

Grundbegriffe der mathematischen Logik

Hans Adler

27. Juni 2011

Inhaltsverzeichnis

1	Primitive Rekursivität	2
1.1	Primitiv rekursive Funktionen	2
1.2	LOOP-Programme	4
1.3	Cantorsche Paarungsfunktion und simultane primitive Rekursion	8
1.4	Mehr zu LOOP-Programmen	10
2	Rekursivität	14
2.1	Hyperoperatoren	14
2.2	Rekursive Funktionen und Mengen	18
2.3	GOTO-Programme	21
2.4	Codierung von GOTO-Programmen	24
2.5	Kleenesche Normalform	25
2.6	Rekursiv aufzählbare Mengen	27
3	Logik	28
3.1	Strings und Sprachen	28
3.2	Signaturen, Strukturen und Terme	30
3.3	Syntax der Prädikatenlogik 1. Stufe	32
3.4	Semantik der Prädikatenlogik	34
3.5	Beweisbarkeit	35
3.6	Vollständigkeitssatz	38
3.7	Unvollständigkeitssatz	42

Kapitel 1

Primitive Rekursivität

1.1 Primitiv rekursive Funktionen

Gegeben ein beliebiges Polynom $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ mit ganzzahligen Koeffizienten $a_0, a_1, \dots, a_{n-1} \in \mathbb{Z}$, 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) 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 $p(x, y, z, \dots)$ hat entweder eine Nullstelle oder nicht. Daher hat es wenig Sinn, die Frage auf ein einzelnes Polynom einzuschränken.¹ 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.² Wir werden uns außerdem ein anderes Problem (das Halteproblem) anschauen, von dem wir relativ leicht zeigen können, dass es nicht algorithmisch lösbar ist. Zuvor schauen wir uns aber in diesem Kapitel ein eng verwandtes Thema an.

¹Für 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.

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

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

- F enthält die konstante Nullfunktion $0: \mathbb{N}^0 \rightarrow \mathbb{N}$.³
- F enthält die durch $S(n) = n + 1$ definierte Nachfolgerfunktion $S: \mathbb{N} \rightarrow \mathbb{N}$.
- F enthält für alle $i, k \in \mathbb{N}$ mit $i \leq k$ die durch $\pi_i^k(x_1, \dots, x_k) = x_i$ gegebene Projektionsfunktion $\pi_i^k: \mathbb{N}^k \rightarrow \mathbb{N}$.
- F ist abgeschlossen unter Zusammensetzung. Das heißt, wenn (für $\ell \in \mathbb{N}$) die Funktionen $f: \mathbb{N}^\ell \rightarrow \mathbb{N}$ und $g_1, \dots, g_\ell: \mathbb{N}^k \rightarrow \mathbb{N}$ alle in F liegen, dann liegt auch die durch

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

gegebene Funktion $h: \mathbb{N}^k \rightarrow \mathbb{N}$ in F .

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

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

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

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

Übungsaufgabe 1.1 Ein Hüllenoperator auf einer Menge X ist eine Abbildung $F: \mathcal{P}(X) \rightarrow \mathcal{P}(X)$, welche die folgenden Bedingungen erfüllt:⁴

- (**extensiv**) $F(A) \supseteq A$ für alle $A \subseteq X$;
- (**monoton**) $F(A) \subseteq F(B)$ für alle $A \subseteq B \subseteq X$;
- (**idempotent**) $F(F(A)) = F(A)$ für alle $A \subseteq X$.

Eine Menge $A \subseteq X$ heißt abgeschlossen unter einer Abbildung $F: \mathcal{P}(X) \rightarrow \mathcal{P}(X)$, falls $F(A) = A$ gilt.

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

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

Proposition 1.3 Die folgenden Funktionen sind primitiv rekursiv:

- $f: \mathbb{N}^2 \rightarrow \mathbb{N}$, $f(x, y) = x + y$.
- $f: \mathbb{N}^2 \rightarrow \mathbb{N}$, $f(x, y) = x \cdot y$.

³Eine nullstellige Funktion $f: \mathbb{N}^0 \rightarrow \mathbb{N}$ ist im Wesentlichen dasselbe wie eine Konstante, in diesem Fall also die Konstante $0 \in \mathbb{N}$.

⁴ $\mathcal{P}(X) = \{A \mid A \subseteq X\}$ ist die Potenzmenge von X .

- $f: \mathbb{N}^2 \rightarrow \mathbb{N}, f(x, y) = x^y.$
- $f: \mathbb{N} \rightarrow \mathbb{N}, f(x) = x! = 1 \cdot 2 \cdot \dots \cdot x.$

Übungsaufgabe 1.2 Die folgenden Funktionen sind primitiv rekursiv:

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

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

1.2 LOOP-Programme

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. Zum Beispiel: `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()`
- `y = Val(x)`
- `y = Inc(x)`
- `y = Dec(x)`

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 Variablen 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)
```

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 *Schleifenkörper* bezeichnen. Zum Beispiel schreibt das folgende LOOP-Programm die Summe der Werte von x und y ins Register z .

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

Die Schleifen machen eigentlich den **Zero**-Befehl überflüssig, denn man kann $y = \text{Zero}()$ 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. 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)
  }
}
```

Übungsaufgabe 1.3 *Jedes LOOP-Programm endet nach endlich vielen Schritten. (Das ist nicht selbstverständlich. Es gilt sogar: Wenn eine Programmiersprache diese Eigenschaft hat, dann gibt es eine durch Computer berechenbare Funktion, die nicht in dieser Sprache berechenbar ist.)*

Übungsaufgabe 1.4 Zusätzlich ist auch der **Val**-Befehl eigentlich überflüssig. Wie kann man ihn ersetzen? Was ist dabei zu beachten? (Von den 4 einfachen Befehlen genügen also **Inc** und **Dec**.)

Übungsaufgabe 1.5 Der **Dec**-Befehl und der **Val**-Befehl kann man weglassen, wenn man alle anderen hat. (Von den vier einfachen Befehlen genügen also **Inc** und **Zero**.)

Definition 1.4 Sei $k \in \mathbb{N}$. Eine Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ heißt LOOP-berechenbar, falls 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, \dots, x_k stehen hat sowie in allen anderen Registern die Zahl 0, dann steht am Ende im Register **output** die Zahl $f(x_1, x_2, \dots, x_k)$.

Übungsaufgabe 1.6 Die durch $f(x, y) = x^y$ gegebene Funktion $f: \mathbb{N}^2 \rightarrow \mathbb{N}$ ist LOOP-berechenbar.

Übungsaufgabe 1.7 Die durch $f(x) = x! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot x$ gegebene Fakultätsfunktion $f: \mathbb{N} \rightarrow \mathbb{N}$ ist LOOP-berechenbar.

Übungsaufgabe 1.8 Es gibt eine Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$, die nicht LOOP-berechenbar ist. Es gibt sogar überabzählbar viele Funktionen $f: \mathbb{N} \rightarrow \mathbb{N}$, die nicht LOOP-berechenbar sind.

Übungsaufgabe 1.9 Man könnte zusätzlich noch die folgende Art von Anweisung in LOOP-Programmen erlauben.

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

Der Programmteil, für den die drei Punkte stehen, würde dann übersprungen werden, falls x den Wert Null hat, sonst aber einmal ausgeführt werden. Zeigen Sie, dass jede Funktion, die durch ein in diesem Sinne erweitertes LOOP-Programm berechenbar ist, auch durch ein gewöhnliches LOOP-Programm berechenbar ist. (Hinweis: Es hilft, zu zeigen, dass die Funktion, die 0 auf 0 und alle anderen Zahlen auf 1 abbildet, durch ein LOOP-Programm berechenbar ist. Aber noch einfacher geht es direkt.)

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

Beweis 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`, `output_2`, ... nicht vorkommen. Gegeben $f: \mathbb{N}^\ell \rightarrow \mathbb{N}$ und $g_1, \dots, g_\ell: \mathbb{N}^k \rightarrow \mathbb{N}$, seien P_1, \dots, P_ℓ LOOP-Programme, die g_1, \dots, g_ℓ berechnen und die genannten zusätzlichen Bedingungen erfüllen. Dann erhält man ein LOOP-Programm, das die durch $h(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_\ell(x_1, \dots, x_k))$ 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.) ■

1.3 Cantorsche Paarungsfunktion und simultane primitive Rekursion

Proposition 1.6 *Die folgenden Funktionen sind primitiv rekursiv:*

- $\text{not}(x) = \begin{cases} 0 & \text{falls } x > 0 \\ 1 & \text{sonst} \end{cases}$
- $\text{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{sonst} \end{cases}$
- $\text{gt}(x, y) = \begin{cases} 1 & \text{falls } x > y \\ 0 & \text{sonst} \end{cases}$
- $\text{eq}(x, y) = \begin{cases} 1 & \text{falls } x = y \\ 0 & \text{sonst} \end{cases}$
- $\text{min}(x, y)$

Beweis $\text{not}(x) = 1 - x$. $\text{sgn}(x) = \text{not}(\text{not}(x))$. $\text{gt}(x, y) = \text{sgn}(x - y)$. $\text{eq}(x, y) = \text{gt}(x + 1, y) \cdot \text{gt}(y + 1, x)$. $\text{min}(x, y) = x \cdot \text{gt}(y, x) + y \cdot \text{gt}(x, y) + x \cdot \text{eq}(x, y)$. ■

Übungsaufgabe 1.10 *Wenn $f_1, f_2, g: \mathbb{N}^k \rightarrow \mathbb{N}$ 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 $h: \mathbb{N}^k \rightarrow \mathbb{N}$ primitiv rekursiv.

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 1.7 *Wenn $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ und $g: \mathbb{N}^k \rightarrow \mathbb{N}$ primitiv rekursiv sind, dann ist auch die durch*

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

definierte Funktion $h: \mathbb{N}^k \rightarrow \mathbb{N}$ primitiv rekursiv.

Beweis Wir zeigen zunächst, dass die durch $h'(\bar{x}, z) = \min\{y \leq z \mid y = z \text{ oder } f(\bar{x}, y) > 0\}$ definierte Funktion $h': \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ 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 das nächste Kapitel handelt.

Korollar 1.8 Sei $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ primitiv rekursiv. Die durch

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

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

Korollar 1.9 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 $\lceil \frac{x}{y} \rceil = \min\{z \leq x \mid z = x \text{ oder } y \cdot z \geq x\}$. ■

Übungsaufgabe 1.11 Die Funktion $f(x, y) = \begin{cases} \lfloor \frac{x}{y} \rfloor & \text{falls } y > 0 \\ 0 & \text{sonst} \end{cases}$ ist primitiv rekursiv.

Satz 1.10 (Cantorsche Paarungsfunktion) Die Funktion

$$J: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$(x, y) \mapsto J(x, y) = \frac{(x+y)(x+y+1)}{2} + y$$

ist eine primitiv rekursive Bijektion zwischen \mathbb{N}^2 und \mathbb{N} . Die Komponenten der Umkehrfunktion sind ebenfalls primitiv rekursiv.

Beweis Dass J eine Bijektion ist, wurde in einer Übungsaufgabe gezeigt. Da J aus primitiv rekursiven Funktionen zusammengesetzt ist, ist J auch primitiv rekursiv. Seien $L: \mathbb{N} \rightarrow \mathbb{N}$ und $R: \mathbb{N} \rightarrow \mathbb{N}$ 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 zeigen wir, dass $L'(z, u) = \min(L(z), u+1)$ primitiv rekursiv ist:

$$L'(z, u) = \min\{x \mid x = u+1 \text{ oder es existiert ein } y < u+1, \text{ so dass } J(x, y) = z\}$$

$$= \min\{x \mid x = u+1 \text{ oder } \min\{y \mid y = u+1 \text{ oder } J(x, y) = z\} < u+1\}.$$

Daraus erhalten wir dann $L(z) = L'(z, z+1)$, da immer $L(z) \leq z$ ist. Analog können wir auf dem Umweg über $R'(x, y) = \min(R(x), y+1)$ zeigen, dass auch R primitiv rekursiv ist. ■

Satz 1.11 Die primitiv rekursiven Funktionen sind abgeschlossen unter simultaner primitiv rekursiver Definition von mehreren Funktionen. Genauer:

Wenn die Funktionen $f_1, \dots, f_m: \mathbb{N}^k \rightarrow \mathbb{N}$ und $g_1, \dots, g_m: \mathbb{N}^{k+1+m} \rightarrow \mathbb{N}$ primitiv rekursiv sind, dann sind auch die durch

$$\begin{aligned} h_1(\bar{x}, 0) &= f_1(\bar{x}) \\ &\vdots \\ h_m(\bar{x}, 0) &= f_m(\bar{x}) \\ h_1(\bar{x}, y+1) &= g_1(\bar{x}, y, h_1(\bar{x}, y), \dots, h_m(\bar{x}, y)) \\ &\vdots \\ h_m(\bar{x}, y+1) &= g_m(\bar{x}, y, h_1(\bar{x}, y), \dots, h_m(\bar{x}, y)) \end{aligned}$$

definierten Funktionen $h_1, \dots, h_m: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ primitiv rekursiv.

Beweis Im Fall $m = 0$ oder $m = 1$ ist nichts zu zeigen. Im Fall $m = 2$ wenden wir die Cantorsche Paarfunktion 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:

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

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. Der allgemeine Fall geht ähnlich. ■

1.4 Mehr zu LOOP-Programmen

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.⁵

Ein LOOP-Orakelprogramm einer gegebenen Orakelsignatur ist wie ein LOOP-Programm im vorigen Abschnitt zusammengesetzt aus LOOP-Anweisungen sowie Anweisungen der Gestalt

- $y = F()$, falls F ein nullstelliges Element der Signatur ist,
- $y = F(x_1)$, falls F ein einstelliges Element der Signatur ist,

⁵Im 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 $\{\text{Zero}, \text{Val}, \text{Inc}, \text{Dec}\}$, wobei **Zero** nullstellig ist und alle anderen Elemente einstellig sind.“

- $y = F(x_1, x_2)$, 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 $f: \mathbb{N}^k \rightarrow \mathbb{N}$. 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 oder gegenseitig aufrufen!)

Satz 1.12 *Die folgenden Mengen von Funktionen sind identisch.*

1. *Alle primitiv rekursiven Funktionen.*
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 2 beschriebenen Menge. Und nach dem folgenden Lemma sind alle in der durch 3 beschriebenen Menge liegenden Funktionen primitiv rekursiv. ■

Lemma 1.13 *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.

Der einzige interessante Fall ist, wenn das Programm aus einer großen loop-Schleife mit einem beliebigen LOOP-Programm im Innern besteht. Für das Programm im Innern 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 1.11. Sie sind daher ebenfalls primitiv rekursiv. ■

Satz 1.12 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 Cantorsche 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 1.14 Für jedes $k \in \mathbb{N} \setminus \{0\}$ definieren wir eine Bijektion $J_k: \mathbb{N}^k \rightarrow \mathbb{N}$ 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 $J_{k+1}(x_0, \dots, x_k) = J(x_0, J_k(x_1, \dots, x_k))$.

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

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

Proposition 1.15 Die folgenden Funktionen sind alle primitiv rekursiv:

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

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

Dabei wird das Tupel im Fall $i \geq k$ mit sovielen Nullen aufgefüllt, dass es die Länge $i+1$ hat.⁶

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)
}

```

⁶Das ist eine nachträgliche Korrektur, damit es sich in Kapitel 2 ausgeht.

