Grundbegriffe der mathematischen Logik

Hans Adler
Kurt Gödel Research Center
Universität Wien

Sommersemester 2012
Version vom 15. April 2013

Inhaltsverzeichnis

Kapitel 1

Primitive Rekursivität

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

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

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

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

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

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

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

1.1 Primitiv rekursive und LOOP-berechenbare Funktionen

Primitiv rekursive Funktionen

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

induktive Definition

primitiv rekursive Funktion

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

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

Nachfolgerfunktion S

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

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

  gegebene Funktion $h\colon \mathbb{N}^k \to \mathbb{N}$ in F. Wir schreiben auch $h = f \circ (g_1 \times g_2 \times \ldots \times g_k)$.[5]

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

primitive Rekursion

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

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

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

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

Proposition 1.3 *Die folgenden Funktionen sind primitiv rekursiv:*

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

Fakultätsfunktion

Beweis f_1 kann man durch primitive Rekursion wie folgt definieren:

$$\begin{aligned} f_1(x, 0) &= x \\ f_1(x, y+1) &= S(f_1(x, y)) = (S \circ \pi^3_3)(x, y, f_1(x, y)) \end{aligned}$$

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

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

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

$$\begin{aligned} f_2(x,0) &= 0 \\ &= 0() \\ f_2(x,y+1) &= f_1(f_2(x,y),x) \\ &= (f_1 \circ (\pi_3^3 \times \pi_1^3))(x,y,f_2(x,y)) \end{aligned} \qquad \begin{aligned} f_3(x,0) &= 1 \\ &= (S \circ 0)() \\ f_3(x,y+1) &= f_2(f_3(x,y),x) \\ &= (f_2 \circ (\pi_3^3 \times \pi_1^3))(x,y,f_3(x,y)) \end{aligned}$$

Zuletzt noch f_4:

$$\begin{aligned} f_4(0) &= 1 \\ f_4(y+1) &= f_2(f_4(y),S(y)) \\ &= (f_2 \circ (\pi_2^2 \times (S \circ \pi_1^2)))(y,f_4(y)). \end{aligned}$$

∎

LOOP-Programme

LOOP-Programm

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

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

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

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

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

`loop ... times` Schleife

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

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

Schleifenkörper

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

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

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

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

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

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

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

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

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

Primitiv rekursive Funktionen sind LOOP-berechenbar

LOOP-berechenbare Funktion

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

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

induktiver Beweis

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

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

```
output = Zero()
```

Ein LOOP-Programm, das die Nachfolgerfunktion berechnet:

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

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

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

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

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

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

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

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

Wir beginnen mit einem LOOP-Programm (angedeutet durch drei Punkte), das f berechnet, ohne die Werte der Variablen `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 Werte der Variablen `input_1`, `input_2` etc. und `y` nicht verändern.) ∎

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

`if ... then`

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

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

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

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

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

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

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

Übungsaufgaben

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Äquivalenz von primitiver Rekursivität und LOOP-Berechenbarkeit

Cantorsche Paarungsfunktion und simultane primitive Rekursion

Proposition 1.6 *Die folgenden Funktionen sind primitiv rekursiv:*

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

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

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

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

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

Fallunterscheidung

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Beweis $\Rightarrow$: Wir können einfach $g = h$ wählen.
$\Leftarrow$: $h(\bar{x}) = \min\{y \leq g(\bar{x}) \mid y = g(\bar{x}) \text{ oder } f(\bar{x}, y) > 0\}$. ∎

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

Beweis Zunächst einmal ist $h(x, y) \leq x$. Für $y > 0$ ist $\lceil \frac{x}{y} \rceil = \mu z(yz \geq x) = \mu z(yz - x \geq 0)$. Also ist generell $h(x, y) = \mu z(y(yz - x) \geq 0) = \mu z((1 + yyz) \dot{-} yx > 0)$. ∎

Der Beweis lässt sich offensichtlich verallgemeinern: Umkehrfunktionen von primitiv rekursiven Funktionen sind immer primitiv rekursiv, es sei denn, sie lassen sich nicht durch primitiv rekursive Funktionen nach oben abschätzen.

Satz 1.11 (Cantorsche Paarungsfunktion)[9] *Die Funktion* Cantorsche Paarungsfunktion J, L, R

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

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

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

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

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

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

$$\begin{aligned} R'(z, x) &= \min\{y \leq z \mid y = z \text{ oder } J(x, y) = z\} \\ &= \mu y(J(x, y) = z) \end{aligned}$$

nach Proposition 1.8 sicher primitiv rekursiv. Weil $R(z) \leq z$ ist, ist $R'(z, x) = R(z)$ falls $x = L(z)$ ist. Daraus folgt zusammen mit $L(z) \leq z$, dass

$$\begin{aligned} L(z) &= \min\{x \leq z \mid z = J(x, R'(z, x))\} \\ &= \mu x(J(x, R'(x, z)) = x). \end{aligned}$$

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

Simultane primitive Rekursion

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

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

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

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

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

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

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

Beweis Wir zeigen nur 1, weil der allgemeinere Fall 2 im Wesentlichen genauso geht. Im Fall $m = 2$ wenden wir die Cantorsche Paarungsfunktion[9] J und ihre Umkehrfunktionen L und R an. Die Funktion $h(\bar{x}) = J(h_1(\bar{x}), h_2(\bar{x}))$ ist primitiv rekursiv, weil wir sie wie folgt durch primitive Rekursion erhalten:

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

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

Für den Fall $m > 3$ können wir J, L und R geeignet zusammensetzen, um m-Tupel von natürlichen Zahlen zu codieren und decodieren. ∎

Satz 1.13 *Die primitiv rekursiven Funktionen sind genau diejenigen Funktionen, die sich durch ein LOOP-Programm berechnen lassen.*

Beweis $\Rightarrow$: Nach Satz 1.5 lassen sich alle primitiv rekursiven Funktionen durch ein LOOP-Programm berechnen. $\Leftarrow$: Spezialfall des folgenden Lemmas. ∎

Lemma 1.14 *Wenn man ein LOOP-Programm ausführt, dann ergibt sich der Endwert jeder Variablen durch eine primitiv rekursive Funktion aus den Anfangswerten aller Variablen.*

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

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

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

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

Beispiel

Satz 1.13 macht es relativ einfach, zu zeigen, dass eine gegebene Funktion primitiv rekursiv ist: Dazu genügt es, ein LOOP-Programm anzugeben, das die Funktion berechnet. Im Beweis, dass jede LOOP-berechenbare Funktion primitiv rekursiv ist, haben wir keine besonderen Eigenschaften der durch `Inc`, `Dec`, `Val` und `Zero` gebraucht, sondern lediglich die Tatsache, dass es sich jeweils um primitiv rekursive Funktionen handelt. Es ist also harmlos, bei der LOOP-Programmierung zusätzliche Befehle zuzulassen, mit denen jeweils weitere primitiv rekursive bzw. (das ist ja äquivalent) LOOP-berechenbare Funktionen berechnet werden. Wir sprechen von erweiterten LOOP-Programmen.

Erweitertes LOOP-Programm

Als Beispiel leiten wir aus der Cantorschen Paarungsfunktion J eine Bijektion zwischen $\mathbb{N}$ und $\mathbb{N}^* = \bigcup_{k\in\mathbb{N}} \mathbb{N}^k$ ab und zeigen, dass alle damit zusammenhängenden Funktionen primitiv rekursiv sind.

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

J_k

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

$\langle x_0, \ldots, x_{k-1}\rangle$

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

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

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

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

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

Beweis Dass die Einschränkungen von $\langle\cdot\rangle$ auf $\mathbb{N}^k$ primitiv rekursiv sind, beweist man einfach über die primitive Rekursivität von J_k, die sich durch Induktion nach k ergibt. Noch einfacher sieht man, dass Lg primitiv rekursiv ist:

$$\mathrm{Lg}(z) = \begin{cases} L(z-1) + 1 & \text{falls } z > 0 \\ 0 & \text{sonst.} \end{cases}$$

Für Component geben wir ein erweitertes LOOP-Programm an.

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

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

index_is_last = Eq(input_2, maxindex)
if index_is_last then {
```

```
  x = Val(input_1)
  loop input_2 times {
    x = R(x)
  }
  output = R(x)
}
```

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

Die Funktion Replace wird durch das folgende erweiterte LOOP-Programm berechnet.

