Zahl der Dezimalstellen grob abschätzen ($2^{10} = 1024 \approx 10^3$).*

Übungsaufgabe 1.15 *Sei n die kleinste natürliche Zahl, die man mit einem deutschen Satz von höchstens zwanzig Wörtern definieren kann. Ist n gerade oder ungerade?*