```

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 if-then-else-Kontrollstruktur aus Übungsaufgabe 1.9. Das zweistellige Orakel **Lt** wird interpretiert durch die primitiv rekursive Funktion $\chi_{<}$, die 1 zurückgibt, falls das erste Argument echt kleiner als das zweite ist und 0 sonst. Das zweistellige Orakel **Eq** wird interpretiert durch die primitiv rekursive Funktion $\chi_{=}$, die 1 zurückgibt, falls beide Argumente gleich sind und 0 sonst. Die einstelligen Orakel **L** und **R** werden interpretiert durch die Funktionen L und R , die zusammen das Inverse der Cantorsche 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**.

■

Kapitel 2

Rekursivität

2.1 Hyperoperatoren

In den Zwanzigerjahren des 20. Jahrhunderts vermutete David Hilbert, dass die primitiv rekursiven Funktionen den intuitiven Begriff der Berechenbarkeit genau beschreiben. Schon kurze Zeit später fand jedoch Wilhelm Ackermann eine Funktion, deren Werte man mit Papier und Bleistift im Prinzip alle ausrechnen kann, die jedoch nicht primitiv rekursiv ist. Dabei handelte es sich um eine Variante der Hyperoperatoren, wie wir sie in diesem Abschnitt betrachten werden.

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 den Nachfolgeroperator nehmen (was ein bisschen künstlich ist).

$$\begin{array}{ll} x + (y + 1) = (x + y) + 1 & x + 0 = x \\ x \cdot (y + 1) = x + (x \cdot y) & x \cdot 0 = 0 \\ x^{y+1} = x \cdot x^y & x^0 = 1 \end{array}$$

Wie man sieht, sind die Definitionen im Fall $y = 0$ (rechts) jeweils unterschiedlich, aber im Fall $y' = y + 1$ (links) kann man ein einheitliches Prinzip erkennen. Die *Hyperoperatoren* erhalten wir, indem wir diese Folge nach demselben Prinzip fortsetzen, wobei wir für jeden der neuen Operatoren im Fall $y = 0$ genauso vorgehen, wie für die Exponentiation. Donald Knuth hat für die Hyperoperatoren die Schreibweise $x \uparrow^n y$ eingeführt, wobei $x \uparrow^0 y = x \cdot y$ und $x \uparrow^1 y = x^y$ ist. Die allgemeine Definition ist wie folgt.

Definition 2.1 (Hyperoperatoren in Knuthscher Pfeilschreibweise)

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

Die ursprüngliche, dreistellige Ackermannfunktion ist eine Variante der dreistelligen Funktion \uparrow . Heute arbeitet man meist mit einer vereinfachten, zweistelligen Version, die auf Rózsa Péter zurückgeht. Weil die Hyperoperatoren etwas natürlicher sind als die Ackermannfunktion(en), werden wir jedoch direkt mit

ihnen arbeiten, bzw. beim Beweis, dass \uparrow nicht primitiv rekursiv ist, mit der Funktion $A(y, n) = 2 \uparrow^n (y + 3)$, die der zweistelligen Ackermannfunktion sehr ähnlich ist.

Bemerkung 2.2 \uparrow^0 ist die Multiplikation. $\uparrow = \uparrow^1$ ist die Exponentiation $x \uparrow y = x^y$. $\uparrow\uparrow = \uparrow^2$ ist der Potentzturm, das heißt

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

Ziel dieses Abschnitts ist der Beweis von Satz 2.10.

Lemma 2.3 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: $2 \uparrow^{n+1} 1 = 2 \uparrow^n (2 \uparrow^{n+1} 0) = 2 \uparrow^n 1 = \dots = 2 \uparrow^0 1 = 2 \cdot 1 = 2$. Mit 1 können wir 2 beweisen: $2 \uparrow^{n+1} 2 = 2 \uparrow^n (2 \uparrow^{n+1} 1) = 2 \uparrow^n 2 = \dots = 2 \uparrow^0 2 = 2 \cdot 2 = 4$. Aus 2 folgt nun direkt 3: $2 \uparrow^{n+1} 3 = 2 \uparrow^n (2 \uparrow^{n+1} 2) = 2 \uparrow^n 4$. ■

Lemma 2.4 Jede der einstelligen Funktionen $2 \uparrow^n: \mathbb{N} \rightarrow \mathbb{N}$, $y \mapsto 2 \uparrow^n y$ wächst streng monoton.

Beweis Wir zeigen die Behauptung durch Induktion nach n , wobei wir den Fall $n = 1$ als Induktionsbasis nehmen. Für $n = 0$ ist $2 \uparrow^n y = 2 \uparrow^0 y = 2y$, und für $n = 1$ ist $2 \uparrow^n y = 2 \uparrow^1 y = 2^y$. Beide Funktionen wachsen bekanntlich streng monoton. Wir nehmen nun an, dass $n > 0$ ist und dass \uparrow^n streng monoton wächst.

Wir müssen zeigen, dass auch \uparrow^{n+1} streng monoton wächst. Dazu zeigen wir durch Induktion nach y , dass $2 \uparrow^{n+1} (y + 1) > 2 \uparrow^{n+1} y$ gilt. Die Induktionsbasis $y = 0$ wird durch das letzte Lemma erledigt: $2 \uparrow^{n+1} 1 = 2 > 1 = 2 \uparrow^{n+1} 0$. Angenommen also, dass $2 \uparrow^{n+1} (y + 1) > 2 \uparrow^{n+1} y$ gilt. Dann gilt auch

$$\begin{aligned} 2 \uparrow^{n+1} (y + 2) &= 2 \uparrow^n (2 \uparrow^{n+1} (y + 1)) \\ &> 2 \uparrow^n (2 \uparrow^{n+1} y) \\ &= 2 \uparrow^{n+1} (y + 1) \end{aligned}$$

Der Schritt von der ersten zur zweiten Zeile ergibt sich, weil nach der Induktionsvoraussetzung der inneren Induktion (nach y) $2 \uparrow^{n+1} (y + 1) > 2 \uparrow^{n+1} y$ gilt und die Funktion $2 \uparrow^n$ nach der Induktionsvoraussetzung der äußeren Induktion (nach n) streng monoton wächst. ■

Lemma 2.5 Jede der einstelligen Funktionen $2 \uparrow y: \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto 2 \uparrow^n y$ wächst monoton.

Beweis Wir werden die folgende Behauptung durch Induktion nach n für alle $n \in \mathbb{N}$ zeigen:

$$2 \uparrow^{n+1} y \geq 2 \uparrow^n y \text{ für alle } y \in \mathbb{N}.$$

Für die Induktionsbasis $n = 0$ müssen wir $2 \uparrow^1 y \geq 2 \uparrow^0 y$ überprüfen, das heißt, $2^y \geq 2y$. Dies zeigt man leicht durch Induktion. Wir nehmen nun an, dass die Behauptung für ein gegebenes $n \in \mathbb{N}$ gilt.

Wir müssen zeigen, dass Folgendes gilt:

$$2 \uparrow^{n+2} y \geq 2 \uparrow^{n+1} y \text{ für alle } y \in \mathbb{N}.$$

Dazu machen wir wiederum eine Induktion nach y . Der Fall $y = 0$ ist klar, da $2 \uparrow^{n+2} 0 = 2 \uparrow^{n+1} 0 = 1$ laut Definition. Angenommen also, dass $2 \uparrow^{n+2} y \geq 2 \uparrow^{n+1} y$ gilt. Dann gilt auch

$$\begin{aligned} 2 \uparrow^{n+2} (y+1) &= 2 \uparrow^{n+1} (2 \uparrow^{n+2} y) \\ &\geq 2 \uparrow^{n+1} (2 \uparrow^{n+1} y) \\ &\geq 2 \uparrow^n (2 \uparrow^{n+1} y) \\ &= 2 \uparrow^{n+1} (y+1) \end{aligned}$$

Für den Schritt von der ersten auf die zweite Zeile brauchen wir die innere Induktionsvoraussetzung sowie die nach dem vorherigen Lemma geltende Monotonie von $2 \uparrow^{n+1}$. Von der zweiten zur dritten Zeile brauchen wir die äußere Induktionsvoraussetzung. ■

Lemma 2.6 Für alle $n \in \mathbb{N}$ und alle $y \in \mathbb{N}$ mit $y \geq 3$ gilt: $2 \uparrow^{n+1} y \geq 2 \uparrow^n (y+1)$.

Beweis Für $y \geq 3$ ist $2 \uparrow^{n+1} (y-1) \geq 2 \uparrow^n (y-1) = 2y - 2 \geq y + 1$. Da die Funktion $2 \uparrow^{n+1}$ monoton wächst, folgt $2 \uparrow^{n+1} y = 2 \uparrow^n (2 \uparrow^{n+1} (y-1)) \geq 2 \uparrow^n (y+1)$. ■

Lemma 2.7 Für $n \geq 1$ ist $2 \uparrow^n (y+c) \geq 2^c \cdot (2 \uparrow^n y)$.

Beweis Wir zeigen $2 \uparrow^n (y+1) \geq 2 \cdot (2 \uparrow^n y)$ durch Induktion nach n . Die Induktionsbasis $n = 1$ ist klar: $2 \uparrow^1 (y+1) = 2^{y+1} = 2 \cdot 2^y = 2 \uparrow^1 y$. Wir nehmen nun an, dass $2 \uparrow^n (z+1) \geq 2 \cdot (2 \uparrow^n z)$ für alle $y \in \mathbb{N}$ gilt.

Wir müssen zeigen, dass für alle $y \in \mathbb{N}$ gilt: $2 \uparrow^{n+1} (y+1) \geq 2 \cdot (2 \uparrow^{n+1} y)$. Wir zeigen das durch Induktion nach y . Für die Induktionsbasis $y = 0$ stellen wir fest, dass sogar die Gleichung gilt: $2 \uparrow^{n+1} 1 = 2 = 2 \cdot (2 \uparrow^{n+1} 0)$. Für den inneren Induktionsschritt setzen wir zusätzlich voraus, dass $2 \uparrow^{n+1} (y+1) \geq 2 \cdot (2 \uparrow^{n+1} y)$ gilt, und sehen, dass gilt:

$$\begin{aligned} 2 \uparrow^{n+1} (y+2) &= 2 \uparrow^n (2 \uparrow^{n+1} (y+1)) \\ &\geq 2 \uparrow^n (2 \cdot 2 \uparrow^{n+1} y) \\ &\geq 2 \uparrow^n (2 \uparrow^{n+1} y + 1) \\ &\geq 2 \cdot (2 \uparrow^n (2 \uparrow^{n+1} y)) \\ &= 2 \cdot (2 \uparrow^{n+1} (y+1)). \end{aligned}$$

Für die erste Ungleichung verwenden wir die innere Induktionsvoraussetzung. Die zweite gilt wegen $2 \cdot 2 \uparrow^{n+1} y \geq 2 \uparrow^{n+1} y + 1$ und der Monotonie von $2 \uparrow^n$. Die dritte folgt aus der äußeren Induktionsvoraussetzung. ■

Lemma 2.8 Für alle $k, c, n \in \mathbb{N}$ gibt es ein $N \in \mathbb{N}$, so dass gilt:

$$k \cdot (2 \uparrow^n (y + c)) \leq 2 \uparrow^N (y + 3) \text{ für alle } y \in \mathbb{N}.$$

Beweis Wir können annehmen, dass $n \geq 1$ und folglich nach dem vorigen Lemma $2 \uparrow^n (y + c) \geq 2^c \cdot (2 \uparrow^n y)$ ist. Wir wählen $C \in \mathbb{N}$, $C \geq c + \log_2 k$ und zeigen:

$$\begin{aligned} k \cdot (2 \uparrow^n (y + c)) &\leq 2 \uparrow^n (y + C) \\ &\leq 2 \uparrow^{n+C-3} (y + 3). \end{aligned}$$

Die erste Ungleichung folgt aus dem vorigen Lemma, und die zweite ergibt sich durch wiederholtes Anwenden von Lemma 2.6. ■

Lemma 2.9 Für jede primitiv rekursive Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ gibt es ein $n \in \mathbb{N}$, so dass für alle $x_1, \dots, x_k \in \mathbb{N}$ gilt:

$$f(x_1, \dots, x_k) \leq 2 \uparrow^n (x_1 + \dots + x_k + 3).$$

Beweis Man überprüft das sehr leicht für die Grundfunktionen 0, S und π_i^k .

Einsetzung: Wenn das Lemma für f und g_1, \dots, g_ℓ stimmt, dann stimmt es auch für die durch $h(\bar{x}) = f(g_1(\bar{x}), \dots, g_\ell(\bar{x}))$ definierte Funktion. Wegen Lemma 2.5 können wir für den Beweis annehmen, dass dasselbe n es für die Funktionen f und g_i tut:

$$\begin{aligned} f(\bar{y}) &\leq 2 \uparrow^n (y_1 + \dots + y_\ell + 3) \\ g_1(\bar{x}) &\leq 2 \uparrow^n (x_1 + \dots + x_k + 3) \\ &\dots \\ g_\ell(\bar{x}) &\leq 2 \uparrow^n (x_1 + \dots + x_k + 3) \end{aligned}$$

Damit folgt:

$$\begin{aligned} h(\bar{x}) &= f(g_1(\bar{x}), \dots, g_\ell(\bar{x})) \\ &\leq 2 \uparrow^n \left(\sum_{i=1}^k g_i(\bar{x}) + 3 \right) \\ &\leq 2 \uparrow^n (k \cdot (2 \uparrow^n (x_1 + \dots + x_k + 3)) + 3) \\ &\leq 2 \uparrow^n (2 \uparrow^N (x_1 + \dots + x_k + 3)). \end{aligned}$$

Von der ersten bis zur dritten Zeile kommen wir mit den Induktionsannahmen für die Funktionen f und g_i und mit der Monotonie von $2 \uparrow^n$. Die Existenz eines passenden N für die letzte Ungleichung wird durch Lemma 2.8 garantiert.

Primitive Rekursion: Wenn das Lemma für f und g stimmt, dann stimmt es auch für die durch $h(\bar{x}, 0) = f(\bar{x})$ und $h(\bar{x}, y + 1) = g(\bar{x}, y, h(\bar{x}, y))$ gegebene

Funktion. Wegen Lemma 2.5 können wir für den Beweis annehmen, dass dasselbe n es für die Funktionen f und g tut:

$$\begin{aligned} f(\bar{x}) &\leq 2 \uparrow^n (x_1 + \dots + x_k + 3) \\ g(\bar{x}, y, z) &\leq 2 \uparrow^n (x_1 + \dots + x_k + y + z + 3) \end{aligned}$$

Mit Lemma 2.7 können wir uns überlegen:

$$\begin{aligned} x_1 + \dots + x_k + f(\bar{x}) + 3 &\leq 2 \uparrow^{n+1} (x_1 + \dots + x_k + 3) \\ x_1 + \dots + x_k + (y + 1) + g(\bar{x}, y, z) + 3 &\leq 2 \uparrow^{n+1} (x_1 + \dots + x_k + y + z + 3). \end{aligned}$$

Wir zeigen nun durch Induktion, dass nun sogar gilt:

$$x_1 + \dots + x_k + y + h(\bar{x}, y) + 3 \leq 2 \uparrow^{n+2} (x_1 + \dots + x_k + y + 3).$$

(Dadurch wird der Beweis des Lemmas abgeschlossen.) Die Induktionsbasis ist

$$\begin{aligned} x_1 + \dots + x_k + 0 + h(\bar{x}, 0) + 3 &= x_1 + \dots + x_k + f(\bar{x}) + 3 \\ &\leq 2 \uparrow^{n+1} (x_1 + \dots + x_k + 0 + 3). \end{aligned}$$

Die Ungleichung gilt nach Wahl von N . Der Induktionsschritt von y nach $y + 1$ geht so:

$$\begin{aligned} x_1 + \dots + x_k + (y + 1) + h(\bar{x}, y + 1) + 3 & \\ &= x_1 + \dots + x_k + (y + 1) + g(\bar{x}, y, h(\bar{x}, y)) + 3 \\ &\leq 2 \uparrow^{n+1} (x_1 + \dots + x_k + y + h(\bar{x}, y) + 3) \\ &\leq 2 \uparrow^{n+1} (2 \uparrow^{n+1} (x_1 + \dots + x_k + y + 3)) \\ &= 2 \uparrow^{n+2} (x_1 + \dots + x_k + (y + 1) + 3). \end{aligned}$$

Die erste Ungleichung gilt nach Wahl von n , und die zweite nach Induktionsvoraussetzung und Monotonie von $2 \uparrow^{n+1}$. ■

Satz 2.10 *Jede der zweistelligen Funktionen $\uparrow^n: \mathbb{N}^2 \rightarrow \mathbb{N}$, $(x, y) \mapsto x \uparrow^n y$ ist primitiv rekursiv. Die dreistellige Funktion $\uparrow: \mathbb{N}^3 \rightarrow \mathbb{N}$, $(x, y, n) \mapsto x \uparrow^n y$ ist aber nicht primitiv rekursiv.*

Beweis Die Multiplikation \uparrow^0 ist bekanntlich primitiv rekursiv. Man zeigt leicht, dass die primitive Rekursivität von \uparrow^{n+1} aus der von \uparrow^n folgt.