```
x = input_1
i = input_2
y = input_3
k = Lg(x)
i_plus_1 = Inc(i)
new_k = Max(k, i_plus_1)
n = Dec(new_k)
output = Component(x,n)
loop n times {
  n = Dec(n)
  z = Component(x,n)
  must_switch_value = Eq(n,i)
  if must_switch_value then {
    z = Val(y)
  }
  output = J(z,output)
}
new_k_minus_1 = Dec(new_k)
output = J(new_k_minus_1,output)
output = Inc(output)
```

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

Übungsaufgaben

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

Übungsaufgabe 1.9 *Sei* $f\colon \mathbb{N} \to \mathbb{N}$ *eine streng monotone primitiv rekursive Funktion. Zeigen Sie, dass auch* $g(y) = \mu x(f(x) \geq y)$ *primitiv rekursiv ist.*

Übungsaufgabe 1.10 *Gegeben ein* $k \in \mathbb{N}$ *betrachten Sie die primitiv rekursive Bijektion* $J_k\colon \mathbb{N}^k \to \mathbb{N}$ *aus Definition 1.15 und zeigen Sie, dass auch die Komponenten der Umkehrfunktion primitiv rekursiv sind.*

Übungsaufgabe 1.11 *Die Definition der LOOP-Programmiersprache ist für mathematische Verhältnisse nicht sehr präzise. In der Informatik ist es üblich, Programmiersprachen in der EBNF (erweiterten Backus-Naur-Form) zu definieren, was aber mangels Festlegung gewisser Details leider auch nicht viel genauer ist. Der Grund ist wohl, dass der hohe Aufwand einer präzisen Definition in keinem vernünftigen Verhältnis zum Nutzen steht. Hier ist eine EBNF für erweiterte LOOP-Programme.*

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

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

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

Spezialisieren Sie die angegebene EBNF auf LOOP-Programme im eigentlichen Sinn, d.h. nur mit `Zero`, `Val`, `Inc`, `Dec`. *Geben Sie dabei in ⟨right-value⟩ auch die Stelligkeit genau vor.*

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

1.3 Hyperoperatoren und Ackermannfunktion

In den Zwanzigerjahren des 20. Jahrhunderts fragte David Hilbert, ob die primitiv rekursiven Funktionen den intuitiven Begriff der Berechenbarkeit genau beschreiben. Schon kurze Zeit später fanden jedoch seine Schüler Wilhelm Ackermann und Gabriel Sudan Funktionen, deren Werte man mit Papier und Bleistift im Prinzip alle ausrechnen kann, die jedoch nicht primitiv rekursiv sind. Bei der ursprünglichen, dreistelligen Ackermannfunktion handelt es sich um eine Variante der Hyperoperatoren, wie wir sie in diesem Abschnitt betrachten werden.[10]

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

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

Wie man sieht, sind die Definitionen im Fall $y = 0$ (rechts) jeweils unterschiedlich, aber im Fall $y' = y + 1$ (links) kann man bei genauerem Hinschauen ein einheitliches Prinzip erkennen. Die *Hyperoperatoren* erhalten wir, indem wir die Folge nach demselben Prinzip weiter fortsetzen, wobei wir für jeden der neuen Operatoren im Fall $y = 0$ genauso vorgehen, wir für die Exponentation. Für die Hyperoperatoren wurden viele verschiedene Notationen vorgeschlagen, aber die von Donald Knuth[11] eingeführte suggestive Schreibweise $x \uparrow^n y$ gehört sicherlich zu den bekanntesten und praktischsten. Es ist $x \uparrow^0 y = x \cdot y$ und $x \uparrow^1 y = x^y$, und die allgemeine Definition geht wie folgt.

Hyperoperatoren $\uparrow$

Definition 1.17 (Hyperoperatoren in Knuthscher Pfeilnotation)

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

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

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

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

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

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

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

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

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

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

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

Abbildung 1.1: Ackermann-Péter-Funktion

Ziel dieses Abschnitts ist der Beweis des folgenden Satzes. Weil es natürlich im Prinzip[12] ein Computerprogramm gibt, das für beliebige Eingaben x, y, n den Wert $x \uparrow^n y$ berechnet, zeigt der Satz, dass nicht alle im anschaulichen Sinne berechenbaren Funktionen primitiv rekursiv sind. Das motiviert die Definition der rekursiven Funktionen später in Kapitel 3.

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

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

Definition 1.21 *Die Ackermann-Péter-Funktion* $A\colon \mathbb{N}^2 \to \mathbb{N}$, *meist nicht ganz zutreffend als Ackermannfunktion*[13] *bezeichnet, ist durch die folgenden Gleichungen definiert.*

Ackermann-Péter-Funktion $A(n, x)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Übungsaufgaben

Übungsaufgabe 1.12 *Die ursprüngliche Ackermannfunktion ist durch die folgenden Rekursionsgleichungen definiert.*

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

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

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

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

Übungsaufgabe 1.13 *Geben Sie eine Tabelle der exakten Werte von $2 \uparrow^{n-2} x$ an für alle Paare $(n, x) \in \mathbb{N} \times \{0, 1, \ldots, 10\}$, für die das praktikabel ist. Erweitern Sie die Tabelle, indem Sie, soweit praktikabel, für große Funktionswerte die Zahl der Dezimalstellen grob abschätzen ($2^{10} = 1024 \approx 10^3$).*

Übungsaufgabe 1.14 *Sei n die größte natürliche Zahl, die man mit einem deutschen Satz von höchstens zwanzig Wörtern definieren kann. Kann man entscheiden, ob n gerade oder ungerade ist? (Hinweis: Zählen Sie die Wörter im ersten Satz dieser Aufgabe.)*

Kapitel 2

Elemente der Prädikatenlogik

Wir kommen jetzt zur Logik im engeren Sinne, und hier insbesondere zur Prädikatenlogik. Das Wesen der mathematischen Logik besteht darin, mathematische Objekte und Sachverhalte (Semantik) in einer formalen Sprache (Syntax) zu beschreiben und auf diese Weise zu neuen Erkenntnisse über dieselben zu gewinnen. Die Wörter/Sätze der formalen Sprache heißen meist Terme, Formeln oder Sätze. Ein vertrauter Spezialfall dieses allgemeinen Prinzips sind die Polynome aus der Algebra. Ein weiterer Spezialfall sind die Linearkombinationen aus der linearen Algebra. Diese beiden Beispiele werden wir im ersten Abschnitt verallgemeinern.

Erst danach werden wir im zweiten Abschnitt mit Hilfe von Termen und sogenannten Prädikaten allgemeinere Aussagen (Formeln) über die Elemente der jeweils betrachteten Grundmenge bilden und diese auch mit logischen Konnektoren wie „und“, „oder“ miteinander verknüpfen.

2.1 Begriffe aus der universellen Algebra

Definition 2.1 *Eine* funktionale Struktur *M besteht aus einer Menge $\underline{M}$ (der* Grundmenge *von M) zusammen mit ausgezeichneten, benannten Operationen. Die Operationen sind Funktionen der Form $\underline{M}^k \to \underline{M}$ für $k \in \mathbb{N}$. Die Zahl k heißt hierbei die Stelligkeit der Operation. Nullstellige Operationen entsprechen ausgezeichneten Elementen von $\underline{M}$ und werden daher als Konstanten bezeichnet.*

funktionale Struktur
Grundmenge $\underline{M}$
Operationen
Konstanten

Hier einige Beispiele:

- Die Struktur $(\mathbb{N}, 0, 1, +, \times)$ mit der Grundmenge $\mathbb{N}$, den nullstelligen Operationen $0 \in \mathbb{N}$ und $1 \in \mathbb{N}$ sowie den zweistelligen Operationen $+\colon \mathbb{N}^2 \to \mathbb{N}$ und $\times\colon \mathbb{N}^2 \to \mathbb{N}$.
- Die Struktur $(\mathbb{N}, 0, S)$ mit der Grundmenge $\mathbb{N}$, der nullstelligen Operation 0 und der einstelligen Operation $S\colon \mathbb{N} \to \mathbb{N}$, $x \mapsto x + 1$.
- 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 $-\colon \mathbb{N} \to \mathbb{N}$ (also $x \mapsto -x$), sowie den zweistelligen Operationen $+\colon \mathbb{N}^2 \to \mathbb{N}$ und $\times\colon \mathbb{N}^2 \to \mathbb{N}$.
- 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 $-\colon \mathbb{N} \to \mathbb{N}$ (also $x \mapsto -x$), sowie den zweistelligen Operationen $+\colon \mathbb{N}^2 \to \mathbb{N}$ und $\times\colon \mathbb{N}^2 \to \mathbb{N}$.
- Für jede Gruppe G die Struktur $(G, 1, \times, ^{-1})$, wobei 1 das neutrale Element von G ist, $\times$ die Gruppenoperation und $^{-1}$ die Abbildung, die jedes Element auf sein Inverses abbildet.
- Für jede abelsche Gruppe G die Struktur $(G, 0, +, -)$, wobei 0 das neutrale Element von G ist, $+$ die Gruppenoperation und $-$ die (einstellige) Abbildung, die jedes Element auf sein Inverses abbildet.

Bei den letzten beiden Beispielen ist zu beachten, dass im Falle einer abelschen Gruppe G die beiden Strukturen $(G, 1, \times, ^{-1})$ und $(G, 0, +, -)$ formal verschieden sind, weil die Operationen unterschiedlich benannt sind.

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 an, also die für sie benutzten Symbole. Andererseits stehen sie aber auch für die jeweiligen Operationen selbst. Eigentlich müsste man dazwischen unterscheiden, weil man sonst z.B. Schwierigkeiten bekommt, über eine Struktur zu sprechen, deren Grundmenge $\mathbb{N}$ ist und die eine mit $+$ benannte Operation hat, die durch $+(x, y) = 2^{x^2+y!} - x$ definiert ist.

Definition 2.2 *Die* Signatur *einer Struktur besteht aus den Namen der ausgezeichneten Operationen zusammen mit der Information, wieviele Stellen jede hat. Die Elemente der Signatur heißen* Operationssymbole.

Signatur σ
Operationssymbol
Redukt

Eine Struktur, die man durch Weglassen von einigen der benannten Operationssymbole 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 Paar $\sigma = (\sigma^{\mathrm{Op}}, \mathrm{ar})$, wobei σ^{Op} eine beliebige Menge ist (deren Elemente dann Operationssymbole heißen) und $\mathrm{ar}\colon \sigma^{\mathrm{Op}} \to \mathbb{N}$ die Stelligkeiten der Symbole angibt. Manchmal nehmen wir es genau und unterscheiden zwischen dem (k-stelligen) Operationssymbol f und der zugehörigen Operation $f^M\colon \underline{M}^k \to \underline{M}$. Das ist vor allem dann sinnvoll, wenn wir zwei verschiedene Strukturen derselben Signatur betrachten, oder allgemeiner zwei Strukturen, deren Signaturen sich überschneiden.

σ^{Op}
ar

Definition 2.3 *Ein* Homomorphismus $H\colon M \to N$ *von einer Struktur M zu einer Struktur N derselben Signatur ist eine Abbildung $H\colon \underline{M} \to \underline{N}$ zwischen den Grundmengen, die mit den Operationen vertauscht. Das heißt:*

Homomorphismus

Für jedes k-stellige Operationssymbol f und jedes k-Tupel $(a_1, \ldots, a_k) \in \underline{M}^k$ gilt $H(f(a_1, \ldots, a_k)) = f(H(a_1), \ldots, H(a_k))$.

Zwei Strukturen heißen isomorph*, falls sie dieselbe Signatur haben und es einen Isomorphismus zwischen ihnen gibt, d.h. einen bijektiven Homomorphismus.*

Definition 2.4 *Eine* Substruktur *einer Struktur M ist eine Struktur N mit derselben Signatur, so dass $\underline{N} \subseteq \underline{M}$ gilt und die Operationen von N sich durch Einschränkung der Operationen von M ergeben.*

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

Definition 2.5 *Die Menge der σ-*Terme *ist die kleinste Sprache*[1] *über dem Alphabet $\sigma^{\mathrm{Op}} \cup \mathbb{X}$, welche die folgenden Bedingungen erfüllt.*

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

Lemma 2.6 (Eindeutige Lesbarkeit von Termen) *Jeder σ-Term ist entweder eine Variable oder lässt sich in der Form $ft_1 \dots t_k$ schreiben, wobei $f \in \sigma^{\mathrm{Op}}$ 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.*

Beweis Die Menge aller Zeichenketten, die sich überhaupt in der beschriebenen Form schreiben lassen, erfüllt die Bedingungen aus der Definition der σ-Terme und umfasst daher die Menge der σ-Terme. Es bleibt zu zeigen, dass die Darstellung eindeutig ist. Seien also $ft_1 \dots t_k = f't'_1 \dots t'_{k'}$ zwei solche Darstellungen. Zunächst ist klar, dass $f = f'$ und $k = k'$ sein muss. Nach dem folgenden Lemma kann es aber kein kleinstes i geben, so dass $t_i \neq t'_i$ ist. ∎

Lemma 2.7 *Kein σ-Term ist echtes Anfangsstück eines anderen σ-Terms.*

Beweis Sei s ein Anfangsstück von t. Durch Induktion über die Länge von t zeigen wir, dass $s = t$ ist. Wenn s eine Variable ist, dann muss t ebenfalls diese Variable sein, also $s = t$. Sonst ist $s = fs_1 \dots s_k$ und $t = ft_1 \dots t_k$, wobei f ein k-stelliges Operationssymbol ist. Wenn $s \neq t$ wäre, dann gäbe es ein kleinstes i, so dass $s_i \neq t_i$, und für dieses wäre s_i ein echtes Anfangsstück von t_i oder umgekehrt. Nach Induktionsvoraussetzung ist das unmöglich. ∎

Definition 2.8 *Eine* Belegung *der Variablen in einer Struktur M ist eine Abbildung $\beta\colon \mathbb{X} \to \underline{M}$. Wenn σ die Signatur von M ist, setzen wir eine Belegung β in M wie folgt rekursiv zu einer Abbildung $\bar{\beta}\colon \{t \mid t \text{ ist } \sigma\text{-Term}\} \to \underline{M}$ fort:*

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

Definition 2.9 (Substitution) *Seien σ eine funktionale Signatur, s, t σ-Terme und $x \in \mathbb{X}$ eine Variable. Dann ist $t[\frac{s}{x}]$ die Zeichenkette, die man erhält, indem man jedes Vorkommen von x durch s ersetzt.*

Seien σ eine funktionale Signatur, t ein σ-Term, $x \in \mathbb{X}$ eine Variable, M eine σ-Struktur, $a \in \underline{M}$ und $\beta\colon \mathbb{X} \to \underline{M}$ eine Belegung in M. Dann ist die Belegung $\beta[\frac{a}{x}]$ gegeben durch

$$\beta\left[\frac{a}{x}\right](u) = \begin{cases} a & \text{falls } u = x \\ \beta(u) & \text{sonst.} \end{cases}$$

Man zeigt leicht durch Induktion über t, dass auch $t[\frac{s}{x}]$ wieder ein σ-Term ist.

Lemma 2.10 (Substitutionslemma für Terme) *Seien σ eine funktionale Signatur, s, t σ-Terme, $x \in \mathbb{X}$ eine Variable, M eine σ-Struktur und $\beta\colon \mathbb{X} \to \underline{M}$ eine Belegung. Dann gilt $\bar{\beta}(t[\frac{s}{x}]) = \overline{\beta[\frac{\bar{\beta}(s)}{x}]}(t)$.*

[1] Eine Sprache über einem Alphabet A (d.h. einer Menge A, aufgefasst als Menge von Symbolen) ist eine beliebige Menge $L \subseteq A^* = \bigcup_{k \in \mathbb{N}} A^k$ von Wörtern über A, d.h. von endlichen Tupeln mit Werten in A.

Beweis Beweis durch Induktion über t. Wenn t eine Variable ist, ist die Behauptung klar. Wenn $t = ft_1 \dots t_k$ ist, ist

$$\bar{\beta}((ft_1 \dots t_k)\left[\frac{s}{x}\right]) = \bar{\beta}\left(ft_1\left[\frac{s}{x}\right] \dots t_k\left[\frac{s}{x}\right]\right) = f^M\left(\bar{\beta}\left(t_1\left[\frac{s}{x}\right]\right), \dots, \bar{\beta}\left(t_k\left[\frac{s}{x}\right]\right)\right)$$
$$= f^M\left(\overline{\beta\left[\frac{\bar{\beta}(s)}{x}\right]}(t_1), \dots, \overline{\beta\left[\frac{\bar{\beta}(s)}{x}\right]}(t_k)\right) = \bar{\beta}\left[\frac{\bar{\beta}(s)}{x}\right](t),$$

wobei die Gleichung zwischen der ersten und zweiten Zeile aus der Induktionsvoraussetzung folgt. ∎

Übungsaufgaben

Übungsaufgabe 2.1 *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 2.2 *Gegeben zwei funktionale Strukturen M und N derselben Signatur heißt eine Abbildung $h\colon \underline{M} \to \underline{N}$ 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.

Übungsaufgabe 2.3 *Bestimmen Sie jeweils alle Signaturen σ, so dass $\left|\sigma^{\mathrm{Op}}\right| = 2$ und t ein σ-Term ist. Manchmal gibt es keine Lösung oder mehrere Lösungen.*

- $t = \mathtt{f}\overset{11}{\mathtt{xxc}}$.
- $t = \overset{17}{\mathtt{f}}\mathtt{x}\overset{9}{\mathtt{g}}\mathtt{xc}$.
- $t = \overset{3}{\mathtt{f}}\mathtt{x}\overset{58}{\mathtt{g}}\mathtt{xx}$.
- $t = \texttt{----------}\texttt{----------}\texttt{----------}\texttt{----------}\overset{12}{\texttt{+}}\texttt{xx}$.
- $t = \texttt{++++++++++}\texttt{++++++++++}\texttt{++++++++++}\texttt{++++++++++}\overset{12}{\texttt{+}}\texttt{xx}$.
- $t = \texttt{++nn}\overset{2}{\texttt{x}}\texttt{+nn+nnn+nx}\overset{7}{\texttt{n}}$.
- $t = \texttt{AABB}\overset{0000}{\texttt{xxxx}}$.

Übungsaufgabe 2.4 *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 äquivalent 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.)*

2.2 Tarskis Definition der Wahrheit

Wir nehmen stillschweigend an, dass die speziellen Symbole $=$, $\exists$, $\neg$, $\wedge$, die wir im Folgenden benutzen werden, ebenso wie Variablennamen nie in einer Signatur auftreten. (Sollte das doch einmal geschehen, müssten wir zwischen dem Symbol als Teil der Signatur und dem Symbol der Prädikatenlogik unterscheiden wie bei disjunkten Vereinigungen.)

Definition 2.11 *Eine* Signatur σ *besteht aus einer Menge* σ^{Op} *von* Operationssymbolen *(auch* Funktionssymbole *genannt), einer Menge* σ^{Rel} *von* Relationssymbolen *(mit* $\sigma^{\mathrm{Op}} \cap \sigma^{\mathrm{Rel}} = \emptyset$*), sowie einer Abbildung* $\mathrm{ar}\colon \sigma^{\mathrm{Op}} \cup \sigma^{\mathrm{Rel}} \to \mathbb{N}$*, die jedem Symbol seine* Stelligkeit *zuordnet.*

Eine σ*-Struktur* M *besteht aus einer Grundmenge* $\underline{M} \neq \emptyset$ *sowie einer Funktion* $f^M\colon \underline{M}^k \to \underline{M}$ *für jedes* k*-stellige Operationssymbol* $f \in \sigma^{\mathrm{Op}}$ *und einer Teilmenge* $R^M \subseteq \underline{M}^k$ *für jedes* k*-stellige Relationssymbol* $R \in \sigma^{\mathrm{Rel}}$*.*

Gegenüber den funktionalen Signaturen und Strukturen aus dem letzten Abschnitt sind hier nur die Relationssymbole dazugekommen. Dadurch können wir beispielsweise eine lineare Ordnung in natürlicher Weise als Bestandteil einer Struktur auffassen.

Funktionale Signaturen und funktionale Strukturen können wir jetzt als den Spezialfall der allgemeinen Signaturen und Strukturen mit $\sigma^{\mathrm{Rel}} = \emptyset$ auffassen. Aus einer beliebigen Signatur oder Struktur erhalten wir eine funktionale, indem wir die Relationssymbole vergessen. Insbesondere werden wir auch für nicht funktionale Signaturen σ von σ-Termen sprechen. Die Relationssymbole sind im Zusammenhang mit Termen bedeutungslos, d.h. ein σ-Term ist dasselbe wie ein $(\sigma^{\mathrm{Op}}, \mathrm{ar}|_{\sigma^{\mathrm{Op}}})$-Term, und Relationssymbole können in Termen nicht vorkommen.

Definition 2.12 *Die Menge der* σ*-Formeln (der Prädikatenlogik 1. Stufe) ist die kleinste Sprache über dem Alphabet* $\sigma^{\mathrm{Op}} \cup \sigma^{\mathrm{Rel}} \cup \{=, \exists, \neg, \wedge\} \cup \mathbb{X}$*, welche die folgenden Bedingungen erfüllt.*

- *Wenn* t_1 *und* t_2 σ*-Terme sind, dann ist* $=t_1t_2$ *eine* σ*-Formel.*
- *Wenn* $R \in \sigma^{\mathrm{Rel}}$ *und* $k = \mathrm{ar}(R)$*, und* $t_1, \ldots, t_k$ *sind* σ*-Terme, dann ist* $Rt_1 \ldots t_k$ *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.*

Lemma 2.13 (Eindeutige Lesbarkeit von Formeln) *Jede* σ*-Formel lässt sich in genau einer der folgenden Weisen zerlegen:*

- *Als* $=t_1t_2$*, wobei* t_1, t_2 σ*-Terme sind.*
- *Als* $Rt_1 \ldots t_k$*, wobei* $R \in \sigma^{\mathrm{Rel}}$ *und* $k = \mathrm{ar}(R)$*, und* $t_1, \ldots, t_k$ *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* $t_1, \ldots, t_k$*, bzw.* φ *bzw.* φ, ψ *bzw.* x *und* φ *sind eindeutig bestimmt.*

Beweis Die Menge aller Zeichenketten, die sich überhaupt in der beschriebenen Form schreiben lassen, erfüllt die Bedingungen aus der Definition der σ-Formeln und umfasst daher die Menge der σ-Formeln. Es bleibt zu zeigen, dass die Darstellung eindeutig ist. Zunächst sehen wir, dass die erste Zerlegung nur möglich ist, wenn das erste Symbol der Formel $=$ ist, die zweite nur, wenn das erste Symbol ein Relationssymbol ist, usw. Daher lässt sich jede Formel nur gemäß höchstens (also genau) einem der Unterpunkte zerlegen.

Im Falle eines der ersten beiden Unterpunkte können wir Lemma 2.7 anwenden und genauso argumentieren wie im Beweis von Lemma 2.6: Wenn $=t_1t_2 = =t_1't_2'$ ist, dann ist t_1 ein Anfangsstück von t_1' oder t_1' ein Anfangsstück von t_1. In beiden Fällen folgt $t_1 = t_1'$ und dann auch $t_2 = t_2'$. Wenn $Rt_1 \ldots t_k = R't_1' \ldots t_k'$ ist, dann ist natürlich $R = R'$. Also ist t_1 ein Anfangsstück von t_1' oder t_1' ein Anfangsstück von t_1, woraus $t_1 = t_1'$ folgt. Folglich ist t_2 ein Anfangsstück von t_2' oder t_2' ein Anfangsstück von t_2, woraus $t_2 = t_2'$ folgt, usw.

Im Falle eines der letzten drei Unterpunkte wenden wir stattdessen Lemma 2.14 an und argumentieren wieder analog. ∎

Lemma 2.14 *Keine σ-Formel ist echtes Anfangsstück einer anderen σ-Formel.*

Beweis Sei φ ein Anfangsstück von ψ. Durch Induktion über die Länge von ψ (des längeren Stücks) zeigen wir, dass $\varphi = \psi$ ist.

Wenn $\varphi = {=}t_1t_2$ ist, dann beginnt auch ψ mit $=$, und folglich ist $\psi = {=}t_1't_2'$. Es ist klar, dass t_1 ein Anfangsstück von t_1' ist oder t_1' ein Anfangsstück von t_1. Nach Lemma 2.7 ist $t_1 = t_1'$. Folglich ist t_2 ein Anfangsstück von t_2' oder t_2' ein Anfangsstück von t_2. Wiederum mit Lemma 2.7 folgt $t_2 = t_2'$. Der Fall $\varphi = Rt_1 \dots t_k$ geht ganz ähnlich.

Wenn $\varphi = {\wedge}\varphi_1\varphi_2$ ist, dann beginnt auch ψ mit $\wedge$, und folglich ist $\psi = {\wedge}\psi_1\psi_2$. Es ist klar, dass φ_1 ein Anfangsstück von ψ_1 ist oder ψ_1 ein Anfangsstück von φ_1. Nach Induktionsvoraussetzung ist daher $\varphi_1 = \psi_1$. Folglich ist φ_2 ein Anfangsstück von ψ_2 oder ψ_2 ein Anfangsstück von φ_2. Wiederum nach Induktionsvoraussetzung ist daher $\varphi_2 = \psi_2$. Die Fälle $\varphi = {\neg}\psi$ und $\varphi = {\exists}x\psi$ gehen ganz ähnlich. ∎

Unter einer Belegung der Variablen in einer Struktur entspricht jedem Term ein Element (der Grundmenge) 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 2.15 (Tarskis Definition der Wahrheit) [2] *Wenn σ die Signatur von M ist, definieren wir für jede Belegung β in M wie folgt rekursiv eine Abbildung $\hat{\beta}\colon \{\varphi \mid \varphi \text{ ist } \sigma\text{-Formel}\} \to \{0,1\}$:*

- $\hat{\beta}({=}t_1t_2) = \begin{cases} 1 & \text{falls } \bar{\beta}(t_1) = \bar{\beta}(t_2) \\ 0 & \text{sonst.} \end{cases}$
- $\hat{\beta}(Rt_1 \dots t_k) = \begin{cases} 1 & \text{falls } (\bar{\beta}(t_1), \dots, \bar{\beta}(t_k)) \in R^M \\ 0 & \text{sonst.} \end{cases}$
- $\hat{\beta}({\neg}\varphi) = 1 - \hat{\beta}(\varphi)$.
- $\hat{\beta}({\wedge}\varphi\psi) = \hat{\beta}(\varphi) \cdot \hat{\beta}(\psi)$.
- $\hat{\beta}({\exists}x\varphi) = \begin{cases} 1 & \text{falls es ein Element } e \in \underline{M} \text{ gibt, so dass } \widehat{\beta[\frac{e}{x}]}(\varphi) = 1 \\ 0 & \text{sonst,} \end{cases}$
  wobei $\beta[\frac{e}{x}]$ (siehe Definition 2.9) sich von β nur durch $\beta(x) = e$ unterscheidet.

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

Die Belegungen sind ein leider notwendiges technisches Hilfsmittel. Nachdem wir mit ihrer Hilfe die Wahrheit definiert haben, können wir zeigen, dass es in bestimmten Fällen nicht auf sie ankommt. Dazu brauchen wir den Begriff eines σ-Satzes, d.h. einer σ-Formel ohne freie Variable.

Definition 2.16 *Die Menge der Variablen, die in einer σ-Formel frei vorkommen, ist wie folgt rekursiv definiert.*

- $\mathrm{Frei}({=}t_1t_2)$ *ist die Menge aller Variablen, die in t_1 oder t_2 vorkommen.*
- $\mathrm{Frei}(Rt_1 \dots t_k)$ *ist die Menge aller Variablen, die in $t_1, t_2, \dots$ oder t_k vorkommen.*
- $\mathrm{Frei}({\neg}\varphi) = \mathrm{Frei}(\varphi)$.
- $\mathrm{Frei}({\wedge}\varphi\psi) = \mathrm{Frei}(\varphi) \cup \mathrm{Frei}(\psi)$.
- $\mathrm{Frei}({\exists}x\varphi) = \mathrm{Frei}(\varphi) \setminus \{x\}$.

Lemma 2.17 *Sei σ eine Signatur, φ eine σ-Formel und M eine Struktur der Signatur σ. Seien $\beta_1, \beta_2\colon \mathbb{X} \to \underline{M}$ zwei Belegungen der Variablen in M. Falls β_1 und β_2 auf $\mathrm{Frei}(\varphi)$ übereinstimmen (d.h. falls $\beta_1(x) = \beta_2(x)$ für alle $x \in \mathrm{Frei}(\varphi)$), dann ist $\hat{\beta}_1(\varphi) = \hat{\beta}_2(\varphi)$.*

[2] Alfred Tarski (1901–1983) bezeichnete sich selbst mit einiger Berechtigung als “the greatest living sane logician”. (Der größte lebende Logiker war damals eindeutig Kurt Gödel.) Tarski war ein Workaholic und Frauenheld. Zum Zeitpunkt des deutschen Überfalls auf Polen 1939 war er auf einem Kongress an der Harvard-Universität. Seine in Warschau zurückgelassene Familie sah er erst 1946 wieder; ein großer Teil seiner weiteren Verwandtschaft überlebte nicht.

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

Definition 2.18 *Ein σ-Satz ist eine σ-Formel φ mit* $\mathrm{Frei}(\varphi) = \emptyset$. *Wir sagen, dass ein σ-Satz φ in einer σ-Struktur M gilt, oder auch, dass M ein* Modell von φ *ist, wenn für eine (und damit für jede) Belegung β der Variablen in M gilt: $\hat{\beta}(\varphi) = 1$. Die übliche Notation für diese Beziehung ist $M \models \varphi$.*

Übungsaufgaben

Übungsaufgabe 2.5 *Bestimmen Sie die vier Redukte von $(\mathbb{N}, 0, 1, +, \times, <)$, die isomorph zu Redukten von $(\mathbb{Q}, 0, 1, -, +, \times)$ sind.*

Übungsaufgabe 2.6 *Wir haben Homomorphismen nur für funktionale Strukturen definiert. Zeigen Sie, dass unsere Definition von Isomorphismus in diesem Fall im folgenden Sinn korrekt war: Zu jedem Isomorphismus (= bijektiven Homomorphismus) $f\colon M \to N$ gibt es (genau) einen Isomorphismus $g\colon N \to M$, der das beidseitige Inverse von f ist: $f \circ g$ und $g \circ f$ sind jeweils die Identität auf N bzw. M.*

Überlegen Sie sich, wie Sie Homomorphismen für allgemeinere (nicht funktionale) Strukturen definieren würden. Ist obiger Sachverhalt bei Ihrer Definition auch noch gegeben? (Hintergrund dazu: In der Graphentheorie gibt es den Begriff des Homomorphismus zwischen Graphen. Den kann man auf allgemeine Strukturen erweitern, aber obiges Prinzip gilt dann nicht, so dass man bei der Definition von Isomorphismen besser aufpassen muss.)

2.3 Gödelisierung

Definition 2.19 *Die (abzählbare)* Universalsignatur σ_U *ist gegeben durch* $\sigma_U^{\mathrm{Op}} = \{f_n^k \mid n,k \in \mathbb{N}\}$, $\sigma_U^{\mathrm{Rel}} = \{R_n^k \mid n,k \in \mathbb{N}\}$ *und* $\mathrm{ar}_{\sigma_U}(f_n^k) = \mathrm{ar}_{\sigma_U}(R_n^k) = k$.

Der Vorteil von σ_U: Nach etwaigem Umbenennen der Symbole können wir jede endliche (oder sogar abzählbare) Signatur σ als Teilsignatur von σ_U auffassen, d.h. $\sigma^{\mathrm{Op}} \subseteq \sigma_U^{\mathrm{Op}}$, $\sigma^{\mathrm{Rel}} \subseteq \sigma_U^{\mathrm{Rel}}$ und $\mathrm{ar} = \mathrm{ar}_U|_{\sigma^{\mathrm{Op}} \cup \sigma^{\mathrm{Rel}}}$. Da in jeder Formel nur endlich viele Symbole vorkommen, ist also jede Formel bis auf Umbenennen von Symbolen eine σ_U-Formel.[3]

Definition 2.20 *Wie legen die folgende Aufzählung* s_U *des Alphabets* $A_U = \sigma_U^{\mathrm{Op}} \cup \sigma_U^{\mathrm{Rel}} \cup \{=, \exists, \neg, \wedge\} \cup \mathbb{X}$ *als* Standardaufzählung *fest:*

$$
\begin{aligned}
s_U(0) &= = & s_U(3m+4) &= \mathtt{x}^{\mathtt{m}} \\
s_U(1) &= \exists & s_U(3m+5) &= f_{L(m)}^{R(m)} \\
s_U(2) &= \neg & s_U(3m+6) &= R_{L(m)}^{R(m)}. \\
s_U(3) &= \wedge
\end{aligned}
$$

Dabei sind L *und* R *die Komponenten der Umkehrung der Cantorschen Paarungsfunktion (siehe Satz 1.11). Dadurch ist für jedes Symbol in* A_U *eine natürliche Zahl festgelegt, die dieses eindeutig codiert.*

Jedes Wort $w = w_0 w_1 \dots w_{k-1} \in A_U^* = \bigcup_{\ell \in \mathbb{N}} A_U^\ell$ *wird durch seine* Gödelnummer

$$\langle w \rangle := \langle s_U^{-1}(w_0), s_U^{-1}(w_1), \dots, s_U^{-1}(w_{k-1}) \rangle$$

codiert. (Für die Definition von $\langle\rangle$ *siehe Definition 1.15.)*

Satz 2.21 *Die Mengen* $\{\langle t \rangle \mid t \text{ ist } \sigma_U\text{-Term}\}$ *und* $\{\langle \varphi \rangle \mid \varphi \text{ ist } \sigma_U\text{-Formel}\}$ *sind primitiv rekursiv.*

Beweis Wir zeigen das nur für die erste Menge. Für die zweite Menge geht der Beweis dann ähnlich. Wir müssen zeigen, dass die Funktion $\chi_{\{\langle t \rangle | t \text{ ist } \sigma_U\text{-Term}\}}(x)$ primitiv rekursiv ist. Dazu skizzieren wir ein LOOP-Programm, das die Funktion berechnet. Das Programm setzt die Ausgabe zunächst auf 1 (d.h. Erfolg: es handelt sich um einen Term) und geht dann von links nach rechts durch das Wort durch, um zu überprüfen, dass es sich wirklich um einen Term handelt. Falls dabei irgendwann ein Problem auftritt, setzt es die Ausgabe auf 0.(Da die Ausgabe später niemals wieder auf einen von 0 verschiedenen Wert gesetzt wird, müssen wir uns in diesem Fall nicht darum kümmern, was danach geschieht.) Am Anfang müssen wir genau einen Term lesen, also hat `termsleft` den Wert 1. Wenn wir dann z.B. ein zweistelliges Relationssymbol gelesen haben, haben wir einen der noch ausstehenden Terme zu lesen angefangen (`termsleft` verringert sich um 1), müssen dafür aber zwei weitere Terme lesen (`termsleft` erhöht sich um 2). Insgesamt erhöht sich daher `termsleft` beim Lesen eines k-stelligen Relationssymbols um $k-1$.