Wenn die dreistellige Funktion \uparrow primitiv rekursiv wäre, dann wäre auch die Funktion $f(x) = 2 \uparrow^x (x + 3) + 1$ primitiv rekursiv, und daher gäbe es nach Lemma 2.9 ein $n \in \mathbb{N}$, so dass $f(x) \leq 2 \uparrow^n (x + 3)$ für alle $x \in \mathbb{N}$. Dann wäre aber $2 \uparrow^n (n + 3) + 1 = f(n) \leq 2 \uparrow^n (n + 3)$, ein Widerspruch. ■

2.2 Rekursive Funktionen und Mengen

Definition 2.11 *Die Menge der rekursiven Funktionen ist die kleinste Menge $F \subseteq \bigcup_{k \in \mathbb{N}} \{f: \mathbb{N}^k \rightarrow \mathbb{N}\}$, welche die folgenden Bedingungen erfüllt:*

- F enthält die Nullfunktion 0 , die Nachfolgerfunktion S und die Projektionsfunktion π_i^k ($1 \leq i \leq k$).

- F ist abgeschlossen unter Zusammensetzung. Das heißt, wenn die Funktionen $f: \mathbb{N}^\ell \rightarrow \mathbb{N}$ und $g_1, \dots, g_\ell: \mathbb{N}^k \rightarrow \mathbb{N}$ alle in F liegen, dann liegt auch die durch

$$h(\bar{x}) = f(g_1(\bar{x}), \dots, g_\ell(\bar{x}))$$

gegebene Funktion $h: \mathbb{N}^k \rightarrow \mathbb{N}$ in F .

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

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

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

- F ist abgeschlossen unter μ -Rekursion. Das heißt, wenn die Funktion $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ in F liegt und außerdem für alle $\bar{x} \in \mathbb{N}^k$ ein $y \in \mathbb{N}$ existiert, so dass $f(\bar{x}, y) > 0$ ist, dann liegt auch die durch

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

gegebene Funktion $g: \mathbb{N}^k \rightarrow \mathbb{N}$ in F .

Bemerkung 2.12 Alle primitiv rekursiven Funktionen sind rekursiv.

Beweis Sei P die Menge aller primitiv rekursiven Funktionen und R die Menge aller rekursiven Funktionen. Sowohl P als auch R sind von der in Definition 1.1 betrachteten Art. Da P als die kleinste solche Menge definiert ist, gilt $P \subseteq R$. ■

Definition 2.13 Eine Menge $A \subseteq \mathbb{N}^k$ heißt rekursiv, wenn ihre charakteristische Funktion

$$\begin{aligned} \chi_A: \mathbb{N}^k &\rightarrow \mathbb{N}, \\ \bar{x} &\mapsto \begin{cases} 1 & \text{falls } \bar{x} \in A \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

rekursiv ist.

Bemerkung 2.14 Eine Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ ist genau dann rekursiv, wenn ihr Graph $\Gamma_f \subseteq \mathbb{N}^{k+1}$ rekursiv ist.

Beweis $\chi_=: \mathbb{N}^2 \rightarrow \mathbb{N}$, die charakteristische Funktion der Identität, ist sicher (primitiv) rekursiv. Wenn f rekursiv ist, dann ist wegen $\chi_{\Gamma_f}(\bar{x}, y) = \chi_=(f(\bar{x}), y)$ auch Γ_f rekursiv. Wenn umgekehrt Γ_f rekursiv ist, dann ist auch $f(\bar{x}) = \mu y (\chi_{\Gamma_f}(\bar{x}, y) > 0)$ rekursiv. ■

Proposition 2.15 Boolesche Kombinationen von rekursiven Mengen sind wieder rekursiv.

Beweis Seien $A, B \subseteq \mathbb{N}^k$ rekursive Mengen. Dann gilt:

$$\begin{aligned}\chi_{A \cup B}(\bar{x}) &= \text{sgn}(\chi_A(\bar{x}) + \chi_B(\bar{x})) \\ \chi_{A \cap B}(\bar{x}) &= \chi_A(\bar{x}) \cdot \chi_B(\bar{x}) \\ \chi_{\mathbb{N} \setminus A}(\bar{x}) &= 1 - \chi_A(\bar{x})\end{aligned}$$

Daher sind die charakteristischen Funktionen von $A \cup B$, $A \cap B$ und $\mathbb{N} \setminus A$ rekursiv. ■

Proposition 2.16 *Durch Einsetzung von rekursiven Funktionen erhält man aus einer rekursiven Menge ebenfalls wieder eine rekursive Menge.*

Proposition 2.17 *Wenn $f, g: \mathbb{N}^k \rightarrow \mathbb{N}$ rekursive Funktionen sind und $B \subseteq \mathbb{N}^k$ eine rekursive Menge ist, dann ist auch*

$$h(\bar{x}) = \begin{cases} f(\bar{x}) & \text{falls } \bar{x} \in B \\ g(\bar{x}) & \text{sonst} \end{cases}$$

eine rekursive Funktion.

Proposition 2.18 *Wenn $A \in \mathbb{N}^{k+1}$ und $f: \mathbb{N}^k \rightarrow \mathbb{N}$ rekursiv sind, dann sind auch folgende Mengen rekursiv:*

$$\begin{aligned}\{(\bar{x}, z) \in \mathbb{N}^{k+1} \mid \exists y < z ((\bar{x}, y) \in A)\} \\ \{(\bar{x}, z) \in \mathbb{N}^{k+1} \mid \forall y < z ((\bar{x}, y) \in A)\}\end{aligned}$$

Jetzt können wir zeigen, dass es rekursive Funktionen gibt, die nicht primitiv rekursiv sind.

Satz 2.19 *Die dreistellige Funktion $\uparrow: \mathbb{N}^3 \rightarrow \mathbb{N}$, $(x, y, n) \mapsto x \uparrow^n y$ ist rekursiv.*

Beweis Die Funktion \uparrow ist in einem intuitiven Sinn berechenbar. Die Beweisidee besteht darin, eine solche Berechnung, aus der sich $x \uparrow^n y = z$ ergibt, formal zu beschreiben. Die Berechnung wird aus Quadrupeln $(x, y, n, z) \in \Gamma_{\uparrow}$ bestehen, die wir durch die Zahlen $\langle x, y, n, z \rangle$ codieren. Die Quadrupel bilden insgesamt ein endliches Tupel, das wir wiederum mit der Funktion $\langle \cdot \rangle$ codieren. Für jedes Quadrupel der Berechnung fordern wir, dass entsprechend der rekursiven Definition von \uparrow auch diejenigen Quadrupel in der Berechnung vorhanden sind, aus denen es folgt. Genauer:

$$\begin{aligned}S = \{ \langle x, y, n, z \rangle \mid n = 0 \text{ und } x \cdot y = z \} \\ \cup \{ \langle x, y, n, z \rangle \mid n > 0 \text{ und } y = 0 \text{ und } z = 1 \} \subset \mathbb{N},\end{aligned}$$

$$R = \{ (\langle x, y + 1, n + 1, z \rangle, \langle x, y, n + 1, u \rangle, \langle x, u, n, z \rangle) \mid x, y, n, z, u \in \mathbb{N} \} \subset \mathbb{N}^3.$$

$$\begin{aligned}B = \{ b \in \mathbb{N} \mid \forall i < \text{Lg}(b) (\\ \text{Component}(b, i) \in S \text{ oder } \exists s < \text{Lg}(b) \exists t < \text{Lg}(b) (\\ (\text{Component}(b, i), \text{Component}(b, s), \text{Component}(b, t)) \in R \\)) \} \subseteq \mathbb{N}.\end{aligned}$$

Man überlegt sich leicht, dass S , R und B rekursiv sind, und dass B die Menge der Berechnungen b im obigen Sinne ist. Daher können wir die Funktion \uparrow wie folgt beschreiben:

$$\begin{aligned}\beta(x, y, n) &= \mu b \left(b \in B \text{ und } \exists i < \text{Lg}(b) \exists z [< b] \left(\text{Component}(b, i) = \langle x, y, n, z \rangle \right) \right) \\ \gamma(x, y, n) &= \mu i \left(\exists z \left(\text{Component}(b, i) = \langle x, y, n, z \rangle \right) \right) \\ x \uparrow^n y &= \text{Component} \left(\text{Component}(\beta(x, y, n), \gamma(x, y, n)), 3 \right)\end{aligned}$$

$\beta(x, y, n)$ gibt uns den numerisch kleinsten Code einer Berechnung, in der das Quadrupel $\langle x, y, n, x \uparrow^n y \rangle$ vorkommt.¹ Dann müssen wir nur noch in der Berechnung den Wert von $x \uparrow^n y$ nachschlagen. Man überlegt sich jetzt leicht, dass β und folglich auch \uparrow rekursiv ist. ■

Übungsaufgabe 2.1 Eine Menge $A \subseteq \mathbb{N}^k$ heißt *primitiv rekursiv*, wenn ihre charakteristische Funktion primitiv rekursiv ist. Sei $f: \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion, deren Graph Γ_f primitiv rekursiv ist. Zeigen Sie, dass f rekursiv ist, aber im Allgemeinen nicht primitiv rekursiv.

2.3 GOTO-Programme

GOTO-Programme sind definiert wie LOOP-Programme, nur dass wir an Stelle von Schleifen bedingte Sprunganweisungen erlauben. Eine Neuerung hierbei ist, dass ein GOTO-Programm aus mit 0 beginnend fortlaufend durchnummerierten Zeilen besteht. In jeder Zeile kann wie üblich einer der folgenden vertrauten Befehle stehen.

- $y = \text{Zero}()$
- $y = \text{Val}(x)$
- $y = \text{Inc}(x)$
- $y = \text{Dec}(x)$

Allerdings gibt es in GOTO-Programmen keine Schleifen im bisherigen Sinne. Diese sind durch eine flexiblere Anweisung der Art **if x goto 42** ersetzt. Bei der Ausführung des Programms bewirkt diese Anweisung einen Sprung in die Zeile 42, falls der Inhalt des durch x bezeichneten Registers nicht die Zahl 0 ist. Dabei kann natürlich an Stelle von x wieder ein beliebiger Variablenname stehen und an Stelle von 42 eine beliebige natürliche Zahl. Ein GOTO-Programm hält an, sobald es eine Zeile ohne Anweisung erreicht.

Die GOTO-Programme erlauben es uns, sogenannten *Spagetti-Code* zu schreiben, also im schlimmsten Fall Programme, die ohne erkennbare Struktur kreuz und quer springen. Man kann aber zeigen (siehe Übungsaufgaben), dass die einzige wirklich wesentliche Erweiterung gegenüber LOOP-Programmen darin besteht, dass wir nun Schleifen implementieren können, bei denen nicht mehr am Anfang feststeht, wie oft sie durchlaufen werden. Insbesondere können wir jetzt auch Schleifen implementieren, die gar nicht enden:

¹Es ist nicht wichtig, dass es der kleinste Code ist. Es kommt nur darauf an, dass es immer eine solche Berechnung gibt und wir von β den Code einer solchen erhalten.

```

0  false = Zero()
1  true = Inc(false)
2  output = Zero()
3  output = Inc(output)
4  if true goto 3

```

Dieses Programm läuft unendlich lange, und bei jedem Durchgang durch den Schleifenkörper ändert sich der Wert von **output**. Daher hat es keinen Sinn, diesem Programm eine Funktion zuzuschreiben, die dadurch berechnet wird.

Wir „lösen“ dieses Problem, indem wir von *partiellen (k -stelligen) Funktionen* sprechen, das heißt, Funktionen, deren Definitionsbereich nur eine Teilmenge von \mathbb{N}^k ist.

Definition 2.20 *Eine partielle Funktion $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist eine Funktion $f: X \rightarrow \mathbb{N}$, wobei $X \subseteq \mathbb{N}^k$.*

Wenn einfach nur von einer „Funktion“ die Rede ist, ist aber nach wie vor immer eine totale Funktion gemeint.

Definition 2.21 *Sei $k \in \mathbb{N}$. Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ heißt GOTO-berechenbar, falls es ein GOTO-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 $x_1, x_2, \dots, x_k \in \mathbb{N}$ stehen hat und in allen anderen Registern die Zahl 0, dann passiert Folgendes:*

- Falls $f(x_1, x_2, \dots, x_k)$ definiert ist, dann hält das Programm an, und am Ende steht im Register **output** die Zahl $f(x_1, x_2, \dots, x_k)$.
- Falls $f(x_1, x_2, \dots, x_k)$ nicht definiert ist, dann hält das Programm nicht an.

Zwei GOTO-Programme heißen äquivalent, wenn sie für jedes $k \in \mathbb{N}$ dieselbe partielle k -stellige Funktion $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ berechnen.

Das folgende GOTO-Programm berechnet beispielsweise das Produkt von zwei natürlichen Zahlen.

```

0  false = Zero()
1  true = Inc(false)
2  i = Val(input_1)
3  output = Zero()
4  if i goto 6
5  if true goto 14
6  j = Val(input_2)
7  if j goto 9
8  if true goto 12
9  output = Inc(output)
10 j = Dec(j)
11 if true goto 7
12 i = Dec(i)
13 if true goto 4

```

Das Programm ist leichter zu verstehen, wenn man sich klarmacht, dass es sich einfach um eine Übersetzung des folgenden LOOP-Programms handelt.

```

output = Zero()
do input_1 times {
  do input_2 times {
    output = Inc(output)
  }
}

```

Übungsaufgabe 2.2 Schreiben Sie ein GOTO-Programm, das die dreistellige Hyperoperatorfunktion berechnet. Können Sie das auch mit einem LOOP-Programm? Warum ist das so viel schwerer?

GOTO-Orakelprogramme sind auf die offensichtliche Weise definiert.

Lemma 2.22 Jede Funktion, die durch ein GOTO-Orakelprogramm berechenbar ist, wobei jeder Orakelname durch eine GOTO-berechenbare Funktion interpretiert wird, ist selbst GOTO-berechenbar.

Lemma 2.23 Jede rekursive Funktion ist GOTO-berechenbar.

Beweis Der Beweis geht im Grunde genauso wie der Beweis von Satz 1.5. Wir müssen zeigen, dass die GOTO-berechenbaren (totalen) Funktionen eine Menge F wie in der Definition der rekursiven Funktionen bilden. Wir führen hier nur den Abschluss unter μ -Rekursion aus.