```
w = input
output = 1
termsleft = 1
loop Lg(w) times{
  if not(termsleft) then {
    output = 0
  }
  arity = ...
  w = ...
  termsleft = termsleft - 1 + arity
}
if termsleft then {
  output = 0
}
```

[3] Wenn man allerdings überabzählbar viele Formeln hat, kann es sein, dass man die Umbenennungen nicht kohärent durchführen kann.

Wir durchlaufen die Schleife so oft, wie das Wort lang ist. In jedem Durchlauf überprüfen wir zuerst, dass wir noch nicht am Ende des Terms angekommen sind (d.h. dass `termsleft` größer als 0 ist) und melden sonst einen Fehler. Die Zeile `arity = ...` deutet an, dass wir das erste Zeichen untersuchen und, wenn es sich um ein k-stelliges Funktionssymbol oder um eine Variable handelt (die wir als gleichwertig mit einem 1-stelligen Funktionssymbol behandeln), die Variable `arity` auf k setzen. Andernfalls (und das ist im Listing nicht zu sehen) melden wir durch `output = 0` einen Fehler. Die Zeile `w = ...` entfernt von w, aufgefasst als ein Tupel von natürlichen Zahlen, das erste Element. (Dazu muss man sich eigentlich erst noch überlegen, dass das mit einer primitiv rekursiven Funktion geht.) Nach dem letzten Durchlauf, d.h., wenn das Wort vorbei ist, überprüfen wir noch, dass wir wirklich am Ende des Terms angekommen sind. Falls nicht, melden wir einen Fehler.

Die zweite Menge kann man ganz ähnlich behandeln. Allerdings muss man dazu zuerst noch das obige Programm so umschreiben, dass es überprüft, ob ein Wort mit einem Term beginnt und ggf. den Rest des Wortes zurückgibt. ∎

Jetzt sind wir also soweit, dass wir die Syntax (Formeln, Terme, Sätze usw.) mit Computern behandeln können. Unsere Motivation ist natürlich letztlich, dass wir etwas über die Semantik (mathematische Strukturen) aussagen wollen. Dazu sind Definitionen wie die folgende hilfreich:

Definition 2.22 *Eine σ-Formel φ heißt* allgemeingültig*, falls für alle σ-Strukturen M und alle Belegungen $\beta\colon \mathbb{X} \to \underline{M}$ der Variablen in M gilt: $\hat{\beta}(\varphi) = 1$.*

Ein nichttriviales Beispiel für eine allgemeingültige σ-Formel (bei beliebigem σ) ist jede Formel der Gestalt $(x = y \wedge y = z) \to x = z$, oder genauer $\neg\wedge\wedge{=}xy{=}yz\neg{=}xz$ für beliebige Variable $x, y, z \in \mathbb{X}$. Also konkret z.B. $\neg\wedge\wedge{=}\mathtt{X}\overset{0}{\mathtt{X}}{=}\mathtt{X}\overset{1}{\mathtt{X}}\neg{=}\mathtt{X}\overset{2}{\mathtt{X}}$.

Wir können also jetzt die Frage stellen, ob die Menge $\{\langle\varphi\rangle | \varphi$ allgemeingültige σ_U-Formel$\}$ primitiv rekursiv ist. Diese Frage werden wir in Kapitel 4 negativ beantworten. Genauer werden wir sogar zeigen, dass es grundsätzlich kein Computerprogramm geben kann, das für jede natürliche Zahl entscheidet, ob sie in der Menge ist oder nicht.

Aussagenlogik

Um das Problem etwas zu vereinfachen, schauen wir uns noch kurz die Aussagenlogik an.

Definition 2.23 *Eine* aussagenlogische Signatur *ist eine Signatur σ, so dass $\sigma^{\mathrm{Op}} = \emptyset$ und $\mathrm{ar}(R) = 0$ für alle $R \in \sigma^{\mathrm{Op}}$. Ein* aussagenlogischer Satz *ist ein σ-Satz für eine aussagenlogische Signatur σ, in dem keine Variablen oder Gleichheitszeichen vorkommen. Ein allgemeingültiger aussagenlogischer Satz heißt* Tautologie*.*

Analog zu Lemma 2.17 kann man beweisen:

Bemerkung 2.24 *Sei σ eine aussagenlogische Signatur, φ ein aussagenlogischer σ-Satz und M, M' zwei σ-Strukturen, so dass $R^M = R^{M'}$ für alle $R \in \sigma^{\mathrm{Rel}}$. Dann gilt $M \models \varphi \iff M' \models \varphi$.*

Die Elemente einer Struktur spielen für die Gültigkeit aussagenlogischer Sätze also überhaupt keine Rolle.

Da σ_U aussagenlogische Teilsignaturen hat, gibt es auch aussagenlogische σ_U-Sätze.

Satz 2.25 *Die Menge $\{\langle\varphi\rangle \mid \varphi$ ist (eine σ_U-Formel und) Tautologie$\}$ ist primitiv rekursiv.*

Beweis Beweisidee: Wir können für jeden möglichen Wahrheitswert der nullstelligen Relationssymbole, die in dem Satz wirklich vorkommen, überprüfen, ob der Satz gilt. Es sind genau 2^n Fälle zu untersuchen, wenn n die Anzahl der verschiedenen Relationssymbole im Satz ist. In jedem einzelnen Fall gehen wir von rechts nach links durch den Satz durch und merken uns eine endliche Folge von Wahrheitswerten (codiert als einzelne Zahl). Jede nullstellige Relation hängt ihren Wahrheitswert hinten an. Jedes $\neg$ negiert den aktuell letzten Wert. Bei jedem werden die beiden letzten Werte entfernt und durch deren Konjunktion ersetzt. Bei einer gültigen Formel haben wir am Ende einen einzigen Wahrheitswert in der endlichen Folge: Den Wahrheitswert der Folge. ∎

Es gibt keine universelle primitiv rekursive Menge

Gödelisierung kann man auch auf LOOP-Programme statt auf Formeln anwenden. In diesem Fall wählen wir eine Aufzählung des LOOP-Alphabets

$$A = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}, \texttt{0}, \ldots, \texttt{9}, \texttt{(}, \texttt{)}, \texttt{\{}, \texttt{\}}, \texttt{,}, \texttt{=}, \_, \text{Leerzeichen}\}.$$

In derselben Weise wie bei Formeln können wir dann jedem Wort über A einen Gödelcode zuordnen.

Als Vorgeschmack auf das nächste Kapitel schauen wir uns nun die folgende Menge an:

$$PR = \{(p, x) \in \mathbb{N}^2 \mid p \text{ ist Gödelcode eines LOOP-Programms, das bei Eingabe } \texttt{input\_1} = x, \texttt{input\_2} = \texttt{input\_3} = \ldots = 0 \text{ den Wert 1 ausgibt}\}.$$

Proposition 2.26 *PR ist nicht primitiv rekursiv.*

Beweis Wenn PR primitiv rekursiv wäre, gäbe es ein LOOP-Programm, das die charakteristische Funktion von PR berechnet. Dann gäbe es auch ein LOOP-Programm, das bei der Eingabe $\texttt{input\_1} = x$, $\texttt{input\_2} = \texttt{input\_3} = \ldots = 0$ genau dann 0 ausgibt, wenn $(x, x) \in PR$ gilt. Sei p seine Gödelnummer. Nach Wahl des Programms würde dann gelten:

$$(p, p) \in PR \iff \text{Das durch } p \text{ codierte Programm gibt bei Eingabe } p \text{ den Wert 0 aus.}$$

Nach Definition von PR würde aber andererseits gelten:

$$(p, p) \in PR \iff \text{Das durch } p \text{ codierte Programm gibt bei Eingabe } p \text{ den Wert 1 aus.}$$

Der Widerspruch zeigt, dass PR nicht primitiv rekursiv ist. ∎

Kapitel 3

Rekursivität

3.1 Rekursive und GOTO-berechenbare Funktionen

Definition 3.1 *Die Menge der* rekursiven Funktionen *ist die kleinste Menge* $F \subseteq \bigcup_{k\in\mathbb{N}}\{f\colon \mathbb{N}^k \to \mathbb{N}\}$*, 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 $f\colon \mathbb{N}^\ell \to \mathbb{N}$ und $g_1, \ldots, g_\ell\colon \mathbb{N}^k \to \mathbb{N}$ alle in F liegen, dann liegt auch die durch*

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

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

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

  gegebene Funktion $h\colon \mathbb{N}^{k+1} \to \mathbb{N}$ in F.
- *F ist abgeschlossen unter* μ-Rekursion*. Das heißt, wenn die Funktion $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ 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 $g\colon \mathbb{N}^k \to \mathbb{N}$ in F.

Bemerkung 3.2 *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 3.3 *Eine Menge $A \subseteq \mathbb{N}^k$ heißt* rekursiv*, wenn ihre charakteristische Funktion*

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

rekursiv ist.

Bemerkung 3.4 *Eine Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ ist genau dann rekursiv, wenn ihr Graph $\Gamma_f \subseteq \mathbb{N}^{k+1}$ rekursiv ist.*

Beweis $\chi_=\colon \mathbb{N}^2 \to \mathbb{N}$, 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. ∎

Bemerkung 3.5 *Boolesche Kombinationen von rekursiven Mengen sind wieder rekursiv.*

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

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

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

Bemerkung 3.6 *Durch Einsetzung von rekursiven Funktionen erhält man aus einer rekursiven Menge ebenfalls wieder eine rekursive Menge. Genauer: Wenn $A \subseteq \mathbb{N}^k$ eine rekursive Menge ist und $f_1, \ldots, f_k \colon \mathbb{N}^m \to \mathbb{N}$ rekursive Funktionen sind, dann ist auch $\{\bar{a} \in \mathbb{N}^m \mid (f_1(\bar{a}), \ldots, f_k(\bar{a})) \in A\}$ eine rekursive Menge.*

Beweis Die charakteristische Funktion dieser Menge ergibt sich durch Zusammensetzung von χ_A mit den Funktionen f_i. ∎

Bemerkung 3.7 *Wenn $f, g \colon \mathbb{N}^k \to \mathbb{N}$ 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 3.8 *Wenn $A \in \mathbb{N}^{k+1}$ und $f \colon \mathbb{N}^k \to \mathbb{N}$ rekursiv sind, dann sind auch folgende Mengen rekursiv:*

$$\{\bar{x} \in \mathbb{N}^k \mid \exists y < f(\bar{x})\big((\bar{x}, y) \in A\big)\}$$
$$\{\bar{x} \in \mathbb{N}^k \mid \forall y < f(\bar{x})\big((\bar{x}, y) \in A\big)\}$$

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

Satz 3.9 *Die dreistellige Funktion $\uparrow \colon \mathbb{N}^3 \to \mathbb{N}$, $(x, y, n) \mapsto x \uparrow^n y$ 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:

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

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

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

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

$$\beta(x, y, n) = \mu b\Big(b \in B \text{ und } \exists i < \mathrm{Lg}(b) \exists z < b\big(\mathrm{Component}(b, i) = \langle x, y, n, z\rangle\big)\Big)$$
$$\gamma(x, y, n, b) = \mu i\Big(\exists z\big(\mathrm{Component}(b, i) = \langle x, y, n, z\rangle\big)\Big)$$
$$x \uparrow^n y = \mathrm{Component}\big(\mathrm{Component}(\beta(x, y, n), \gamma(x, y, n, \beta(x, y, n))), 3\big)$$

$\beta(x, y, n)$ 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. ∎

Für eine alternative Beschreibung der rekursiven Funktionen benutzen wir Programme in einer Programmiersprache, die Ähnlichkeiten mit der früher sehr verbreiteten Sprache BASIC hat. 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.

[1] Es 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.

- `y = Zero()`
- `y = Val(x)`
- `y = Inc(x)`
- `y = 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:

```
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 3.10 *Eine* partielle Funktion $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ *ist eine Funktion* $f\colon X \to \mathbb{N}$, *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 3.11 *Sei* $k \in \mathbb{N}$. *Eine partielle Funktion* $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ *heißt* GOTO-berechenbar, *falls es 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* $x_1, x_2, \ldots, x_k \in \mathbb{N}$ *stehen hat und in allen anderen Registern die Zahl 0, dann passiert Folgendes:*

- *Falls* $f(x_1, x_2, \ldots, x_k)$ *definiert ist, dann hält das Programm an, und am Ende steht im Register* `output` *die Zahl* $f(x_1, x_2, \ldots, x_k)$.
- *Falls* $f(x_1, x_2, \ldots, x_k)$ *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* $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ *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)
  }
}
```

GOTO-Orakelprogramme sind auf die offensichtliche Weise definiert.

Lemma 3.12 *Jede Funktion, die durch ein GOTO-Orakelprogramm berechenbar ist, wobei jeder Orakelname durch eine GOTO-berechenbare Funktion interpretiert wird, ist selbst GOTO-berechenbar.*

Beweis Man kann sich leicht überlegen, dass man jede Zeile, die ein Orakel benutzt, durch das (leicht abgewandelte) GOTO-Programm ersetzen kann, das diese interpretiert. Die hinteren Zeilen ändern dabei ihre Zeilennummern, so dass die GOTO-Sprünge ebenfalls angepasst werden müssen. ∎

Lemma 3.13 *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 $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ 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 goto 5
4   if y goto 1
5   output = Dec(y)
```

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

Übungsaufgaben

Übungsaufgabe 3.1 *Eine Menge $A \subseteq \mathbb{N}^k$ heißt primitiv rekursiv, wenn ihre charakteristische Funktion primitiv rekursiv ist. Sei $f\colon \mathbb{N} \to \mathbb{N}$ eine Funktion, deren Graph Γ_f primitiv rekursiv ist. Zeigen Sie, dass f rekursiv ist, aber im Allgemeinen nicht primitiv rekursiv.*

Übungsaufgabe 3.2 *Schreiben Sie ein GOTO-Programm, das die dreistellige Hyperoperatorfunktion berechnet. Können Sie das auch mit einem LOOP-Programm? Warum ist das so viel schwerer?*

Übungsaufgabe 3.3 *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 3.4 *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 diese Kontrollstruktur mit Hilfe von `loop x times` *ausdrücken?*

Übungsaufgabe 3.5 *Zu jedem GOTO-Programm gibt es ein äquivalentes WHILE-Programm. (Hinweis: Es geht mit nur einer einzigen WHILE-Schleife! Benutzen Sie Orakel und Aufgabe 3.4, und führen Sie in einer Variable Buch über die Zeile des GOTO-Programms, in der sich Ihr WHILE-Programm gerade befindet.)*

Übungsaufgabe 3.6 *Den Begriff der Rekursivität kann man auch auf partielle Funktionen ausweiten, indem man bei der μ-Rekursion auf die Bedingung verzichtet, die die Totalität sicherstellt. Welche weiteren Anpassungen muss man dann in Definition 3.1 noch machen? Zeigen Sie: Jede rekursive Funktion ist eine rekursive partielle Funktion, und jede rekursive partielle Funktion ist GOTO-berechenbar. Mit welchem Resultat aus dem nächsten Abschnitt kann man daraus folgern, dass jede rekursive partielle Funktion, die außerdem eine (totale) Funktion ist, eine rekursive Funktion ist?*

3.2 Äquivalenz von Rekursivität und GOTO-Berechenbarkeit

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: $\ulcorner \texttt{output} \urcorner = 0$, $\ulcorner \texttt{input\_1} \urcorner = 1$, $\ulcorner \texttt{input\_2} \urcorner = 2$ usw.

Bemerkung 3.14 *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:

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

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

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

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 $\mathcal{R} = \langle x_0, x_1, \ldots, x_{R-1} \rangle$ 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:

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

wobei wir zur besseren Lesbarkeit die folgenden Abkürzungen verwendet haben:

$$\ell = (\mathcal{Z})_0 \qquad \mathcal{R} = (\mathcal{Z})_1 \qquad C = (\mathcal{P})_\ell$$
$$c = (C)_0 \qquad x = (C)_1 \qquad y = (C)_2$$

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

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

gegebene Funktion $A^k \colon \mathbb{N}^k \to \mathbb{N}$ 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 3.15 *Es gibt primitiv rekursive Funktionen $N \colon \mathbb{N}^2 \to \mathbb{N}$ und $A^k \colon \mathbb{N}^k \to \mathbb{N}$ sowie für jedes GOTO-Programm eine Zahl $\mathcal{P} \in \mathbb{N}$, so dass Folgendes gilt.*

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

Kleenesche Normalform

Satz 3.16 *Es gibt eine primitiv rekursive Funktion $\mathcal{U}\colon \mathbb{N} \to \mathbb{N}$ und für jedes $k \in \mathbb{N}$ eine primitiv rekursive Funktion $\mathcal{T}_k\colon \mathbb{N}^{k+2} \to \mathbb{N}$, so dass Folgendes gilt:*

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

$$f(x_1, \ldots, x_k) = \mathcal{U}\Big(\mu y\big(\mathcal{T}_k(e, x_1, \ldots, x_k, y) > 0\big)\Big).$$

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

Lemma 3.17 *Es gibt eine primitiv rekursive Funktion $\mathcal{U}\colon \mathbb{N} \to \mathbb{N}$ und für jedes $k \in \mathbb{N}$ eine primitiv rekursive Funktion $\mathcal{T}_k\colon \mathbb{N}^{k+2} \to \mathbb{N}$, so dass Folgendes gilt:*

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

$$f(x_1, \ldots, x_k) = \mathcal{U}\Big(\mu y\big(\mathcal{T}_k(\mathcal{P}, x_1, \ldots, x_k, y) > 0\big)\Big).$$

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

Beweis Mit Hilfe der Funktionen aus Lemma 3.15 definieren wir

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

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

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

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

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

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

zu wählen. ∎

Korollar 3.18 *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.

Übungsaufgaben

Übungsaufgabe 3.7 *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 $f_U\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ die durch U berechnete partielle Funktion ist, dann ist die durch P berechnete partielle Funktion f_P gegeben durch $f_P(x_1, \ldots, x_k) = f_U(x_1, \ldots, x_k, p)$. (Hinweis: Wie Aufgabe 3.5. Benutzen Sie die beschriebene Codierung, um das ganze Programm in einer einzigen Variable halten zu können.)*

Übungsaufgabe 3.8 *Zeigen Sie mit Hilfe des Kleene-Prädikats direkt (also ohne Aufgabe 3.5 oder 3.7 zu benutzen): Jede GOTO-berechenbare partielle Funktion ist WHILE-berechenbar.*

3.3 Rekursive Aufzählbarkeit und das Halteproblem

Definition 3.19 *Eine Menge $A \subseteq \mathbb{N}^k$ heißt* rekursiv aufzählbar, *falls A Projektion einer rekursiven Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$ ist: $(x_1, \ldots, x_k) \in A$ genau dann, wenn es ein y gibt, so dass $(x_1, \ldots, x_k, y) \in \bar{A}$.*

Bemerkung 3.20 *Alle rekursiven Mengen sind rekursiv aufzählbar. Die rekursiv aufzählbaren Mengen sind abgeschlossen unter Projektion.*

Beweis Dass jede rekursive Menge rekursiv aufzählbar ist, sieht man einfach mit Hilfe von Projektionen und Zusammensetzung. Jede rekursiv aufzählbare Menge $A \subseteq \mathbb{N}^{k+1}$ hat laut Definition die Form $A = \{(\bar{x}, y) \mid \exists z : (\bar{x}, y, z) \in \bar{A}\}$, wobei $\bar{A} \subseteq \mathbb{N}^{k+1}$ rekursiv ist. Die Projektion von A hat daher die Form

$$\{\bar{x} \in \mathbb{N}^k \mid \exists y : (\bar{x}, y) \in A\} = \{\bar{x} \mid \exists y \exists z : (\bar{x}, y, z) \in \bar{A}\} = \{\bar{x} \mid \exists u : (\bar{x}, L(u), R(u)) \in \bar{A}\},$$

wobei L und R die beiden Inversen der Cantorschen Paarungsfunktion J sind. ∎

Proposition 3.21 *Die folgenden Bedingungen an eine Menge $A \subseteq \mathbb{N}^k$ sind äquivalent.*

1. *A ist Projektion einer Menge, deren charakteristische Funktion primitiv rekursiv ist (einer primitiv rekursiven Menge).*
2. *A ist rekursiv aufzählbar.*
3. *A ist Projektion einer rekursiv aufzählbaren Menge.*

Beweis $1 \Rightarrow 2 \Rightarrow 3$ ist trivial. $3 \Rightarrow 2$ heißt einfach, dass die Projektion einer rekursiv aufzählbaren Menge wieder rekursiv aufzählbar ist.

$2 \Rightarrow 1$: Sei $A \subseteq \mathbb{N}^k$ eine rekursiv aufzählbare Menge, also $A = \{\bar{x} \mid \exists y : (\bar{x}, y) \in \bar{A}\}$ für eine rekursive Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$. Nach Satz 3.16 gilt $\chi_{\bar{A}}(\bar{x}, y) = \mathcal{U}(\mu u(\mathcal{T}_{k+1}(e, \bar{x}, y, u) > 0))$, wobei $\mathcal{T}_{k+1}$ und $\mathcal{U}$ primitiv rekursive Funktionen sind und e eine Konstante ist. Es folgt $A = \{\bar{x} \mid \exists w : (\bar{x}, w) \in \bar{A}'\}$, wobei $\bar{A}' \subseteq \mathbb{N}^{k+1}$ definiert ist durch

$$\chi_{\bar{A}'}(\bar{x}, w) = \begin{cases} 1 & \text{falls } \mathcal{T}_{k+1}(e, \bar{x}, L(w), R(w)) > 0 \\ & \quad \text{und } \forall y < L(w) : \mathcal{T}_{k+1}(e, \bar{x}, y, R(w)) = 0 \\ & \quad \text{und } \mathcal{U}(L(w)) = 1, \\ 0 & \text{sonst.} \end{cases}$$

Dabei sind L und R die Umkehrfunktionen der Cantorschen Paarungsfunktion. Man sieht mit den Ergebnissen von Kapitel 1 leicht, dass $\chi_{\bar{A}'}$ primitiv rekursiv ist. ∎

Proposition 3.22 *Die folgenden Bedingungen an eine Menge $A \subseteq \mathbb{N}^k$ sind äquivalent.*

1. *A ist rekursiv aufzählbar.*
2. *$A = \emptyset$ oder A ist das Bild einer kompenentenweise primitiv rekursiven Funktion $f \colon \mathbb{N} \to \mathbb{N}^k$.*
3. *$A = \emptyset$ oder A ist das Bild einer komponentenweise rekursiven Funktion $f \colon \mathbb{N} \to \mathbb{N}^k$.*

Beweis $1 \Rightarrow 2$: Wenn $A \subseteq \mathbb{N}$ rekursiv aufzählbar ist, dann ist A von der Form $A = \{x \mid \exists y : (x, y) \in \bar{A}\}$ für eine primitiv rekursive Menge $\bar{A}$. Sei $a_0 \in A$ beliebig. Dann ist A das Bild der primitiv rekursiven Funktion

$$f(x, y) = \begin{cases} x & \text{falls } (x, y) \in \bar{A}, \\ a_0 & \text{sonst.} \end{cases}$$

Das klappt nur dann nicht, wenn es wegen $A = \emptyset$ kein solches a_0 gibt. Auf dieselbe Weise erhalten wir für $k > 1$ eine komponentenweise primitiv rekursive Funktion $f \colon \mathbb{N}^k \to \mathbb{N}^k$, die sich mit Hilfe der Umkehrungen der Cantorschen Paarungsfunktion zu einer Funktion $f' \colon \mathbb{N} \to \mathbb{N}^k$ umschreiben lässt. $2 \Rightarrow 3$ ist trivial.

$3 \Rightarrow 1$: Klar für $A = \emptyset$. Wenn $A \subseteq \mathbb{N}$ das Bild einer rekursiven Funktion f ist, dann gilt $A = \{x \in \mathbb{N} \mid \exists u : f(u) = x\}$. Durch die Bedingung $f(u) = x$ wird eine rekursive Menge definiert (vgl. Bemerkung3.4). ∎

Bemerkung 3.23 *Eine Menge ist genau dann rekursiv, wenn sie selbst und ihr Komplement rekursiv aufzählbar sind.*

Beweis Sei $A \subseteq \mathbb{N}^k$ rekursiv aufzählbar, also $A = \{\bar{x} \mid \exists y : (\bar{x}, y) \in \bar{A}\}$ für eine rekursive Menge $\bar{A} \subseteq \mathbb{N}^{k+1}$. Sei außerdem auch $B = \mathbb{N}^k \setminus A$ rekursiv aufzählbar, also $B = \{\bar{x} \mid \exists y : (\bar{x}, y) \in \bar{B}\}$ für eine rekursive Menge $\bar{B} \subseteq \mathbb{N}^{k+1}$. Dann können wir eine rekursive Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ definieren durch $f(\bar{x}) = \mu y((\bar{x}, y) \in \bar{A}$ oder $(\bar{x}, y) \in \bar{B})$.[2] Damit ist $\chi_A(\bar{x}) = \chi_{\bar{A}}(\bar{x}, f(\bar{x}))$ eine rekursive Funktion. ∎

Satz 3.24 *Es gibt eine universelle rekursiv aufzählbare Menge $W \subseteq \mathbb{N}^2$: W selbst ist rekursiv aufzählbar, und für jede rekursiv aufzählbare Menge $A \subseteq \mathbb{N}$ gibt es ein $c \in \mathbb{N}$, so dass $A = \{x \in \mathbb{N} \mid (c, x) \in W\}$.*

Beweis Die Menge

$$W = \{(c, x) \in \mathbb{N}^2 \mid \exists y(\mathcal{T}_1(c, x, y) > 0)\}$$

tut es. Weil $\mathcal{T}_1$ (primitiv) rekursiv ist, ist W 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 $A = \{x \in \mathbb{N} \mid (\mathcal{P}, x) \in W\}$. ∎

Korollar 3.25 *W ist rekursiv aufzählbar, aber nicht rekursiv.*

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

Übungsaufgaben

Übungsaufgabe 3.9 *Eine Menge $A \subseteq \mathbb{N}^k$ ist genau dann rekursiv aufzählbar, wenn sie der Definitionsbereich einer GOTO-berechenbaren partiellen Funktion ist.*

[2] Genaugenommen: $f(\bar{x}) = \mu y\big(\chi_{\bar{A}}(\bar{x}, y) + \chi_{\bar{B}}(\bar{x}, y) > 0\big)$.

Kapitel 4

Prädikatenlogik der 1. Stufe

4.1 Beweisbarkeit

Definition 4.1 *Eine σ-Formel φ heißt* erfüllbar*, falls es eine σ-Struktur M und eine Belegung β der Variablen in M gibt, so dass $\hat{\beta}(\varphi) = 1$ ist. Sie heißt* allgemeingültig *(oder logisch allgemeingültig), falls für jede σ-Struktur M und jede Belegung β in M gilt: $\hat{\beta}(\varphi) = 1$.*

Definition 4.2 (Substitution) *Die σ-Formel, die man aus einer σ-Formel φ erhält, indem man ähnlich zu Definition 2.9 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:*

- $(=t_1t_2)[\frac{t}{x}] = =(t_1[\frac{t}{x}])(t_2[\frac{t}{x}])$.
- $(Rt_1 \dots t_k)[\frac{t}{x}] = R(t_1[\frac{t}{x}]) \dots (t_k[\frac{t}{x}])$.
- $(\neg\varphi[\frac{t}{x}]) = \neg(\varphi[\frac{t}{x}])$.
- $(\wedge\varphi\psi)[\frac{t}{x}] = \wedge\varphi[\frac{t}{x}]\psi[\frac{t}{x}]$.
- $\exists y\varphi[\frac{t}{x}] = \exists y(\varphi[\frac{t}{x}])$ *falls* $y \neq x$.
- $\exists x\varphi[\frac{t}{x}] = \exists x\varphi$ *falls* $y = x$.

Lemma 4.3 (Substitutionslemma für Formeln) *Sei σ eine Signatur, M eine σ-Struktur, $\beta\colon \mathbb{X} \to \underline{M}$ eine Belegung in M, $x \in V$ eine Variable, t ein σ-Term, dessen Variablen nicht in φ vorkommen, und φ eine σ-Formel. Dann ist $\hat{\beta}(\varphi[\frac{t}{x}]) = (\hat{\beta}[\frac{\bar{\beta}(t)}{x}](\varphi)$.*

Beweis Durch Induktion über den Aufbau von φ. ∎

Definition 4.4 *Die beweisbaren σ-Formeln sind die kleinste Menge Φ von σ-Formeln, so dass gilt:*

- (Verallgemeinerte) Tautologien: *Wenn man in einer Tautologie der Aussagenlogik (allenfalls in einer erweiterten Signatur) alle Prädikate durch σ-Formeln ersetzt, so dass dasselbe Prädikat durch dieselbe Formel ersetzt wird, dann ist die resultierende Formel in Φ.*
- Modus ponens: *Wenn φ und $\neg\wedge\varphi\neg\psi$ in Φ sind, dann auch ψ.*
- Gleichheitsaxiome: *Die folgenden Formeln sind in Φ:*
  - $=^{0\,0}_{\mathtt{X}\,\mathtt{X}}$
  - $\neg\wedge=^{0\,1}_{\mathtt{X}\,\mathtt{X}}\neg=^{1\,0}_{\mathtt{X}\,\mathtt{X}}$
  - $\neg\wedge\wedge=^{0\,1}_{\mathtt{X}\,\mathtt{X}}=^{1\,2}_{\mathtt{X}\,\mathtt{X}}\neg=^{0\,2}_{\mathtt{X}\,\mathtt{X}}$
  - $\neg\wedge^k=^{1\,k+1}_{\mathtt{X}\,\mathtt{X}}=^{2\,k+2}_{\mathtt{X}\,\mathtt{X}}\dots=^{k\,2k}_{\mathtt{X}\,\mathtt{X}}\neg=f^{1\,2\,\dots\,k}_{\mathtt{X}\,\mathtt{X}\,\dots\,\mathtt{X}}f^{k+1\,k+2\,\dots\,2k}_{\mathtt{X}\,\mathtt{X}\,\dots\,\mathtt{X}}$ *für jedes k-stellige Operationssymbol $f \in \sigma^{\mathrm{Op}}$*
  - $\neg\wedge^k=^{1\,k+1}_{\mathtt{X}\,\mathtt{X}}=^{2\,k+2}_{\mathtt{X}\,\mathtt{X}}\dots=^{k\,2k}_{\mathtt{X}\,\mathtt{X}}\wedge R^{1\,2\,\dots\,k}_{\mathtt{X}\,\mathtt{X}\,\dots\,\mathtt{X}}\neg R^{k+1\,k+2\,\dots\,2k}_{\mathtt{X}\,\mathtt{X}\,\dots\,\mathtt{X}}$ *für jedes k-stellige Relationssymbol $R \in \sigma^{\mathrm{Rel}}$.*
- 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.*
- Existenzeinführung: *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 4.5 (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 4.6 *Sei σ eine endliche Signatur, und sei $\alpha\colon \mathbb{N} \to A = \sigma^{\mathrm{Op}} \cup \sigma^{\mathrm{Rel}} \cup \{=, \exists, \neg, \wedge\} \cup \mathbb{X}$ eine Aufzählung des Alphabets A. Sei $\ulcorner\varphi\urcorner = \langle a_1, a_2, \dots, a_k\rangle \in \mathbb{N}$ für jede Zeichenkette $\varphi = a_1a_2 \dots a_k \in A^k$. Dann ist $\{\ulcorner\varphi\urcorner \mid \varphi$ beweisbare σ-Formel$\} \subseteq \mathbb{N}$ eine rekursiv aufzählbare Menge.*

Beweis ... ∎

Bemerkung 4.7 *Wenn φ und ψ beweisbar sind, dann auch $\mathtt{\wedge}\varphi\psi$.*

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

Übungsaufgaben

Übungsaufgabe 4.1 *Sei σ eine beliebige Signatur. Wir betrachten die folgenden σ-Sätze:*

- $\varphi = \mathtt{\neg\wedge\exists}\overset{0}{\mathtt{x}}\mathtt{=}\overset{00}{\mathtt{xx}}\mathtt{\neg\exists}\overset{0}{\mathtt{x}}\mathtt{=}\overset{00}{\mathtt{xx}}$.
- $\varphi = \mathtt{\neg\wedge\exists}\overset{0}{\mathtt{x}}\mathtt{=}\overset{00}{\mathtt{xx}}\mathtt{\neg\exists}\overset{1}{\mathtt{x}}\mathtt{=}\overset{11}{\mathtt{xx}}$.
- $\varphi = \mathtt{\neg\exists}\overset{0}{\mathtt{x}}\mathtt{\neg=}\overset{00}{\mathtt{xx}}$.