Sei also eine GOTO-berechenbare Funktion $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ gegeben, so dass $g(\bar{x}) = \mu y (f(\bar{x}, y) > 0)$ ebenfalls eine totale Funktion ist.

```

0  y = Zero()
1  z = F(input_1, ..., input_k, y)
2  y = Inc(y)
3  if z then goto 5
4  if y then goto 1
5  output = Dec(y)

```

Man macht sich leicht klar: Wenn man den $(k + 1)$ -stelligen Orakelnamen F in im obigen GOTO-Orakelprogramm durch f interpretiert, dann berechnet das Programm g . ■

Übungsaufgabe 2.3 WHILE-Programme sind definiert wie LOOP-Programme, nur dass zusätzlich noch die folgende neue Art von Schleifen erlaubt ist.

```

while x do {
  ...
}

```

Wieder kann x durch einen beliebigen Variablennamen ersetzt werden, und die drei Punkte stehen für ein beliebiges WHILE-Programm. Der Schleifenkörper wird nur dann ausgeführt, wenn der Wert der Variable x nicht Null ist. Nach jeder Ausführung des Schleifenkörpers wird x aber erneut inspiziert. Wenn der Wert dann immer noch nicht Null ist, wird die Schleife erneut ausgeführt.

WHILE-Programme sind näher an modernen Programmiersprachen als GOTO-Programme. Sie sind in der Regel übersichtlicher und leichter zu verstehen.

Zeigen Sie: Zu jedem WHILE-Programm gibt es ein äquivalentes WHILE-Programm, in welchem nur die neue Art von Schleifen vorkommt. Zu jedem WHILE-Programm gibt es ein äquivalentes GOTO-Programm.

Übungsaufgabe 2.4 Betrachten Sie Kontrollstrukturen der folgenden Art, mit der offensichtlichen Bedeutung:

```

if  $x$  then {
    ...
} else {
    ...
}

```

Wie kann man WHILE-Programme in der so erweiterten Sprache in GOTO-Programme übersetzen? Wie kann man sie in gewöhnliche WHILE-Programme übersetzen?

Übungsaufgabe 2.5 Zu jedem GOTO-Programm gibt es ein äquivalentes WHILE-Programm. (Hinweis: Es geht mit nur einer einzigen WHILE-Schleife! Benutzen Sie Orakel und Aufgabe 2.4, und führen Sie in einer Variable Buch über die Zeile des GOTO-Programms, in der sich Ihr WHILE-Programm gerade befindet.)

2.4 Codierung von GOTO-Programmen

Ab jetzt werden wir stillschweigend voraussetzen, dass in jedem GOTO-Programm außer der Variable **output** nur Variablen der Form **input_1**, **input_2**, **input_3**, ..., **input_9**, **input_10**, **input_11**, ...vorkommen. Dadurch können wir in offensichtlicher Weise jeden Variablennamen s durch eine natürliche Zahl $\ulcorner s \urcorner$ codieren: $\ulcorner \text{output} \urcorner = 0$, $\ulcorner \text{input}_1 \urcorner = 1$, $\ulcorner \text{input}_2 \urcorner = 2$ usw.

Bemerkung 2.24 Jedes GOTO-Programm kann in ein äquivalentes GOTO-Programm transformiert werden, das diese Bedingung erfüllt.

Eine Zeile eines GOTO-Programms codieren wir wie folgt als natürliche Zahl:

- $\ulcorner y = \text{Zero}() \urcorner = \langle 1, 0, \ulcorner y \urcorner \rangle$
- $\ulcorner y = \text{Val}(x) \urcorner = \langle 2, \ulcorner x \urcorner, \ulcorner y \urcorner \rangle$
- $\ulcorner y = \text{Inc}(x) \urcorner = \langle 3, \ulcorner x \urcorner, \ulcorner y \urcorner \rangle$
- $\ulcorner y = \text{Dec}(x) \urcorner = \langle 4, \ulcorner x \urcorner, \ulcorner y \urcorner \rangle$
- $\ulcorner \text{if } x \text{ goto } \ell \urcorner = \langle 5, \ulcorner x \urcorner, \ell \rangle$

Hierbei stehen x , y für Variablennamen, die den eben beschriebenen Einschränkungen entsprechen (so dass $\ulcorner x \urcorner$ und $\ulcorner y \urcorner$ auch definiert sind), und ℓ steht für eine natürliche Zahl bzw. im Programmcode für ihre Dezimaldarstellung.

Durch diese Codierung sind wir richtigen Mikroprozessoren nähergekommen. Wir können uns einen idealisierten Mikroprozessor vorstellen, mit fortlaufend nummerierten Registern, deren jedes eine natürliche Zahl fassen kann. Wir codieren ein Programm mit L Zeilen durch die Zahl $\mathcal{P} = \langle p_0, p_1, \dots, p_{L-1} \rangle$, wobei $p_i \in \mathbb{N}$ der Code für die i -te Zeile ist.

Übungsaufgabe 2.6 Für jedes $k \in \mathbb{N}$ gibt es ein WHILE-Programm U mit der folgenden Eigenschaft. Für jedes GOTO-Programm P existiert eine natürliche Zahl $p \in \mathbb{N}$, so dass gilt: Wenn $f_U: \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ die durch U berechnete partielle Funktion ist, dann ist die durch P berechnete partielle Funktion f_P gegeben durch $f_P(x_1, \dots, x_k) = f_U(x_1, \dots, x_k, p)$. (Hinweis: Wie Aufgabe 2.5. Benutzen Sie die beschriebene Codierung, um das ganze Programm in einer einzigen Variable halten zu können.)

Wir schauen uns nun die Ausführung eines Programms im Detail an. Dabei interessieren wir uns für den Zeitpunkt, wenn der Mikroprozessor gerade die Zeile gewechselt, die neue Zeile aber noch nicht ausgeführt hat. Zu jedem solchen Zeitpunkt enthält jedes Register i eine natürliche Zahl $x_i \in \mathbb{N}$ und der Programmzähler befindet sich in der als nächstes auszuführenden Zeile $\ell \in \mathbb{N}$. Wir codieren den Zustand der Register durch die Zahl $\mathcal{R} = \langle x_0, x_1, \dots, x_{R-1} \rangle$ und den gesamten Zustand des Mikroprozessors durch $\mathcal{Z} = \langle \ell, \mathcal{R} \rangle$.

Wir benutzen von jetzt an die Schreibweise $(x)_i = \text{Component}(x, i)$. Der jeweils nächste Zustand ist dann durch die folgende Funktion gegeben:

$$N_{\mathcal{P}}(\mathcal{Z}) = \begin{cases} \langle \ell + 1, \text{Replace}(\mathcal{R}, y, 0) \rangle & \text{falls } c = 1 \\ \langle \ell + 1, \text{Replace}(\mathcal{R}, y, (\mathcal{R})_x) \rangle & \text{falls } c = 2 \\ \langle \ell + 1, \text{Replace}(\mathcal{R}, y, (\mathcal{R})_x + 1) \rangle & \text{falls } c = 3 \\ \langle \ell + 1, \text{Replace}(\mathcal{R}, y, (\mathcal{R})_x - 1) \rangle & \text{falls } c = 4 \\ \langle \ell + 1, \mathcal{R} \rangle & \text{falls } c = 5 \text{ und } (\mathcal{R})_x = 0 \\ \langle y, \mathcal{R} \rangle & \text{falls } c = 5 \text{ und } (\mathcal{R})_x \neq 0 \\ \mathcal{Z} & \text{sonst,} \end{cases}$$

wobei wir zur besseren Lesbarkeit die folgenden Abkürzungen verwendet haben:

$$\begin{array}{lll} \ell = (\mathcal{Z})_0 & \mathcal{R} = (\mathcal{Z})_1 & C = (\mathcal{P})_\ell \\ c = (C)_0 & x = (C)_1 & y = (C)_2 \end{array}$$

Diese Funktion $N: \mathbb{N}^2 \rightarrow \mathbb{N}, (\mathcal{P}, \mathcal{Z}) \mapsto N_{\mathcal{P}}(\mathcal{Z})$ ist primitiv rekursiv. Natürlich ist auch für jedes $k \in \mathbb{N}$ die durch

$$A^k(x_1, \dots, x_k) = \langle 0, \langle 0, x_1, \dots, x_k \rangle \rangle$$

gegebene Funktion $A^k: \mathbb{N}^k \rightarrow \mathbb{N}$ primitiv rekursiv. Diese Funktion gibt uns den anfänglichen Zustand bei einer Berechnung, in Abhängigkeit von den Eingabewerten.

Wir halten fest, was wir bisher herausgefunden haben.

Lemma 2.25 *Es gibt primitiv rekursive Funktionen $N: \mathbb{N}^2 \rightarrow \mathbb{N}$ und $A^k: \mathbb{N}^k \rightarrow \mathbb{N}$ sowie für jedes GOTO-Programm eine Zahl $\mathcal{P} \in \mathbb{N}$, so dass Folgendes gilt.*

Sei $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ die durch das Programm berechnete partielle Funktion und seien $x_1, \dots, x_k \in \mathbb{N}$. Wenn s die kleinste Zahl ist, so dass für $\mathcal{Z} = N_{\mathcal{P}}^s(A^k(x_1, \dots, x_k))$ und $\ell = (\mathcal{Z})_0$ die Bedingung $(\mathcal{P})_\ell = 0$ erfüllt ist, dann ist $f(x_1, \dots, x_k) = (\mathcal{R})_0$, wobei $\mathcal{R} = (\mathcal{Z})_1$. Wenn es keine solche Zahl s gibt, dann ist auch $f(x_1, \dots, x_k)$ nicht definiert.

2.5 Kleenesche Normalform

Satz 2.26 *Es gibt eine primitiv rekursive Funktion $\mathcal{U}: \mathbb{N} \rightarrow \mathbb{N}$ und für jedes $k \in \mathbb{N}$ eine primitiv rekursive Funktion $\mathcal{T}_k: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$, so dass Folgendes gilt:*

Für jede rekursive Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ existiert eine Zahl $e \in \mathbb{N}$, so dass für alle $x_1, \dots, x_k \in \mathbb{N}$ ein $y \in \mathbb{N}$ existiert, so dass $\mathcal{T}^k(\mathcal{P}, x_1, \dots, x_k, y) > 0$ ist, und es gilt

$$f(x_1, \dots, x_k) = \mathcal{U}\left(\mu y (\mathcal{T}_k(e, x_1, \dots, x_k, y) > 0)\right).$$

Da alle rekursiven Funktionen GOTO-berechenbar sind, folgt der Satz sofort aus dem folgenden Lemma.

Lemma 2.27 *Es gibt eine primitiv rekursive Funktion $\mathcal{U}: \mathbb{N} \rightarrow \mathbb{N}$ und für jedes $k \in \mathbb{N}$ eine primitiv rekursive Funktion $\mathcal{T}_k: \mathbb{N}^{k+2} \rightarrow \mathbb{N}$, so dass Folgendes gilt:*

Wenn \mathcal{P} ein GOTO-Programm codiert, dann hat die durch das Programm definierte partielle k -stellige Funktion $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ die Form

$$f(x_1, \dots, x_k) = \mathcal{U}\left(\mu y (\mathcal{T}_k(\mathcal{P}, x_1, \dots, x_k, y) > 0)\right).$$

(Insbesondere existiert $\mu y (\mathcal{T}_k(\mathcal{P}, x_1, \dots, x_k, y) > 0)$ genau dann, wenn $f(x_1, \dots, x_k)$ definiert ist.)

Beweis Mit Hilfe der Funktionen aus Lemma 2.25 definieren wir

$$\mathcal{T}_k^*(\mathcal{P}, x_1, \dots, x_k, y, 0) = \begin{cases} 1 & \text{falls } (y)_0 = A^k(x_1, \dots, x_k) \\ 0 & \text{sonst,} \end{cases}$$

$$\mathcal{T}_k^*(\mathcal{P}, x_1, \dots, x_k, y, n+1) = \begin{cases} 1 & \text{falls } \mathcal{T}_k^*(\mathcal{P}, x_1, \dots, x_k, y, n) > 0 \\ & \text{und } (y)_{n+1} = N_{\mathcal{P}}((y)_n) \\ 0 & \text{sonst.} \end{cases}$$

Man sieht leicht, dass die so definierte Funktion primitiv rekursiv ist. Außerdem macht man sich leicht klar, dass genau dann $\mathcal{T}_k^*(\mathcal{P}, x_1, \dots, x_k, y, \lg y) > 0$ gilt, wenn y eine anfängliche Folge von Zuständen codiert, die bei der Ausführung des Programms mit Startwerten (x_1, \dots, x_k) in auftreten. Wir definieren nun \mathcal{T}_k selbst wie folgt.

$$\mathcal{T}_k(\mathcal{P}, x_1, \dots, x_k, y) = \begin{cases} 1 & \text{falls } \mathcal{T}_k^*(\mathcal{P}, x_1, \dots, x_k, y, \lg y) > 0 \text{ und } (\mathcal{P})_\ell = 0, \\ & \text{wobei } \ell = (\mathcal{Z})_0 \text{ und } \mathcal{Z} = (y)_{\lg y - 1} \\ 0 & \text{sonst,} \end{cases}$$

Wenn das Programm bei den Eingaben (x_1, \dots, x_k) anhält, dann gibt es ein y , so dass $\mathcal{T}_k(\mathcal{P}, x_1, \dots, x_k, y) > 0$ ist. (Und wenn es das Programm nicht anhält, gibt es gar kein solches y .) Für jedes solche y ist $\mathcal{Z} = (y)_{\lg y - 1}$ der Code des letzten Zustands, $\mathcal{R} = (\mathcal{Z})_1$ codiert die Registerinhalte des letzten Zustands, und $(\mathcal{R})_0 = ((\mathcal{Z})_1)_0 = \left(\left((y)_{\lg y - 1}\right)_1\right)_0$ ist der Inhalt des Ausgaberegisters am Ende. Also genügt es,

$$\mathcal{U}(y) = \left(\left((y)_{\lg y - 1}\right)_1\right)_0$$

zu wählen. ■

Korollar 2.28 *Die rekursiven Funktionen sind genau die GOTO-berechenbaren (totalen) Funktionen.*

Darüberhinaus zeigt die Kleenesche Normalform, dass man jede rekursive Funktion durch Komposition, primitive Rekursion und eine einzige Anwendung der μ -Rekursion aus den Grundfunktionen erhalten kann.

Übungsaufgabe 2.7 *Zeigen Sie mit Hilfe des Kleene-Prädikats direkt (also ohne Aufgabe 2.5 oder 2.6 zu benutzen): Jede GOTO-berechenbare partielle Funktion ist WHILE-berechenbar.*

2.6 Rekursiv aufzählbare Mengen

Definition 2.29 Eine Menge $A \subseteq \mathbb{N}^k$ heißt rekursiv aufzählbar, falls A Projektion einer rekursiven Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$ ist: $(x_1, \dots, x_k) \in A$ genau dann, wenn es ein y gibt, so dass $(x_1, \dots, x_k, y) \in \bar{A}$.

Bemerkung 2.30 Alle rekursiven Mengen sind rekursiv aufzählbar. Die rekursiv aufzählbaren Mengen sind abgeschlossen unter Projektion.

Proposition 2.31 Die folgenden Bedingungen an eine Menge $A \subseteq \mathbb{N}$ sind äquivalent.

1. A ist rekursiv aufzählbar.
2. $A = \emptyset$ oder A ist das Bild einer rekursiven Funktion.

Bemerkung 2.32 Eine Menge ist genau dann rekursiv, wenn sie selbst und ihr Komplement rekursiv aufzählbar sind.

Satz 2.33 Es gibt eine universelle rekursiv aufzählbare Menge $U \subseteq \mathbb{N}^2$: U selbst ist rekursiv aufzählbar, und für jede rekursiv aufzählbare Menge A gibt es ein $c \in \mathbb{N}$, so dass $A = \{x \in \mathbb{N} \mid (c, x) \in U\}$.

Beweis Die Menge

$$U = \{(c, x) \in \mathbb{N}^2 \mid \exists y(\mathcal{T}^1(c, x, y))\}$$

tut es. Weil \mathcal{T}^1 (primitiv) rekursiv ist, ist U rekursiv aufzählbar. Um die Universalität zu zeigen, betrachten wir eine beliebige rekursiv aufzählbare Menge $A = \{x \in \mathbb{N} \mid \exists z((x, z) \in \bar{A})\}$, wobei \bar{A} rekursiv ist. Sei \mathcal{P} der Code eines GOTO-Programms, das $\mu z(\chi_{\bar{A}}(x, z) > 0)$ berechnet. Das Programm hält genau dann an, wenn $x \in A$ ist. Daher ist $A = \{x \in \mathbb{N} \mid (\mathcal{P}, x) \in U\}$. ■

Korollar 2.34 U ist rekursiv aufzählbar, aber nicht rekursiv.

Beweis Wenn U rekursiv wäre, dann wäre auch $A = \{x \in \mathbb{N} \mid (x, x) \notin U\}$ rekursiv und insbesondere rekursiv aufzählbar. Sei $c \in \mathbb{N}$ so, dass $A = \{x \in \mathbb{N} \mid (c, x) \in U\}$. Es folgt $(c, c) \notin U \iff c \in A \iff (c, c) \in U$, also ein Widerspruch. ■

Übungsaufgabe 2.8 Man könnte eine Menge $A \subseteq \mathbb{N}^k$ primitiv rekursiv aufzählbar nennen, falls A Projektion einer primitiv rekursiven Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$ ist: $(x_1, \dots, x_k) \in A$ genau dann, wenn es ein y gibt, so dass $(x_1, \dots, x_k, y) \in \bar{A}$. Warum liest man davon nichts in Rekursionstheoriebüchern?

Kapitel 3

Logik

3.1 Strings und Sprachen

Definition 3.1 Sei A eine beliebige Menge. Die Elemente von $A^* = \bigcup_{k \in \mathbb{N}} A^k$ heißen Strings, Zeichenketten oder Wörter über dem Alphabet A .

Eine Sprache über dem Alphabet A ist eine beliebige Menge $L \subseteq A^*$ von Strings über A .

Wenn wir eine Aufzählung $A = \{a_0, a_1, a_2, \dots\}$ eines (höchstens abzählbaren) Alphabets A gegeben haben, können wir jeden String $a_{n_1} a_{n_2} \dots a_{n_k}$ durch die natürliche Zahl $\langle n_1, n_2, \dots, n_k \rangle \in \mathbb{N}$ repräsentieren und daher eine Sprache über A als Teilmenge von \mathbb{N} auffassen.

Wir nennen eine Sprache primitiv rekursiv, rekursiv oder rekursiv aufzählbar, wenn die so erhaltene Teilmenge von \mathbb{N} die jeweilige Eigenschaft hat. Diese Definition hängt implizit auch von der Aufzählung des Alphabets ab!

Übungsaufgabe 3.1 Ob eine Sprache über einem endlichen Alphabet primitiv rekursiv, rekursiv oder rekursiv aufzählbar ist, hängt nicht von der Wahl des Alphabets ab. Man finde eine Sprache über einem abzählbaren Alphabet, die unter einer Abzählung des Alphabets primitiv rekursiv ist, aber unter einer anderen nicht einmal rekursiv aufzählbar.

Sprachen kann man rekursiv definieren, so wie wir die Menge der primitiv rekursiven Funktionen und die Menge der rekursiven Funktionen rekursiv definiert haben. Als Beispiel betrachten wir die Sprache der Aussagenlogik mit Infix-Notation, definiert über dem Alphabet $A = \{\neg, \wedge, \vee, (,), \overset{0}{\mathbf{p}}, \overset{1}{\mathbf{p}}, \overset{2}{\mathbf{p}}, \overset{3}{\mathbf{p}}, \overset{4}{\mathbf{p}}, \dots\}$.

Definition 3.2 Die Sprache der Aussagenlogik in Infix-Notation ist die kleinste Menge $\mathcal{L}_0^{\text{inf}} \subseteq A^*$ über dem Alphabet $A = \{\neg, \wedge, \vee, (,), \overset{0}{\mathbf{p}}, \overset{1}{\mathbf{p}}, \overset{2}{\mathbf{p}}, \overset{3}{\mathbf{p}}, \overset{4}{\mathbf{p}}, \dots\}$, welche die folgenden Bedingungen erfüllt:

- $\overset{0}{\mathbf{p}} \in \mathcal{L}_0^{\text{inf}}, \overset{1}{\mathbf{p}} \in \mathcal{L}_0^{\text{inf}}, \overset{2}{\mathbf{p}} \in \mathcal{L}_0^{\text{inf}}, \overset{3}{\mathbf{p}} \in \mathcal{L}_0^{\text{inf}}$ usw.
- Wenn $\varphi \in \mathcal{L}_0^{\text{inf}}$ gilt, dann gilt auch $(\neg\varphi) \in \mathcal{L}_0^{\text{inf}}$.
- Wenn $\varphi_1, \varphi_2 \in \mathcal{L}_0^{\text{inf}}$ gilt, dann gilt auch $(\varphi_1 \wedge \varphi_2) \in \mathcal{L}_0^{\text{inf}}$.

- Wenn $\varphi_1, \varphi_2 \in \mathcal{L}_0^{\text{inf}}$ gilt, dann gilt auch $(\varphi_1 \mathbf{v} \varphi_2) \in \mathcal{L}_0^{\text{inf}}$.

Übungsaufgabe 3.2 Zeigen Sie, dass die Zeichenkette $(\neg((\mathbf{p} \mathbf{v} (\neg(\mathbf{p} \wedge \mathbf{p}))) \wedge \mathbf{p}))$ in der Sprache der Aussagenlogik mit Infix-Notation liegt. Zeigen Sie, dass die folgenden Zeichenketten nicht in dieser Sprache liegen:

- $\mathbf{)}\mathbf{p}(\mathbf{)}$
- $\neg((\mathbf{p} \mathbf{v} (\neg(\mathbf{p} \wedge \mathbf{p}))) \wedge \mathbf{p})$
- $(\mathbf{p} \wedge \mathbf{p} \wedge \mathbf{p})$
- $(\mathbf{p} \mathbf{v} ((\mathbf{p} \mathbf{v} \mathbf{p}) \mathbf{v} (\mathbf{p} \mathbf{v} \mathbf{p}))) \mathbf{v} \mathbf{p}$

Übungsaufgabe 3.3 Mittels der Aufzählung von A und der Bijektion $\langle \cdot \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$ können wir die Sprache $\mathcal{L}_0^{\text{inf}}$ der Aussagenlogik mit Infix-Notation als eine Teilmenge $L_0^{\text{inf}} \subset \mathbb{N}$ auffassen. Zeigen Sie, dass diese Teilmenge rekursiv ist.

Definition 3.3 Die Sprache der Aussagenlogik in polnischer Notation ist die kleinste Menge $\mathcal{L}_0^{\text{pol}} \subseteq A^*$ über dem Alphabet $A = \{\neg, \wedge, \mathbf{v}, (,), \mathbf{p}, \mathbf{p}, \mathbf{p}, \mathbf{p}, \mathbf{p}, \dots\}$, welche die folgenden Bedingungen erfüllt:

- $\mathbf{p} \in \mathcal{L}_0^{\text{pol}}, \mathbf{p} \in \mathcal{L}_0^{\text{pol}}, \mathbf{p} \in \mathcal{L}_0^{\text{pol}}, \mathbf{p} \in \mathcal{L}_0^{\text{pol}}$ usw.
- Wenn $\varphi \in \mathcal{L}_0^{\text{pol}}$ gilt, dann gilt auch $\neg\varphi \in \mathcal{L}_0^{\text{pol}}$.
- Wenn $\varphi_1, \varphi_2 \in \mathcal{L}_0^{\text{pol}}$ gilt, dann gilt auch $\wedge\varphi_1\varphi_2 \in \mathcal{L}_0^{\text{pol}}$.
- Wenn $\varphi_1, \varphi_2 \in \mathcal{L}_0^{\text{pol}}$ gilt, dann gilt auch $\mathbf{v}\varphi_1\varphi_2 \in \mathcal{L}_0^{\text{pol}}$.

Die Elemente von $\mathcal{L}_0^{\text{pol}}$ heißen aussagenlogische Formeln.

Übungsaufgabe 3.4 Natürlich ist die Sprache der Aussagenlogik in polnischer Notation, aufgefasst als $L_0^{\text{pol}} \subset \mathbb{N}$, ebenfalls rekursiv. Geben Sie eine rekursive Abbildung $f: \mathbb{N} \rightarrow \mathbb{N}$ an, deren Einschränkung eine Bijektion $f: L_0^{\text{pol}} \rightarrow L_0^{\text{inf}}$ ergibt.

Definition 3.4 Eine Belegung der Prädikate $\mathbf{p}, \mathbf{p}, \mathbf{p}, \dots$ ist eine Funktion $\beta: \{\mathbf{p}, \mathbf{p}, \mathbf{p}, \dots\} \rightarrow \{0, 1\}$. Die Fortsetzung einer Belegung der Prädikate auf $\mathcal{L}_0^{\text{pol}}$ ist die wie folgt rekursiv definierte Funktion $\bar{\beta}: \mathcal{L}_0^{\text{pol}} \rightarrow \{0, 1\}$:

- $\bar{\beta}(\mathbf{p}) = \beta(\mathbf{p}), \bar{\beta}(\mathbf{p}) = \beta(\mathbf{p}), \bar{\beta}(\mathbf{p}) = \beta(\mathbf{p}),$ usw.
- $\bar{\beta}(\neg\varphi) = 1 - \bar{\beta}(\varphi).$
- $\bar{\beta}(\wedge\varphi_1\varphi_2) = \bar{\beta}(\varphi_1) \cdot \bar{\beta}(\varphi_2).$
- $\bar{\beta}(\mathbf{v}\varphi_1\varphi_2) = 1 - (1 - \bar{\beta}(\varphi_1)) \cdot (1 - \bar{\beta}(\varphi_2)).$

Eine Tautologie ist eine aussagenlogische Formel $\varphi \in \mathcal{L}_0^{\text{pol}}$, so dass für jede Belegung $\beta: \{\mathbf{p}, \mathbf{p}, \mathbf{p}, \dots\} \rightarrow \{0, 1\}$ gilt: $\bar{\beta}(\varphi) = 1$.

Übungsaufgabe 3.5 Die Menge aller Tautologien ist rekursiv.

3.2 Signaturen, Strukturen und Terme

Definition 3.5 Eine Struktur M besteht aus einer Menge \underline{M} (der Grundmenge von M) zusammen mit ausgezeichneten, benannten Operationen und Relationen. Die Operationen heißen auch Funktionen. Sie sind Funktionen der Form $\underline{M}^k \rightarrow \underline{M}$ für $k \in \mathbb{N}$. Die Relationen sind Teilmengen von \underline{M}^k für $k \in \mathbb{N}$.

Hier einige Beispiele:

- Die Struktur $(\mathbb{N}, 0, 1, +, \times, <)$ mit der Grundmenge \mathbb{N} , den nullstelligen Operationen $0 \in \mathbb{N}$ und $1 \in \mathbb{N}$, den zweistelligen Operationen $+: \mathbb{N}^2 \rightarrow \mathbb{N}$ und $\times: \mathbb{N}^2 \rightarrow \mathbb{N}$ und der zweistelligen Relation $< \subseteq \mathbb{N}^2$.
- Die Struktur $(\mathbb{N}, 0, S)$ mit der Grundmenge \mathbb{N} , dem nullstelligen Operator 0 und dem einstelligen Operator $S: \mathbb{N} \rightarrow \mathbb{N}$, $x \mapsto x + 1$.
- Die Struktur $(\mathbb{Z}, 0, 1, -, +, \times)$ mit der Grundmenge \mathbb{Z} , den nullstelligen Operationen $0 \in \mathbb{N}$ und $1 \in \mathbb{N}$, der einstelligen Operation $-: \mathbb{N} \rightarrow \mathbb{N}$ (also $x \mapsto -x$), sowie den zweistelligen Operationen $+: \mathbb{N}^2 \rightarrow \mathbb{N}$ und $\times: \mathbb{N}^2 \rightarrow \mathbb{N}$.
- Die Struktur $(\mathbb{Q}, 0, 1, -, +, \times)$ mit der Grundmenge \mathbb{Q} , den nullstelligen Operationen $0 \in \mathbb{N}$ und $1 \in \mathbb{N}$, der einstelligen Operation $-: \mathbb{N} \rightarrow \mathbb{N}$ (also $x \mapsto -x$), sowie den zweistelligen Operationen $+: \mathbb{N}^2 \rightarrow \mathbb{N}$ und $\times: \mathbb{N}^2 \rightarrow \mathbb{N}$.
- ...

Angaben wie $(\mathbb{N}, 0, 1, +, \times, <)$ sind allgemein üblich, aber etwas schlampig. Dabei haben die Symbole wie 0 , $+$ und $<$ zwei Funktionen. Einerseits geben sie die Namen der Operationen und Relationen an, also die für sie benutzten Symbole. Andererseits stehen sie aber auch für die jeweiligen Operationen und Relationen selbst.

Definition 3.6 Die Signatur einer Struktur besteht aus den Namen der ausgezeichneten Operationen und Relationen, zusammen mit der Information, ob es sich jeweils um eine Operation oder eine Relation handelt, und wieviele Stellen sie hat. Die Elemente der Signatur heißen Operationssymbole und Relationssymbole.

Eine Struktur, die man durch Weglassen einiger Operations- oder Relationssymbole erhält, heißt ein Redukt der ursprünglichen Struktur.

Eine Struktur der Signatur σ bezeichnen wir auch als σ -Struktur.

Formal können wir eine Signatur σ beispielsweise auffassen als ein Tripel $\sigma = (\sigma^{\text{Op}}, \sigma^{\text{Rel}}, \text{ar})$, wobei σ^{Op} und σ^{Rel} beliebige disjunkte Mengen sind (die Menge der Operationssymbole und die Menge der Relationssymbole) und $\text{ar}: \sigma^{\text{Op}} \cup \sigma^{\text{Rel}} \rightarrow \mathbb{N}$ die Stelligkeiten der Symbole angibt. Manchmal sind wir präzise und unterscheiden zwischen dem (k -stelligen) Operationssymbol f und der zugehörigen Operation $f^M: \underline{M}^k \rightarrow \underline{M}$ und zwischen dem (k -stelligen) Relationssymbol R und der zugehörigen Relation $R^M \subseteq \underline{M}^k$. Das ist vor allem dann sinnvoll, wenn wir zwei verschiedene Strukturen betrachten, deren Signaturen sich überschneiden.

Definition 3.7 Ein Isomorphismus $h: M \cong N$ zwischen zwei Strukturen M und N derselben Signatur ist eine Bijektion $h: \underline{M} \rightarrow \underline{N}$ zwischen den Grundmengen, die mit den Operationen und Relationen vertauscht. Genauer:

- Für jedes k -stellige Operationssymbol g und jedes k -Tupel $(x_1, \dots, x_k) \in \underline{M}^k$ gilt $h(g(x_1, \dots, x_k)) = g(h(x_1), \dots, h(x_k))$.
- Für jedes k -stellige Relationssymbol R und jedes k -Tupel $(x_1, \dots, x_k) \in \underline{M}^k$ gilt $(x_1, \dots, x_k) \in R^M \iff (h(x_1), \dots, h(x_k)) \in R^N$.