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 4.2 *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 $\varphi = \mathtt{\neg\exists}\overset{1}{\mathtt{x}}\mathtt{\neg=}\overset{10}{\mathtt{xx}}$ eine σ-Struktur angeben, in der die Formel $\mathtt{\neg\wedge}\varphi[\overset{1}{\underset{0}{\mathtt{x}}}]\mathtt{\neg\exists}\overset{0}{\mathtt{x}}\varphi = \mathtt{\neg\wedge\neg\exists}\overset{1}{\mathtt{x}}\mathtt{\neg=}\overset{11}{\mathtt{xx}}\mathtt{\neg\neg\exists}\overset{0}{\mathtt{x}}\mathtt{\neg\exists}\overset{1}{\mathtt{x}}\mathtt{\neg=}\overset{10}{\mathtt{xx}}$ nicht gilt.*

Übungsaufgabe 4.3 *Sei σ die Signatur mit $\sigma^{\mathrm{Op}} = \emptyset$, $\sigma^{\mathrm{Rel}} = \{\mathtt{I}\}$ und $\mathrm{ar}(\mathtt{I}) = 2$. Wir betrachten die σ-Struktur M mit*

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

und

$$\begin{aligned}\mathtt{I}^M = \{&\{1, \textit{Karlsplatz}\}, \{1, \textit{Stephansplatz}\}, \{1, \textit{Schwedenplatz}\}, \{1, \textit{Praterstern}\},\\ &\{2, \textit{Karlsplatz}\}, \{2, \textit{Volkstheater}\}, \{2, \textit{Schottenring}\}, \{2, \textit{Praterstern}\},\\ &\{3, \textit{Landstraße}\}, \{3, \textit{Stephansplatz}\}, \{3, \textit{Volkstheater}\}, \{3, \textit{Westbahnhof}\},\\ &\{4, \textit{Längenfeldgasse}\}, \{4, \textit{Karlsplatz}\}, \{4, \textit{Landstraße}\}, \{4, \textit{Schwedenplatz}\},\\ &\{4, \textit{Schottenring}\}, \{4, \textit{Spittelau}\}, \{6, \textit{Längenfeldgasse}\}, \{6, \textit{Westbahnhof}\},\\ &\{6, \textit{Spittelau}\}\}.\end{aligned}$$

Zeigen Sie: $M \models \overset{0}{\mathtt{x}}\overset{1}{\mathtt{x}}\mathtt{\exists}\overset{2}{\mathtt{x}}\mathtt{\neg\exists}\overset{3}{\mathtt{x}}\mathtt{\exists}\overset{4}{\mathtt{x}}\mathtt{\exists}\overset{30}{\mathtt{x}}\overset{32}{\mathtt{x}}\overset{42}{\mathtt{x}}\overset{41}{\mathtt{x}}\mathtt{\wedge\wedge\wedge I}\overset{}{\mathtt{xx}}\mathtt{I}\overset{}{\mathtt{xx}}\mathtt{I}\overset{}{\mathtt{xx}}\mathtt{I}\overset{}{\mathtt{xx}}$. (Hinweis: Wenn man mit der U-Bahn vom Westbahnhof zum Praterstern fährt, muss man zweimal umsteigen.)

4.2 Vollständigkeitssatz

Definition 4.8 *Sei σ eine Signatur. Eine* σ-Theorie *ist eine Menge von σ-Sätzen, also von σ-Formeln ohne freie Variable.*

Eine σ-Theorie heißt erfüllbar*, falls sie ein Modell hat, d.h. eine σ-Struktur M, so dass für jeden Satz $\varphi \in T$ gilt: $M \models \varphi$. Wir schreiben dann $M \models T$. Sie heißt* widerspruchsfrei*, falls es keine $\varphi_1, \dots, \varphi_n \in T$ gibt, so dass $\neg\wedge^{n-1}\varphi_1\varphi_2\dots\varphi_n$ beweisbar ist.*

Es folgt sofort aus dem Korrektheitssatz, dass jede erfüllbare Theorie widerspruchsfrei ist. Der Vollständigkeitssatz besagt, dass umgekehrt jede widerspruchsfreie Theorie erfüllbar ist. Zum Beweis werden wir eine gegebene widerspruchsfreie Theorie so lange erweitern, bis sie eine Struktur genau beschreibt. Unter anderem brauchen wir dabei noch die folgende Eigenschaft.

Definition 4.9 *Eine widerspruchsfreie σ-Theorie heißt* vollständig*, wenn für jeden σ-Satz φ entweder $\varphi \in T$ oder $\neg\varphi \in T$ gilt.*[1]

Zum Beispiel können wir eine beliebige σ-Struktur nehmen und $T = \{\varphi \mid M \models \varphi\}$ setzen. Dann ist T eine vollständige σ-Theorie.

Lemma 4.10 *Jede widerspruchsfreie σ-Theorie lässt sich zu einer vollständigen σ-Theorie ergänzen.*

Beweis Sei T eine widerspruchsfreie σ-Theorie. Falls T nicht vollständig ist, gibt es einen σ-Satz φ, so dass weder $\varphi \in T$ noch $\neg\varphi \in T$. Wenn $T \cup \{\varphi\}$ nicht widerspruchsfrei ist, gibt es $\varphi_1, \dots, \varphi_n \in T$, so dass $\neg\wedge^n\varphi_1\varphi_2\dots\varphi_n\varphi$ beweisbar ist. Wenn $T \cup \{\neg\varphi\}$ nicht widerspruchsfrei ist, gibt es $\varphi'_1, \dots, \varphi'_m \in T$, so dass $\neg\wedge^m\varphi'_1\varphi'_2\dots\varphi'_m\neg\varphi$ beweisbar ist. Beide Fälle können nicht gleichzeitig auftreten, denn sonst könnte man zeigen, dass schon T nicht widerspruchsfrei ist.

Unvollständige widerspruchsfreie Theorien kann man also immer um mindestens einen Satz erweitern. Weil ein Gegenbeispiel zur Widerspruchsfreiheit immer nur endlich viele Formeln betrifft, gilt außerdem, dass die Vereinigung über eine beliebige durch Inklusion linear geordnete Menge (Kette) immer widerspruchsfrei ist. Aus dem Auswahlaxiom folgt mit Hilfe des Zornschen Lemmas, dass es maximale Ketten gibt. 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 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 4.11 *Eine σ-Theorie T heißt* Henkintheorie*, falls für jede σ-Formel φ mit genau einer freien Variablen x eine Konstante (d.h. nullstellige Operation) $c \in \sigma^{\mathrm{Op}}$ existiert, so dass $\neg\wedge\exists x\varphi\neg\varphi[\frac{c}{x}] \in T$.*

Proposition 4.12 *Wenn T eine vollständige (σ-)Henkintheorie ist, kann man wie folgt ein Modell $M \models T$ definieren. Sei $C \subseteq \sigma^{\mathrm{Op}}$ die Menge der Konstanten in σ. Sei σ 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 $f \in \sigma^{\mathrm{Op}}$ 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

[1] Diese Bedeutung des Worts „vollständig“ hat keine direkte Verbindung mit „Vollständigkeit“ im Sinne des Vollständigkeitssatzes, wohl aber im Sinne des Unvollständigkeitssatzes. In einer früheren Version des Skripts stand hier versehentlich eine nicht äquivalente Definition (mit $T \vdash$ statt $\in T$), die ebenfalls üblich aber für uns hier nicht so praktisch ist.

$=df c_1 \dots c_k \in T$. (Sei d so, dass $\neg\wedge\exists x{=}xfc_1 \dots c_k\neg{=}dfc_1 \dots c_k \in T$. Weil $\exists x{=}xfc_1 \dots c_k$ 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 $\beta(x_i) = c_i/\sim$ 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 Vollständigkeitssatzes.

Lemma 4.13 *Sei φ eine σ-Formel und $c \in \sigma^{\mathrm{Op}}$ 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 c folgt. Wir sparen uns alle weiteren Details. ∎

Lemma 4.14 *Sei φ eine σ-Formel mit einer freien Variable x und ψ ein σ-Satz. Falls $c \in \sigma^{\mathrm{Op}}$ eine Konstante ist, die in φ und ψ nicht vorkommt und der Satz $\neg\wedge\psi\neg\wedge\exists x\varphi\neg\varphi[\frac{c}{x}]$ beweisbar ist, dann ist auch der Satz $\neg\psi$ beweisbar.*

Beweis Zunächst betrachten wir die drei Formeln

$$\neg\wedge\psi\neg\wedge\exists x\varphi\neg\varphi[\tfrac{c}{x}]$$
$$\neg\wedge\neg\wedge\psi\neg\wedge\exists x\varphi\neg\varphi[\tfrac{c}{x}]\neg\neg\wedge\psi\neg\exists x\varphi$$
$$\neg\wedge\psi\neg\exists x\varphi$$

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.

$$\neg\wedge\psi\neg\wedge\exists x\varphi\neg\varphi[\tfrac{c}{x}]$$
$$\neg\wedge\neg\wedge\psi\neg\wedge\exists x\varphi\neg\varphi[\tfrac{c}{x}]\neg\neg\wedge\psi\varphi[\tfrac{c}{x}]$$
$$\neg\wedge\psi\varphi[\tfrac{c}{x}]$$

Weil ψ ein Satz ist, kommt x nicht frei in ψ vor. Wir können daher die zuletzt abgeleitete Formel auch als $(\neg\wedge\psi\varphi)[\frac{c}{x}]$ schreiben. Mit Lemma 4.13 folgt, dass auch $\neg\wedge\psi\varphi$ beweisbar ist. Mit Hilfe einer Tautologie und modus ponens können wir folgern, dass auch $\neg\wedge\varphi\neg\neg\psi$ beweisbar ist. Mittels Existenzeinführung folgt, dass $\neg\wedge\exists x\varphi\neg\neg\psi$ beweisbar ist. Wiederum mittels einer Tautologie und modus ponens folgt die Beweisbarkeit von $\neg\wedge\psi\exists x\varphi$.

Mit Hilfe von Bemerkung 4.7 können wir die zuerst bewiesene Formel $\neg\wedge\psi\neg\exists x\varphi$ und die gerade eben bewiesene Formel $\neg\wedge\psi\exists x\varphi$ zur ebenfalls beweisbaren Formel $\chi = \wedge\neg\wedge\psi\neg\exists x\varphi\neg\wedge\psi\exists x\varphi$ kombinieren. Zusammen mit der Tautologie $\neg\wedge\chi\neg\neg\psi$ liefert uns der modus ponens schließlich $\neg\psi$, wie gewünscht. ∎

Damit ist der Beweis unseres Hauptergebnisses relativ einfach geworden.

Lemma 4.15 *Eine Theorie ist genau dann widerspruchsfrei, wenn sie erfüllbar ist.*

Beweis Es folgt aus dem Korrektheitssatz, 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 4.14 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 4.10 vollständig. Sie bleibt dabei eine Henkintheorie, und als vollständige Henkintheorie hat sie ein Modell. Dieses ist auch ein Modell der ursprünglichen Theorie. ∎

Definition 4.16 *Sei T eine σ-Theorie und ψ ein σ-Satz. Wir sagen, T* beweist *ψ, und schreiben $T \vdash \psi$, falls es $\varphi_1, \ldots, \varphi_n \in T$ gibt, so dass $\neg\wedge^n\varphi_1 \ldots \varphi_n\neg\psi$ 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 4.17 (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 4.18 (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 4.19 (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. ∎

Übungsaufgaben

Übungsaufgabe 4.4 *Sei σ_{Ab} die Signatur mit $\sigma_{\mathrm{Ab}}^{\mathrm{Op}} = \{\mathtt{+}, \mathtt{-}, \mathtt{0}\}$, $\sigma_{\mathrm{Ab}}^{\mathrm{Rel}} = \emptyset$, $\mathrm{ar}_{\mathrm{Ab}}(\mathtt{+}) = 2$, $\mathrm{ar}_{\mathrm{Ab}}(\mathtt{-}) = 1$ und $\mathrm{ar}_{\mathrm{Ab}}(\mathtt{0}) = 0$.*

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 0^M) genau dann, wenn $M \models T$.