Zwei Strukturen heißen isomorph, falls sie dieselbe Signatur haben und es einen Isomorphismus zwischen ihnen gibt.

Übungsaufgabe 3.6 Bestimmen Sie die vier Redukte von $(\mathbb{N}, 0, 1, +, \times, <)$, die isomorph zu Redukten von $(\mathbb{Q}, 0, 1, -, +, \times)$ sind.

Definition 3.8 Eine Substruktur einer Struktur M ist eine Struktur N mit derselben Signatur, so dass $\underline{N} \subseteq \underline{M}$ gilt und die Operationen und Relationen von N sich durch Einschränkung der Operationen und Relationen von M ergeben.

Übungsaufgabe 3.7 Welche Bedingung muss eine Teilmenge $\underline{N} \subseteq \underline{M}$ erfüllen, damit sie Grundmenge einer Substruktur N von M sein kann? Wieviele solche Substrukturen mit der Grundmenge \underline{N} gibt es?

Übungsaufgabe 3.8 Eine Struktur ohne Relationen heißt funktionale Struktur oder universelle Algebra. Gegeben zwei funktionale Strukturen M und N derselben Signatur heißt eine Abbildung $h: \underline{M} \rightarrow \underline{N}$ ein Homomorphismus von M nach N , falls die folgende Bedingung erfüllt ist:

- Für jedes k -stellige Operationssymbol g und jedes k -Tupel $(x_1, \dots, x_k) \in \underline{M}^k$ gilt $h(g(x_1, \dots, x_k)) = g(h(x_1), \dots, h(x_k))$.

Zeigen Sie: Die Verknüpfung von zwei Homomorphismen ist wieder ein Homomorphismus. Eine funktionale Struktur N ist genau dann eine Substruktur einer Struktur M derselben Signatur, wenn $\underline{N} \subseteq \underline{M}$ gilt und die Inklusionsabbildung ein Homomorphismus ist.

Wir fixieren eine abzählbare (und abgezählte) Menge $\mathbb{X} = \{\overset{0}{\mathbf{X}}, \overset{1}{\mathbf{X}}, \overset{2}{\mathbf{X}}, \dots\}$ von Symbolen, die wir *Variable* nennen.

Definition 3.9 Die Menge der σ -Terme ist die kleinste Sprache über dem Alphabet $\sigma^{\text{Op}} \cup \sigma^{\text{Rel}} \cup \mathbb{X}$, welche die folgenden Bedingungen erfüllt.

- Jede Variable ist ein σ -Term.
- Wenn $f \in \sigma^{\text{Op}}$ und $k = \text{ar}(f)$ gilt, und t_1, \dots, t_k sind σ -Terme, dann ist auch $ft_1 \dots t_k$ ein σ -Term.

Übungsaufgabe 3.9 Bestimmen Sie jeweils alle Signaturen σ , so dass $\sigma^{\text{Rel}} = \emptyset$, $|\sigma^{\text{Op}}| = 2$ und t ein σ -Term ist. Manchmal gibt es keine Lösung oder mehrere Lösungen.

- $t = \overset{11}{\mathbf{fXXC}}$.
- $t = \overset{17}{\mathbf{fXgXC}}$.
- $t = \overset{3}{\mathbf{fXgXX}}$.

- $t = \text{-----}^{12}\text{+XX.}$
- $t = \text{+++++++}^{12}\text{XX.}$
- $t = \text{++nn}^2\text{x+nn+nnn+nxn.}^7$
- $t = \text{AABB}^{0000}\text{xxxx.}$

Lemma 3.10 (Eindeutige Lesbarkeit von Termen) Jeder σ -Term ist entweder eine Variable oder lässt sich in der Form $ft_1 \dots t_k$ schreiben, wobei $f \in \sigma^{\text{Op}}$ ein k -stelliges Operationssymbol ist und t_1, \dots, t_k σ -Terme sind. Im zweiten Fall sind k , f und t_1, \dots, t_k eindeutig bestimmt.

Definition 3.11 Eine Belegung der Variablen in einer Struktur M ist eine Abbildung $\beta: \mathbb{X} \rightarrow \underline{M}$. Wenn σ die Signatur von M ist, setzen wir eine Belegung β in M wie folgt rekursiv zu einer Abbildung $\bar{\beta}: \{t \mid t \text{ ist } \sigma\text{-Term}\} \rightarrow \underline{M}$ fort:

- $\bar{\beta}(\overset{0}{x}) = \beta(\overset{0}{x}), \bar{\beta}(\overset{1}{x}) = \beta(\overset{1}{x}), \bar{\beta}(\overset{2}{x}) = \beta(\overset{2}{x}),$ usw.
- $\bar{\beta}(ft_1 \dots t_k) = f^M(\bar{\beta}(t_1), \dots, \bar{\beta}(t_k)).$

Übungsaufgabe 3.10 Nennen wir zwei σ -Terme t und t' äquivalent, falls für jede Struktur M der Signatur σ und jede Belegung β in M gilt: $\bar{\beta}(t) = \bar{\beta}(t')$. Zeigen Sie, dass zwei σ -Terme genau dann gleich sind, wenn sie identisch sind. (Hinweis: Es gibt sogar für jede Signatur σ eine einzige σ -Struktur M , so dass für alle Belegungen β in M und alle σ -Terme t, t' gilt: $\bar{\beta}(t) = \bar{\beta}(t') \iff t = t'$. Diese Struktur heißt Termalgebra, weil sie aus Termen besteht.)

3.3 Syntax der Prädikatenlogik 1. Stufe

Wir machen die stillschweigende Annahme, dass die speziellen Symbole $=, \neg, \wedge$, die wir im Folgenden benutzen werden, nie in einer Signatur auftreten. (Sollte das doch einmal geschehen, würden wir zwischen dem Symbol als Teil der Signatur und dem Symbol der Prädikatenlogik unterscheiden wie bei disjunkten Vereinigungen.)

Definition 3.12 Die Menge der σ -Formeln (der Prädikatenlogik 1. Stufe) ist die kleinste Sprache über dem Alphabet $\sigma^{\text{Op}} \cup \sigma^{\text{Rel}} \cup \{=, \exists, \neg, \wedge\} \cup \mathbb{X}$, welche die folgenden Bedingungen erfüllt.

- Wenn t_1 und t_2 σ -Terme sind, dann ist $=t_1t_2$ eine σ -Formel.
- Wenn $R \in \sigma^{\text{Rel}}$ und $k = \text{ar}(R)$, und t_1, \dots, t_k sind σ -Terme, dann ist $Rt_1 \dots t_k$ eine σ -Formel.
- Wenn φ eine σ -Formel ist, dann ist auch $\neg\varphi$ eine σ -Formel.
- Wenn φ und ψ σ -Formeln sind, dann ist auch $\wedge\varphi\psi$ eine σ -Formel.
- Wenn φ eine σ -Formel ist und $x \in \mathbb{X}$ eine Variable, dann ist auch $\exists x\varphi$ eine σ -Formel.

Übungsaufgabe 3.11 Bestimmen Sie jeweils alle Signaturen σ , so dass $|\sigma^{\text{Op}} \cup \sigma^{\text{Rel}}| = 3$ und φ eine σ -Formel ist.

- $\varphi = \neg \exists x \neg \exists x = x f + x x$.
- $\varphi = \neg \exists x \neg \wedge = x x = x + x \theta$.
- $\varphi = \neg \wedge = x x \neg = x x$.

Lemma 3.13 (Eindeutige Lesbarkeit von Formeln) Jede σ -Formel lässt sich in genau einer der folgenden Weisen zerlegen:

- Als $=t_1 t_2$, wobei t_1, t_2 σ -Terme sind.
- Als $R t_1 \dots t_k$, wobei $R \in \sigma^{\text{Rel}}$ und $k = \text{ar}(R)$, und t_1, \dots, t_k sind σ -Terme.
- Als $\neg \varphi$ für eine σ -Formel φ .
- Als $\wedge \varphi \psi$ für σ -Formeln φ und ψ .
- Als $\exists x \varphi$ für eine σ -Formel φ und eine Variable x .

Die Zerlegung ist außerdem eindeutig, d.h. t_1, t_2 , bzw. R und t_1, \dots, t_k , bzw. φ bzw. φ, ψ bzw. x und φ sind eindeutig bestimmt.

Definition 3.14 Die Menge der Variablen, die in einer σ -Formel frei vorkommen, ist wie folgt rekursiv definiert.

- $\text{Frei}(=t_1 t_2)$ ist die Menge aller Variablen, die in t_1 oder t_2 vorkommen.
- $\text{Frei}(R t_1 \dots t_k)$ ist die Menge aller Variablen, die in t_1, t_2, \dots oder t_k vorkommen.
- $\text{Frei}(\neg \varphi) = \text{Frei}(\varphi)$.
- $\text{Frei}(\wedge \varphi \psi) = \text{Frei}(\varphi) \cup \text{Frei}(\psi)$.
- $\text{Frei}(\exists x \varphi) = \text{Frei}(\varphi) \setminus \{x\}$.

Übungsaufgabe 3.12 Die Signatur σ_{AG} der abelschen Gruppen besteht aus dem 0-stelligen Operationssymbol θ , dem 2-stelligen Operationssymbol $+$ und dem 1-stelligen Operationssymbol $-$. Welche der folgenden Zeichenketten sind σ_{AG} -Formeln, welche sind σ_{AG} -Terme?

- $=x x$
- \bar{x}
- $++++-1-1-1-1-1$
- $\wedge \exists x \neg = x x = + + + + - 1 - 1 - 1 - 1 - 1 - + + + + 1 1 1 1 1$
- $\wedge = \theta 1 = + + 1 1 1 + \theta \theta$

Übungsaufgabe 3.13 Welche der folgenden Zeichenketten sind σ_{AG} -Formeln? Geben Sie für die Formeln auch an, welche Variablen frei in ihnen vorkommen. (Auch in einer Zeichenkette, die kein Term und keine Formel ist – z.B. schon aus dem einfachen Grund, dass sie Symbole enthält, die nicht im Alphabet über σ_{AG} vorkommen – können wir manchmal durchaus Sinn erkennen. Lassen Sie sich dadurch nicht verwirren und benutzen Sie stur die formale Definition!)

- $\forall q = +q - q \theta$

- $\exists x \neg = x + x 0$
- $\forall x \exists x (x + x = 0)$
- $\neg \exists x \exists x \neg = + x x + x x$
- $\neg \wedge = x 0 \wedge \neg = x 0$

Übungsaufgabe 3.14 Geben Sie eine Signatur σ , eine σ -Formel φ und einen σ -Term t an, so dass das erste Zeichen von φ gleich dem ersten Zeichen von t ist. (Oder beweisen Sie, dass es nicht geht.)

3.4 Semantik der Prädikatenlogik

Unter einer Belegung der Variablen in einer Struktur entspricht jedem Term ein Element der Struktur. Diesen Zusammenhang werden wir jetzt auf Formeln erweitern: Unter einer Belegung der Variablen in einer Struktur entspricht jeder Formel ein Wahrheitswert, d.h. wahr (1) oder falsch (0). In gewissem Sinne können wir „Wahrheit“ also formal definieren:

Definition 3.15 (Tarskis Definition der Wahrheit) Wenn σ die Signatur von M ist, definieren wir für jede Belegung β in M wie folgt rekursiv eine Abbildung $\hat{\beta}: \{\varphi \mid \varphi \text{ ist } \sigma\text{-Formel}\} \rightarrow \{0, 1\}$:

- $\hat{\beta}(=t_1 t_2) = \begin{cases} 1 & \text{falls } \bar{\beta}(t_1) = \bar{\beta}(t_2) \\ 0 & \text{sonst.} \end{cases}$
- $\hat{\beta}(R t_1 \dots t_k) = \begin{cases} 1 & \text{falls } (\bar{\beta}(t_1), \dots, \bar{\beta}(t_k)) \in R^M \\ 0 & \text{sonst.} \end{cases}$
- $\hat{\beta}(\neg \varphi) = 1 - \hat{\beta}(\varphi)$.
- $\hat{\beta}(\wedge \varphi \psi) = \hat{\beta}(\varphi) \cdot \hat{\beta}(\psi)$.
- $\hat{\beta}(\exists x \varphi) = \begin{cases} 1 & \text{falls es ein Element } e \in \underline{M} \text{ gibt, so dass } \widehat{\beta[\frac{e}{x}]}(\varphi) = 1, \text{ wo-} \\ 0 & \text{sonst} \end{cases}$
bei $\beta[\frac{e}{x}]$ sich von β nur durch $\beta(x) = e$ unterscheidet.

Dass $\hat{\beta}$ wohldefiniert ist, folgt wie üblich aus der eindeutigen Lesbarkeit.

Lemma 3.16 Sei σ eine Signatur, φ eine σ -Formel und M eine Struktur der Signatur σ . Seien $\beta_1, \beta_2: \mathbb{X} \rightarrow \underline{M}$ zwei Belegungen der Variablen in M . Falls β_1 und β_2 auf $\text{Frei}(\varphi)$ übereinstimmen (d.h. falls $\beta_1(x) = \beta_2(x)$ für alle $x \in \text{Frei}(\varphi)$), dann ist $\beta_1(\varphi) = \beta_2(\varphi)$.

Beweis Durch Induktion über den Aufbau von φ . Wirklich interessant ist eigentlich nur der Fall, in dem ein Quantor eingeführt wird. ■

Definition 3.17 (Modellbeziehung) Sei σ eine Signatur. Ein σ -Satz ist eine σ -Formel ohne freie Variable. Sei φ ein σ -Satz und M eine σ -Struktur. Falls für eine Belegung $\beta: \mathbb{X} \rightarrow \underline{M}$ der Variablen in M (äquivalent: für alle Belegungen) $\hat{\beta}(\varphi) = 1$ gilt, dann schreiben wir $M \models \varphi$ und sagen, dass φ in M gilt, oder dass M ein Modell von φ ist. Andernfalls schreiben wir $M \not\models \varphi$.

Übungsaufgabe 3.15

Geben Sie für jeden der folgenden Sätze an, ob er in $(\mathbb{N}, 0, 1, +, <)$ gilt oder nicht. Ebenso für $(\mathbb{Z}, 0, 1, +, <)$, $(\mathbb{Q}, 0, 1, +, <)$ und $(\mathbb{R}, 0, 1, +, <)$.

- $\exists x < x \ 0$
- $\exists x \exists x \neg = + x x + x x$
- $\exists x \exists x \wedge < x x \neg \exists x \wedge < x x < x x$

Übungsaufgabe 3.16 Zeigen Sie: Für jeden σ -Satz φ und jede σ -Struktur M mit $\underline{M} \neq \emptyset$ gilt entweder $M \models \varphi$ oder $M \models \neg\varphi$. (Hinweis: Was ist die einzige Methode, die Sie kennen, um derartige Behauptungen zu beweisen? Warum funktioniert sie hier nicht direkt? Können Sie mit derselben Methode eine andere Behauptung beweisen, aus der die gefragte folgt?)

Bei der Definition der σ -Formeln mussten wir wie zuvor schon bei der Definition der Aussagenlogik einige willkürliche Entscheidungen treffen. In diesem Fall haben wir uns beispielsweise das Symbol \forall gespart, weil wir Formeln im allgemeineren Sinne, in denen auch dieses Symbol erlaubt ist, als „Abkürzungen“ für σ -Formeln im von uns definierten präzisen Sinn auffassen können.

Übungsaufgabe 3.17 Definieren Sie verallgemeinerte σ -Formeln über dem größeren Alphabet $\sigma^{\text{Op}} \cup \sigma^{\text{Rel}} \cup \mathbb{X} \cup \{=, \exists, \forall, \neg, \wedge, \vee\}$ in sinnvoller Weise. Insbesondere sollten die σ -Formeln genau diejenigen verallgemeinerten σ -Formeln sein, in welchen die zusätzlichen Symbole \forall und \vee nicht vorkommen, und beispielsweise sollte $\forall x \forall x \vee = x x \neg = x x$ für jede Signatur σ eine verallgemeinerte σ -Formel sein.

Setzen Sie außerdem auch die Definitionen von freien Variablen, von $\hat{\beta}$ für Belegungen β sowie der Modellbeziehung auf verallgemeinerte σ -Formeln bzw. auf verallgemeinerte σ -Sätze (verallgemeinerte σ -Formeln ohne freie Variable) fort.

Übungsaufgabe 3.18 Definieren Sie für jede Signatur σ eine Abbildung

$$I: \{\text{verallgemeinerte } \sigma\text{-Formeln}\} \rightarrow \{\sigma\text{-Formeln}\},$$

so dass für alle verallgemeinerten σ -Formeln φ gilt: $\text{Frei}(I(\varphi)) = \text{Frei}(\varphi)$, sowie $M \models I(\varphi) \iff M \models \varphi$ für alle σ -Strukturen M .

3.5 Beweisbarkeit

Bevor wir uns mit der Beweisbarkeit von Formeln beschäftigen, müssen wir uns noch ein kleines Problem anschauen, das wir bisher unter den Teppich gekehrt haben:

Bemerkung 3.18 Wenn σ eine Signatur ohne nullstellige Operationssymbole (d.h. Konstanten) ist, dann gibt es σ -Strukturen M mit einer Grundmenge $\underline{M} = \emptyset$. Für solche Strukturen gilt $M \models \varphi$ für alle σ -Sätze φ .

Daher werden wir ab jetzt nur Strukturen mit nichtleerer Grundmenge betrachten.

Definition 3.19 Eine σ -Formel φ heißt erfüllbar, falls es eine nichtleere σ -Struktur M und eine Belegung β der Variablen in M gibt, so dass $\hat{\beta}(\varphi) = 1$ ist. Sie heißt allgemeingültig, falls für jede nichtleere σ -Struktur M und jede Belegung β in M gilt: $\hat{\beta}(\varphi) = 1$.

Übungsaufgabe 3.19 Gegeben eine endliche Signatur σ . Für jedes $k \in \mathbb{N}$ sei $m_k \in \mathbb{N}$ die Anzahl der k -stelligen Operationssymbole von σ und $n_k \in \mathbb{N}$ die Anzahl der k -stelligen Relationssymbole. Wieviele σ -Strukturen M mit $\underline{M} = \emptyset$ gibt es?

Definition 3.20 Die σ -Formel, die man aus einer σ -Formel φ erhält, indem man jedes freie Vorkommen einer Variable $x \in \mathbb{X}$ durch denselben σ -Term t ersetzt, bezeichnen wir mit $\varphi[\frac{t}{x}]$. Formal ist $\varphi[\frac{t}{x}]$ wie folgt rekursiv definiert:

- $(=t_1 t_2)[\frac{t}{x}] = =(t_1[\frac{t}{x}]) (t_2[\frac{t}{x}])$.
- $(Rt_1 \dots t_k)[\frac{t}{x}] = R(t_1[\frac{t}{x}]) \dots (t_k[\frac{t}{x}])$.
- $(\neg\varphi[\frac{t}{x}]) = \neg(\varphi[\frac{t}{x}])$.
- $(\wedge\varphi\psi)[\frac{t}{x}] = \wedge\varphi[\frac{t}{x}]\psi[\frac{t}{x}]$.
- $(\exists y\varphi[\frac{t}{x}]) = \exists y(\varphi[\frac{t}{x}])$ falls $y \neq x$.
- $(\exists x\varphi[\frac{t}{x}]) = \exists x\varphi$ falls $y = x$.

Lemma 3.21 Sei σ eine Signatur, M eine σ -Struktur, $\beta: \mathbb{X} \rightarrow \underline{M}$ eine Belegung in M , $x \in V$ eine Variable, t ein σ -Term und φ eine σ -Formel. Dann ist $\hat{\beta}(\varphi[\frac{t}{x}]) = (\hat{\beta}[\frac{\beta(t)}{x}])(\varphi)$.

Beweis Durch Induktion über den Aufbau von φ . ■

Definition 3.22 Die beweisbaren σ -Formeln sind die kleinste Menge Φ von σ -Formeln, so dass gilt:

- Tautologien: Wenn man in einer Tautologie der Aussagenlogik (siehe Definition 3.4) alle Prädikate durch σ -Formeln ersetzt, so dass dasselbe Prädikat durch dieselbe Formel ersetzt wird, dann ist die resultierende Formel in Φ . (Die Formeln der Prädikatenlogik, die man auf diese Weise erhält, heißen ebenfalls Tautologien.)
- Modus ponens: Wenn φ und $\neg\wedge\varphi\neg\psi$ in Φ sind, dann auch ψ .
- Gleichheitsaxiome: Die folgenden Formeln sind in Φ :

$$\begin{aligned}
 & - \overset{0\ 0}{=} \overset{0}{\wedge} \overset{0}{\wedge} \\
 & - \overset{0\ 1}{\neg\wedge} = \overset{0}{\wedge} \overset{1}{\neg} \overset{1\ 0}{\neg} = \overset{0}{\wedge} \overset{1}{\neg} \overset{1}{\neg} \\
 & - \overset{0\ 1}{\neg\wedge\wedge} = \overset{0}{\wedge} \overset{1}{\wedge} = \overset{1\ 2}{\wedge} \overset{0\ 2}{\neg} = \overset{0}{\wedge} \overset{2}{\neg} \\
 & - \neg\wedge^k = \overset{1\ k+1}{\wedge} \overset{2\ k+2}{\wedge} = \overset{2\ k+2}{\wedge} \overset{k\ 2k}{\wedge} \dots = \overset{1\ 2}{\wedge} \overset{k}{\wedge} \overset{k+1\ k+2}{\wedge} \dots \overset{2k}{\wedge} \text{ für jedes } k\text{-stellige} \\
 & \quad \text{Operationssymbol } f \in \sigma^{\text{Op}} \\
 & - \neg\wedge^k = \overset{1\ k+1}{\wedge} \overset{2\ k+2}{\wedge} = \overset{2\ k+2}{\wedge} \overset{k\ 2k}{\wedge} \overset{1\ 2}{\wedge} \overset{k}{\wedge} \overset{k+1\ k+2}{\wedge} \dots \overset{2k}{\wedge} \text{ für jedes } k\text{-stellige} \\
 & \quad \text{Relationssymbol } R \in \sigma^{\text{Rel}}.
 \end{aligned}$$

- Existenzaxiome: Für jede σ -Formel φ , jeden σ -Term t , dessen Variable nicht in φ vorkommen, und jede Variable $x \in \mathbb{X}$ ist auch $\neg\wedge(\varphi[\frac{t}{x}])\neg\exists x\varphi$ in Φ .

- Existenz Einführung: Wenn $\neg \wedge \varphi \neg \psi$ in Φ ist und die Variable x in ψ nicht frei vorkommt, dann ist auch $\neg \wedge \exists x \varphi \neg \psi$ in Φ .

Satz 3.23 (Korrektheitssatz) Jede beweisbare σ -Formel ist allgemeingültig.

Beweis Man überprüft, dass die Menge Φ der allgemeingültigen Formeln die in der Definition geforderten Eigenschaften hat. Sie enthält daher mindestens die beweisbaren Formeln. ■

Beweisbarkeit ist also eine (einseitige) Näherung für Allgemeingültigkeit. Ihr Nutzen ergibt sich aus der rekursiven Aufzählbarkeit der beweisbaren Formeln:

Satz 3.24 Sei σ eine endliche Signatur, und sei $\alpha: \mathbb{N} \rightarrow A = \sigma^{\text{Op}} \cup \sigma^{\text{Rel}} \cup \{=, \exists, \neg, \wedge\} \cup \mathbb{X}$ eine Aufzählung des Alphabets A . Sei $\ulcorner \varphi \urcorner = \langle a_1, a_2, \dots, a_k \rangle \in \mathbb{N}^k$ für jede Zeichenkette $\varphi = a_1 a_2 \dots a_k \in A^k$. Dann ist $\{\ulcorner \varphi \urcorner \mid \varphi \text{ beweisbare } \sigma\text{-Formel}\} \subseteq \mathbb{N}^k$ eine rekursiv aufzählbare Menge.

Bemerkung 3.25 Wenn φ und ψ beweisbar sind, dann auch $\wedge \varphi \psi$.

Beweis Weil φ und die Tautologie $\neg \wedge \varphi \neg \neg \wedge \psi \neg \wedge \varphi \psi$ beweisbar sind, ist mittels modus ponens auch $\neg \wedge \psi \neg \wedge \varphi \psi$. Weil ψ und $\neg \wedge \psi \neg \wedge \varphi \psi$ beweisbar sind, ist mittels modus ponens auch $\wedge \varphi \psi$ beweisbar. ■

Im nächsten Abschnitt werden wir sehen, dass Beweisbarkeit sogar äquivalent zu Allgemeingültigkeit ist. Dadurch erhalten wir das gar nicht offensichtliche Resultat, dass die Menge der allgemeingültigen Formeln rekursiv aufzählbar ist.

Übungsaufgabe 3.20 Sei σ eine beliebige Signatur. Wir betrachten die folgenden σ -Sätze:

- $\varphi = \neg \wedge \exists x = x x \neg \exists x = x x$.
- $\varphi = \neg \wedge \exists x = x x \neg \exists x = x x$.
- $\varphi = \neg \exists x \neg = x x$.

Zeigen Sie jeweils, dass für alle σ -Strukturen M gilt: $M \models \varphi$. Beweisen Sie für jeden der Sätze, dass er eine Tautologie ist bzw. dass er es nicht ist.

Übungsaufgabe 3.21 Sei σ eine beliebige Signatur. Zeigen Sie, dass beim Existenzaxiom die Einschränkung, dass die Variablen von t nicht in φ vorkommen, nötig ist, indem Sie für die σ -Formel $\varphi = \neg \exists x \neg = x x$ eine σ -Struktur angeben, in der die Formel $\neg \wedge \varphi \left[\frac{x}{0} \right] \neg \exists x \varphi = \neg \wedge \neg \exists x \neg = x x \neg \neg \exists x \neg \exists x \neg = x x$ nicht gilt.

Übungsaufgabe 3.22 Sei σ die Signatur mit $\sigma^{\text{Op}} = \emptyset$, $\sigma^{\text{Rel}} = \{\mathbf{I}\}$ und $\text{ar}(\mathbf{I}) = 2$. Wir betrachten die σ -Struktur M mit

$\underline{M} = \{1, 2, 3, 4, 6, \text{Karlsplatz}, \text{Stephansplatz}, \text{Schwedenplatz}, \text{Praterstern}, \text{Landstraße}, \text{Volkstheater}, \text{Schottenring}, \text{Westbahnhof}, \text{Längenfeldgasse}, \text{Spittelau}\},$

Die Vereinigung über eine solche maximale Kette ist widerspruchsfrei, lässt sich aber nicht widerspruchsfrei echt erweitern (ohne auch die Signatur zu erweitern). Daher muss die Vereinigung vollständig sein. ■

Selbst eine vollständige Theorie gibt uns noch nicht direkt ein Modell (außer im Sonderfall, dass das Modell endlich ist). Gegeben eine beliebige σ -Struktur M , können wir die Signatur σ zu σ' erweitern, so dass für jedes Element $m \in \underline{M}$ eine neue Konstante \mathbf{c}_m vorhanden ist. Die Struktur erweitern wir durch die offensichtliche Interpretation der neuen Konstanten zu einer σ' -Struktur M' . Dann ist T' , die σ' -Theorie von M' , eine vollständige Theorie, und man kann M' (und damit auch M) bis auf Isomorphie aus T' ablesen. (Das heißt nicht, dass es nicht neben M' noch weitere, nichtisomorphe, Modelle geben kann, bei denen dann einige Elemente nicht durch eine Konstante repräsentiert sind.) T' hat neben Vollständigkeit noch eine weitere wichtige Eigenschaft:

Definition 3.29 Eine σ -Theorie T heißt Henkintheorie, falls für jede σ -Formel φ mit genau einer freien Variablen x eine Konstante (d.h. nullstellige Operation) $c \in \sigma^{\text{Op}}$ existiert, so dass $\neg \lambda \exists x \varphi \neg \varphi[\frac{c}{x}] \in T$.

Proposition 3.30 Wenn T eine vollständige (σ -)Henkintheorie ist, kann man wie folgt ein Modell $M \models T$ definieren. Sei $C \subseteq \sigma^{\text{Op}}$ die Menge der Konstanten in σ . Sei \sim die Äquivalenzrelation auf C , die durch $c \sim d \iff =cd \in T$ gegeben ist. Als Grundmenge der Struktur M nehmen wir $\underline{M} = C / \sim$. Die Sätze der Form $Rc_1 \dots c_k \in T$ geben uns die Interpretationen R^M der Relationssymbole. Für ein k -stelliges Operationssymbol $f \in \sigma^{\text{Op}}$ ist $f^M(c_1 / \sim, \dots, c_k / \sim) = d / \sim$, wobei $d \in C$ so gewählt ist, dass $=dfc_1 \dots c_k \in T$.

Beweis Die Relation \sim ist eine Äquivalenzrelation, weil T vollständig ist (und wegen der Gleichheitsaxiome). Weil T eine Henkintheorie ist, gibt es immer mindestens eine Konstante d , so dass $=dfc_1 \dots c_k \in T$. (Sei d so, dass $\neg \lambda \exists x =xf c_1 \dots c_k \neg =dfc_1 \dots c_k \in T$. Weil $\exists x =xf c_1 \dots c_k$ allgemeingültig und T vollständig ist, ist auch $=dfc_1 \dots c_k \in T$.) Aus den Gleichheitsaxiomen folgt auch, dass alles wohldefiniert ist.

Man beweist jetzt einfach durch Induktion über den Formelaufbau: Für jede Formel φ und jede Interpretation der Variablen in M ist $\hat{\beta}(\varphi) = 1$ genau dann, wenn $\varphi[\frac{c_1}{x_1}][\frac{c_2}{x_2}] \dots [\frac{c_n}{x_n}] \in T$ ist, wobei x_1, \dots, x_n die freien Variablen von φ sind und die c_i so gewählt sind, dass $\beta(x_i) = c_i$ ist. Insbesondere gilt für Sätze φ die Äquivalenz $M \models \varphi \iff \hat{\beta}(\varphi) = 1 \iff \varphi \in T$. Daher ist $M \models T$, d.h. $M \models \varphi$ für alle $\varphi \in T$. ■

Jetzt müssen wir nur noch einen Weg finden, jede Theorie zu einer vollständigen Henkintheorie zu erweitern. Dabei müssen wir i.a. die Signatur durch zusätzliche Konstanten erweitern. Die beiden folgenden technischen Lemmas sind der schwerste Teil im Beweis des Vollständigkeitsatzes.

Lemma 3.31 Sei φ eine σ -Formel und $c \in \sigma^{\text{Op}}$ eine Konstante, die in φ nicht vorkommt. Wenn $\varphi[\frac{c}{x}]$ beweisbar ist, dann ist auch φ beweisbar.

Beweis Durch Induktion über die Beweisbarkeit von $\varphi[\frac{c}{x}]$. Beispielsweise muss man überprüfen, dass $=cc$ beweisbar ist, was aus dem entsprechenden Gleichheitsaxiom für den nullstelligen Operator \mathbf{c} folgt. Wir sparen uns alle weiteren Details. ■

Lemma 3.32 Sei φ eine σ -Formel mit einer freien Variable x und ψ ein σ -Satz. Falls $c \in \sigma^{\text{Op}}$ eine Konstante ist, die in φ und ψ nicht vorkommt und der Satz $\neg\Lambda\psi\neg\Lambda\exists x\varphi\neg\varphi[\frac{c}{x}]$ beweisbar ist, dann ist auch der Satz $\neg\psi$ beweisbar.

Beweis Zunächst betrachten wir die drei Formeln

$$\begin{aligned} &\neg\Lambda\psi\neg\Lambda\exists x\varphi\neg\varphi[\frac{c}{x}] \\ &\neg\Lambda\neg\Lambda\psi\neg\Lambda\exists x\varphi\neg\varphi[\frac{c}{x}]\neg\Lambda\psi\neg\exists x\varphi \\ &\neg\Lambda\psi\neg\exists x\varphi \end{aligned}$$

Die erste ist beweisbar nach Voraussetzung, und die zweite weil sie eine Tautologie ist. Die dritte folgt dann mit modus ponens. Eine weitere Formel leiten wir ganz analog ab.

$$\begin{aligned} &\neg\Lambda\psi\neg\Lambda\exists x\varphi\neg\varphi[\frac{c}{x}] \\ &\neg\Lambda\neg\Lambda\psi\neg\Lambda\exists x\varphi\neg\varphi[\frac{c}{x}]\neg\neg\Lambda\psi\varphi[\frac{c}{x}] \\ &\neg\Lambda\psi\varphi[\frac{c}{x}] \end{aligned}$$

Weil ψ ein Satz ist, kommt x nicht frei in ψ vor. Wir können daher die zuletzt abgeleitete Formel auch als $(\neg\Lambda\psi\varphi)[\frac{c}{x}]$ schreiben. Mit Lemma 3.31 folgt, dass auch $\neg\Lambda\psi\varphi$ beweisbar ist. Mit Hilfe einer Tautologie und modus ponens können wir folgern, dass auch $\neg\Lambda\varphi\neg\neg\psi$ beweisbar ist. Mittels Existenzführung folgt, dass $\neg\Lambda\exists x\varphi\neg\neg\psi$ beweisbar ist. Wiederum mittels einer Tautologie und modus ponens folgt die Beweisbarkeit von $\neg\Lambda\psi\exists x\varphi$.

Mit Hilfe von Bemerkung 3.25 können wir die zuerst bewiesene Formel $\neg\Lambda\psi\neg\exists x\varphi$ und die gerade eben bewiesene Formel $\neg\Lambda\psi\exists x\varphi$ zur ebenfalls beweisbaren Formel $\chi = \Lambda\neg\Lambda\psi\neg\exists x\varphi\neg\Lambda\psi\exists x\varphi$ kombinieren. Zusammen mit der Tautologie $\neg\Lambda\chi\neg\neg\psi$ liefert uns der modus ponens schließlich $\neg\psi$, wie gewünscht. ■

Damit ist der Beweis unseres Hauptergebnisses relativ einfach geworden.

Lemma 3.33 Eine Theorie ist genau dann widerspruchsfrei, wenn sie erfüllbar ist.

Beweis Es folgt aus dem Korrektheitsatz, dass eine Theorie, die ein Modell hat, widerspruchsfrei sein muss. Wir müssen zeigen, dass umgekehrt jede widerspruchsfreie Theorie T ein Modell hat. Wir führen zunächst für jede Formel φ mit einer freien Variable eine neue Konstante ein und fügen mittels Lemma 3.32 die entsprechenden Sätze hinzu. Sie ist dann noch keine Henkintheorie, denn wegen der neuen Konstanten gibt es neue Formeln. Wir wiederholen das unendlich oft und nehmen die Vereinigung über die so entstehende aufsteigende Kette von Theorien. Das Ergebnis ist eine Henkintheorie. Dann machen wir die Theorie mit Lemma 3.28 vollständig. Sie bleibt dabei eine Henkintheorie, und als vollständige Henkintheorie hat sie ein Modell. Dieses ist auch ein Modell der ursprünglichen Theorie. ■

Definition 3.34 Sei T eine σ -Theorie und ψ ein σ -Satz. Wir sagen, T beweist ψ , und schreiben $T \vdash \psi$, falls es $\varphi_1, \dots, \varphi_n \in T$ gibt, so dass $\neg\Lambda^n\varphi_1 \dots \varphi_n \neg\psi$ beweisbar ist. Wir sagen, T impliziert ψ und schreiben $T \models \psi$, falls für jede σ -Struktur M mit $M \models T$ auch $M \models \psi$ gilt.

Es folgt aus dem Korrektheitssatz, dass eine Theorie nur solche Sätze beweist, die auch aus ihr folgen: Falls $M \models T$ und $T \vdash \psi$, so auch $M \models \psi$.

Satz 3.35 (Gödelscher Vollständigkeitssatz) *Jeder allgemeingültige Satz ist beweisbar. Eine Theorie beweist jeden Satz, der aus ihr folgt: Wenn für alle $M \models T$ auch $M \models \varphi$ gilt, dann gilt auch $T \vdash \varphi$.*

Beweis Die erste Behauptung ist einfach das Lemma, angewandt auf eine Theorie, die aus einem einzigen Satz besteht. Zur zweiten Behauptung: Wenn $T \vdash \varphi$ nicht gilt, dann ist $T \cup \{\neg\varphi\}$ widerspruchsfrei, hat also ein Modell M . ■

Satz 3.36 (Kompaktheitssatz) *Eine Theorie ist genau dann erfüllbar, wenn jede endliche Teilmenge erfüllbar ist.*

Beweis Widerspruchsfreiheit hat offensichtlich diese Eigenschaft, und nach dem Lemma ist Erfüllbarkeit = Widerspruchsfreiheit. ■

Korollar 3.37 (Absteigender Satz von Löwenheim-Skolem) *Wenn σ eine höchstens abzählbare Signatur ist, dann hat jede σ -Theorie ein höchstens abzählbares Modell.*

Beweis Bei dem Prozess im Beweis des Vollständigkeitssatzes bleibt die Signatur abzählbar. Weil das so konstruierte Modell höchstens soviele Elemente hat wie Konstanten in der Signatur existieren, ist es höchstens abzählbar. ■

Übungsaufgabe 3.23 Sei σ_{Ab} die Signatur mit $\sigma_{Ab}^{Op} = \{+, -, \mathbf{0}\}$, $\sigma_{Ab}^{Rel} = \emptyset$, $ar_{Ab}(+) = 2$, $ar_{Ab}(-) = 1$ und $ar_{Ab}(\mathbf{0}) = 0$.

Geben Sie eine σ_{Ab} -Theorie T an, so dass für jede σ_{Ab} -Struktur M gilt: M ist eine abelsche Gruppe (mit Gruppenoperation $+^M$, Inversenabbildung $-^M$ und neutralem Element $\mathbf{0}^M$) genau dann, wenn $M \models T$.

Betrachten Sie die σ_{Ab} -Struktur M mit $\underline{M} = \{a, b\}$, wobei $a \neq b$, $+^M(a, a) = +^M(a, b) = +^M(b, b) = a$, $+^M(b, a) = b$, $-^M(a) = -^M(b) = b$ sowie $\mathbf{0}^M = a$. Zeigen Sie, dass $M \not\models T$.

Übungsaufgabe 3.24 Sei σ die Signatur mit einem einzigen einstelligen Operationssymbol f . Was kann man in jedem der folgenden Fälle über die Anzahl der Elemente eines beliebigen Modells $M \models T$ sagen?

- $T = \{\exists x \exists x \neg = x x\}$.
- $T = \{\exists x \exists x \exists x \wedge \wedge \wedge \neg = x x \neg = x x \neg = x x\}$.
- $T = \{\neg \exists x \exists x \exists x \wedge \wedge \wedge \neg = x x \neg = x x \neg = x x, \exists x \exists x \neg = x x\}$.
- $T = \{\neg \exists x \neg = x f f x, \neg \exists x = x f x\}$.
- $T = \{\neg \exists x \exists x \wedge = f x f x \neg = x x, \exists x \neg \exists x = x f x\}$. (Hinweis: Eine Abbildung zwischen zwei endlichen Mengen ist injektiv genau dann, wenn sie surjektiv ist.)

- T besteht nur aus wahren Sätzen der Mengenlehre.
- T kann unendlich sein, lässt sich aber in einem Buch (d.h. auf endlichem Raum) eindeutig beschreiben in der Art, dass man im Prinzip alle Sätze in T (Axiome) aufzählen kann.
- T ist widerspruchsfrei.
- T ist vollständig.

Ein wichtiges Ziel war es, die Widerspruchsfreiheit von T in dem Formalismus selbst auszudrücken und zu beweisen. Der Unvollständigkeitssatz zeigt, dass das unmöglich ist.

Definition 3.38 Sei σ_N die Signatur mit $\sigma_N^{\text{Op}} = \{\mathbf{0}, \mathbf{1}, +, \cdot\}$, $\sigma_N^{\text{Rel}} = \{\prec\}$ und den üblichen Stelligkeiten. Wir fassen \mathbb{N} auf die offensichtliche Weise als σ_N -Struktur auf. Für jede natürliche Zahl $n \in \mathbb{N}$ sei \underline{n} der σ_N -Term $+\mathbf{n}\mathbf{0}\mathbf{1}^n$. Für jede σ_N -Formel φ mit freien Variablen x_1, \dots, x_k setzen wir $\varphi(\mathbb{N}^k) = \{(n_1, \dots, n_k) \in \mathbb{N}^k \mid \mathbb{N} \models \varphi[\underline{n_1}/x_1] \dots [\underline{n_k}/x_k]\}$.

Eine Menge $X \subseteq \mathbb{N}^k$ von k -Tupeln natürlicher Zahlen heißt arithmetisch, falls sie von der Form $\varphi(\mathbb{N}^k)$ für eine σ_N -Formel φ ist. Eine Funktion heißt arithmetisch, falls ihr Graph es ist.

Übungsaufgabe 3.29 (Gödelsche β -Funktion) Für $a, b, i \in \mathbb{N}$ sei $\beta(a, b, i)$ die kleinste natürliche Zahl $n \in \mathbb{N}$, so dass $n \equiv a$ modulo $bi + 1$ ist. Zeigen Sie mit Hilfe des chinesischen Restsatzes, dass es für jedes endliche Tupel $(c_1, \dots, c_k) \in \mathbb{N}^*$ ein Paar $(a, b) \in \mathbb{N}^2$ gibt, so dass $\beta(a, b, i) = c_i$ ist für $i = 1, \dots, k$. Man kann also jedes solche Tupel durch das Tripel (a, b, n) codieren. (Hinweis: Wählen Sie $b = k!$ oder ein Vielfaches, so dass $b > c_i$ für alle i ist.)

Proposition 3.39 Alle rekursiven Mengen sind arithmetisch.

Beweis Es genügt zu zeigen, dass die arithmetischen Funktionen unter den Bedingungen in der Definition der rekursiven Funktionen abgeschlossen sind. Das einzige Problem dabei ist die primitive Rekursion. Diese kann man aber mit Hilfe der β -Funktion aus Übungsaufgabe 3.29 auf die μ -Rekursion zurückführen. ■

Korollar 3.40 Alle rekursiv aufzählbaren Mengen sind arithmetisch.

Beweis Jede rekursiv aufzählbare Menge $A \subseteq \mathbb{N}^k$ ist laut Definition Projektion einer rekursiven Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$. Letztere ist nach der Proposition arithmetisch, d.h. von der Form $\bar{A} = \varphi(\mathbb{N}^{k+1})$. Also ist $A = \psi(\mathbb{N}^k)$, wobei $\psi = \exists \mathbf{x} \varphi$ ist. ■

Wir verwenden ab jetzt wieder wie in Satz 3.24 die Codierung $\ulcorner \varphi \urcorner \in \mathbb{N}$ von σ_N -Formeln. Damit können wir eine Theorie T rekursiv, rekursiv aufzählbar usw. nennen, je nachdem, ob $\{\ulcorner \varphi \urcorner \mid \varphi \in T\}$ es ist.

Korollar 3.41 Es gibt kein Computerprogramm, das für jeden σ_N -Satz φ entscheiden kann, ob $\mathbb{N} \models \varphi$ gilt oder nicht.

Beweis Wenn $T = \{\ulcorner \varphi \urcorner \mid \mathbb{N} \models \varphi\}$ rekursiv wäre, dann wäre auch für jede σ_N -Formel φ mit einer freien Variable x die Menge $\varphi(\mathbb{N}) = \{n \in \mathbb{N} \mid \varphi[n/x] \in T\}$ rekursiv. Es gibt aber rekursiv aufzählbare Mengen, die nicht rekursiv sind. Weil eine solche Menge arithmetisch ist, gibt es σ_N -Formeln φ , so dass $\varphi(\mathbb{N})$ nicht rekursiv ist. ■

Damit erhalten wir eine schwache Form des von Kurt Gödel 1931 veröffentlichten 1. Unvollständigkeitssatzes:

Satz 3.42 (Unvollständigkeitssatz) *Jede rekursive σ_N -Theorie, die nur aus Sätzen besteht, die für die natürlichen Zahlen auch gelten, ist unvollständig in dem starken Sinn, dass es einen Satz φ gibt, so dass weder $T \vdash \varphi$ noch $T \vdash \neg\varphi$ gilt.*

Übungsaufgabe 3.30 *Zeigen Sie mit einem Diagonalargument direkt, dass $\{\ulcorner \varphi \urcorner \mid \mathbb{N} \models \varphi\}$ noch nicht einmal arithmetisch ist. (Hinweis: Betrachten Sie die arithmetische Menge $U = \{(e, n) \in \mathbb{N}^2 \mid e = \ulcorner \varphi \urcorner \text{ und } \mathbb{N} \models \varphi[n/\overset{\circ}{x}]\}$, wobei φ für Formeln steht, die nur $\overset{\circ}{x}$ als freie Variablen haben.)*

Übungsaufgabe 3.31 *Auch die ganzen Zahlen kann man als eine σ_N -Struktur \mathbb{Z} auffassen. Überlegen Sie sich, dass die vollständige Theorie der ganzen Zahlen, $\{\varphi \mid \mathbb{Z} \models \varphi\}$, ebenfalls nicht rekursiv ist. Für die reellen Zahlen gilt das übrigens nicht. Warum funktioniert das Argument in diesem Fall nicht?*

Übungsaufgabe 3.32 (Aufsteigender Satz von Löwenheim-Skolem) *Folgern Sie aus dem Kompaktheitssatz: Wenn eine Theorie T ein unendliches Modell hat und A eine beliebige Menge ist, dann hat T auch ein Modell, dessen Grundmenge mindestens so groß ist wie A . (Hinweis: Erweitern Sie die Signatur um so viele Konstanten, wie A Elemente hat.)*

Übungsaufgabe 3.33 *Sei σ eine endliche Signatur und T eine rekursiv aufzählbare σ -Theorie. Zeigen Sie, dass T „im Wesentlichen“ sogar rekursiv ist, d.h. es gibt eine rekursive Theorie T' , die dieselben Modelle hat wie T . (Hinweis: $\ulcorner \varphi \urcorner \in A \iff \exists n: (\ulcorner \varphi \urcorner, n) \in \bar{A}$ in der Notation von Definition 2.29. Betrachten Sie die Theorie $T' = \{\ulcorner \varphi \urcorner \mid \exists n: (\ulcorner \varphi \urcorner, n) \in \bar{A}\}$.)*

Übungsaufgabe 3.34 *Sei σ eine endliche Signatur, M eine endliche σ -Struktur und $T = \{\varphi \mid M \models \varphi\}$ die vollständige σ -Theorie von M . Überlegen Sie sich, wie Sie beweisen würden, dass T rekursiv ist.*