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

Übungsaufgabe 4.5 *Sei σ die Signatur mit einem einzigen einstelligen Operationssymbol $\mathtt{f}$. Was kann man in jedem der folgenden Fälle über die Anzahl der Elemente eines beliebigen Modells $M \models T$ sagen?*

- $T = \{\overset{0}{\exists}\mathtt{x}\overset{1}{\exists}\mathtt{x}\overset{01}{\neg=}\mathtt{xx}\}$.
- $T = \{\overset{0}{\exists}\mathtt{x}\overset{1}{\exists}\mathtt{x}\overset{2}{\exists}\mathtt{x}\wedge\wedge\overset{01}{\neg=}\mathtt{xx}\overset{02}{\neg=}\mathtt{xx}\overset{12}{\neg=}\mathtt{xx}\}$.
- $T = \{\overset{0}{\neg}\overset{1}{\exists}\mathtt{x}\overset{2}{\exists}\mathtt{x}\overset{01}{\exists}\mathtt{x}\wedge\wedge\overset{01}{\neg=}\mathtt{xx}\overset{02}{\neg=}\mathtt{xx}\overset{12}{\neg=}\mathtt{xx}, \overset{0}{\exists}\mathtt{x}\overset{1}{\exists}\mathtt{x}\overset{01}{\neg=}\mathtt{xx}\}$.
- $T = \{\overset{0}{\neg}\overset{0}{\exists}\mathtt{x}\overset{0}{\neg=}\mathtt{xf}\overset{0}{\mathtt{f}}\mathtt{x}, \overset{0}{\neg}\overset{0}{\exists}\mathtt{x}\overset{0}{=}\mathtt{xf}\overset{0}{\mathtt{x}}\}$.
- $T = \{\overset{0}{\neg}\overset{1}{\exists}\mathtt{x}\overset{01}{\exists}\mathtt{x}\wedge\overset{01}{=}\mathtt{fx}\mathtt{f}\overset{01}{\mathtt{x}}\overset{1}{\neg=}\mathtt{xx}, \overset{0}{\exists}\mathtt{x}\overset{1}{\neg}\overset{0}{\exists}\mathtt{x}\overset{01}{=}\mathtt{xf}\overset{0}{\mathtt{x}}\}$. *(Hinweis: Eine Abbildung zwischen zwei endlichen Mengen ist injektiv genau dann, wenn sie surjektiv ist.)*

Übungsaufgabe 4.6 *Sei σ die Signatur mit einem einzigen zweistelligen Relationssymbol $\mathtt{<}$. Wir betrachten die Menge*

$$T = \{\overset{0}{\neg}\overset{1}{\exists}\mathtt{x}\overset{2}{\exists}\mathtt{x}\overset{01}{\exists}\mathtt{x}\overset{12}{<}\mathtt{xx}\overset{02}{<}\mathtt{xx}\overset{01}{\neg}\overset{01}{<}\mathtt{xx}, \overset{0}{\neg}\overset{00}{\exists}\mathtt{x}\overset{0}{<}\mathtt{xx}, \overset{0}{\neg}\overset{01}{\exists}\mathtt{xx}\wedge\overset{01}{<}\mathtt{xx}\overset{10}{<}\mathtt{xx},$$
$$\neg\overset{0}{\exists}\mathtt{x}\overset{1}{\exists}\mathtt{x}\wedge\wedge\overset{01}{\neg}\overset{01}{<}\mathtt{xx}\overset{01}{\neg=}\mathtt{xx}\overset{10}{\neg}\overset{10}{<}\mathtt{xx}, \overset{0}{\neg}\overset{2}{\exists}\mathtt{x}\overset{02}{\exists}\mathtt{x}\wedge\overset{02}{<}\mathtt{xx}\overset{1}{\neg}\overset{01}{\exists}\mathtt{x}\wedge\overset{01}{<}\mathtt{xx}\overset{12}{<}\mathtt{xx},$$
$$\neg\overset{0}{\exists}\mathtt{x}\overset{1}{\neg}\overset{01}{\exists}\mathtt{x}\overset{01}{<}\mathtt{xx}, \overset{0}{\neg}\overset{1}{\exists}\mathtt{x}\overset{10}{<}\mathtt{xx}\}$$

Eine der Zeichenketten in T ist kein σ-Satz. Korrigieren Sie sie sinnvoll, damit T eine Theorie wird. Der Isomorphiesatz von Cantor besagt, dass je zwei abzählbare dichte lineare Ordnungen ohne Endpunkte isomorph sind. Folgern Sie: Jedes Modell $M \models T$, bei dem $\underline{M}$ abzählbar ist, ist isomorph zu $(\mathbb{Q}, <)$. Geben Sie andererseits unendlich viele paarweise nicht isomorphe Modelle von T an, die all genauso viele Elemente haben wie $\mathbb{R}$. (Hinweis: Kleben Sie Kopien von $\mathbb{R}$ und $\mathbb{Q}$ zusammen.)

Übungsaufgabe 4.7 *Sei $M \models T$ ein abzählbares Modell der Theorie T aus der vorigen Aufgabe, und sei φ ein σ-Satz. Zeigen Sie: Wenn $M \models \varphi$ gilt, dann auch $(\mathbb{Q}, <) \models \varphi$. Zeigen Sie, dass das sogar dann gilt, wenn M nicht abzählbar ist. Zeigen Sie weiter, dass es genau eine vollständige σ-Theorie T' gibt mit $T' \supseteq T$.*

Übungsaufgabe 4.8 *Sei σ_{Kp} die Signatur mit $\sigma_{\mathrm{Kp}}^{\mathrm{Op}} = \{\mathtt{\times}, \mathtt{+}, \mathtt{-}, \mathtt{1}, \mathtt{0}\}$, $\sigma_{\mathrm{Kp}}^{\mathrm{Rel}} = \emptyset$, $\mathrm{ar}_{\mathrm{Kp}}(\mathtt{\times}) = \mathrm{ar}_{\mathrm{Kp}}(\mathtt{+}) = 2$, $\mathrm{ar}_{\mathrm{Kp}}(\mathtt{-}) = 1$ und $\mathrm{ar}_{\mathrm{Kp}}(\mathtt{1}) = \mathrm{ar}_{\mathrm{Kp}}(\mathtt{0}) = 0$. Geben Sie eine σ_{Kp}-Theorie T_{Kp} an, so dass die Modelle von T_{Kp} gerade die Körper sind.*

Übungsaufgabe 4.9 *Ein Körper hat Charakteristik 7, wenn für alle x gilt: $x+x+x+x+x+x+x = 0$. Wie man leicht sieht, ist das äquivalent zu $1+1+1+1+1+1+1 = 0$. Entsprechend für andere Charakteristiken $p \in \mathbb{N} \setminus \{0\}$. Aus der Algebra ist bekannt, dass die Charakteristik eines Körpers eine Primzahl sein muss. Wenn es keine gibt, sagt man, der Körper hat die Charakteristik 0. Geben Sie eine σ_{Kp}-Theorie T an, so dass die Modelle von T genau die Körper der Charakteristik 0 sind.*

Übungsaufgabe 4.10 (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 soviele Konstanten, wie A Elemente hat.)*

4.3 Unvollständigkeitssatz

Am Anfang des 19. Jahrhunderts wurden die mathematische Öffentlichkeit durch die *Russellsche Antinomie* erschüttert. Die Mengenlehre in ihrer ursprünglichen, „naiven" Form stellte sich als widersprüchlich heraus, denn sie ließ es zu, die Menge $\{x \mid x \notin x\}$ zu bilden, die es aber offensichtlich nicht geben kann. Dieser schwere Schlag kam zu einer Zeit, als die Mengenlehre trotz einiger Anfeindungen von konservativer Seite gerade dabei war, zur Grundlage der reinen Mathematik zu werden. Man fand zwar schnell Möglichkeiten, diesen Widerspruch *anscheinend* zu eliminieren, aber es war offen, ob nicht noch andere Widersprüche übrigblieben.

Bald darauf begannen die britischen Mathematiker und Philosophen Bertrand Russell und Alfred North Whitehead, dieses Problem mit ihrem monumentalen Projekt *Principia Mathematica* anzugehen. Ziel war es, ein System von Regeln anzugeben, mit dem alle wahren Sätze der Arithmetik (und nur diese) sich rein mechanisch ableiten lassen. Die von Russell und Whitehead benutzte Logik war im Wesentlichen die Prädikatenlogik 1. Stufe. Es gab eine Theorie (nennen wir sie T), die sich aus Sätzen über die Mengenlehre zusammensetzte. In diesem System sind also alle Sätze ableitbar, die aus T im Sinne von Definition 4.16 beweisbar sind. Die Anforderungen an T können wir so zusammenfassen:

- 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 4.20 *Sei σ_N die Signatur mit $\sigma_N^{\mathrm{Op}} = \{\mathtt{0}, \mathtt{1}, \mathtt{+}, \mathtt{\cdot}\}$, $\sigma_N^{\mathrm{Rel}} = \{\mathtt{<}\}$ 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 $\mathtt{+}^n\mathtt{01}^n$. Für jede σ_N-Formel φ mit freien Variablen $\overset{1}{\mathtt{x}}, \ldots, \overset{k}{\mathtt{x}}$ setzen wir $\varphi(\mathbb{N}^k) = \{(n_1, \ldots, n_k) \in \mathbb{N}^k \mid \mathbb{N} \models \varphi[\underline{n_1}/\overset{1}{\mathtt{x}}] \ldots [\underline{n_k}/\overset{k}{\mathtt{x}}]\}$.*

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.*

Lemma 4.21 *Es gibt eine arithmetische Funktion $\beta\colon \mathbb{N}^3 \to \mathbb{N}$ mit der folgenden Eigenschaft. Für jedes k-Tupel $(x_0, \ldots, x_{k-1})$ existieren $a, b \in \mathbb{N}$, so dass für alle $i < k$ gilt: $\beta(a, b, i) = x_i$.*

Beweis 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 (Gödel β-Funktion). Mit Hilfe des chinesischen Restsatzes kann man zeigen, dass β die geforderte Eigenschaft hat. Siehe Übungsaufgabe 4.11.

Alternativ kann man statt a, b auch p, t, n (n heißt bei uns k) aus Aufgabe 1 von Übungsblatt 2 benutzen. Dann ist β eine arithmetische Funktion von $\mathbb{N}^4$ nach $\mathbb{N}$, aber das ergibt in der Anwendung keinen Unterschied. ∎

Proposition 4.22 *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 offensichtlich arithmetischen β-Funktion aus Übungsaufgabe 4.11 (oder mit der ebenfalls arithmetischen Codierung von Tupeln durch t, p und n aus Aufgabe 1 von Übungsblatt 2) auf die Prädikatenlogik 1. Stufe zurückführen:

Seien f und g arithmetisch, und sei $h(\bar{x}, 0) = f(\bar{x})$ und $h(\bar{x}, y + 1) = g(\bar{x}, y, h(\bar{x}, y))$.

$$h(\bar{x}, y) = z \iff \exists a, b\colon \Big(\beta(a, b, 0) = f(\bar{x}) \text{ und } \forall i < y\colon \beta(a, b, i + 1) = g(\bar{x}, y, \beta(a, b, i))\Big)$$

∎

Korollar 4.23 *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 $\psi = \exists^{k+1} \mathsf{x}\, \varphi$ ist. ∎

Wir verwenden ab jetzt wieder wie in Satz 4.6 die Codierung $\ulcorner\varphi\urcorner \in \mathbb{N}$ von σ_N-Formeln. Damit können wir eine Theorie T rekursiv, rekursiv aufzählbar oder arithmetisch nennen, je nachdem, ob $\{\ulcorner\varphi\urcorner \mid \varphi \in T\}$ es ist.

Korollar 4.24 *Es gibt kein Computerprogramm, das für jeden σ_N-Satz φ entscheiden kann, ob $\mathbb{N} \models \varphi$ gilt oder nicht.*

Beweis Die Funktion, die jedem Code $\ulcorner\varphi\urcorner$ einer σ_N-Formel mit einer freien Variable x und jeder natürlichen Zahl $n \in \mathbb{N}$ den Code des σ_N-Satzes $\varphi[\underline{n}/x]$ zuordnet, ist rekursiv, wie man sich leicht überlegt. Wenn $T = \{\ulcorner\varphi\urcorner \mid \mathbb{N} \models \varphi\}$ rekursiv wäre, dann wäre also 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, d.h. alle arithmetischen Teilmengen von $\mathbb{N}$ wären rekursiv. Da alle rekursiv aufzählbaren Mengen arithmetisch sind, würde folgen, dass alle rekursiv aufzählbaren Teilmengen von $\mathbb{N}$ rekursiv sind. Wir wissen aber aus Korollar 3.25, dass das nicht stimmt: Die universelle rekursiv aufzählbare Menge W ist rekursiv aufzählbar, also arithmetisch, aber sie ist nicht rekursiv. ∎

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

Satz 4.25 (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.*

Übungsaufgaben

Übungsaufgabe 4.11 (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 $(c_1, \ldots, c_k) \in \mathbb{N}^*$ ein Paar $(a, b) \in \mathbb{N}^2$ gibt, so dass $\beta(a, b, i) = c_i$ ist für $i = 1, \ldots, n$. 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.)*

Übungsaufgabe 4.12 *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*

$$U = \{(e, n) \in \mathbb{N}^2 \mid e = \ulcorner\varphi\urcorner \text{ und } \mathbb{N} \models \varphi[n/\overset{0}{\mathsf{x}}]\},$$

wobei φ für Formeln steht, die nur $\overset{0}{\mathsf{x}}$ als freie Variablen haben.)

Übungsaufgabe 4.13 *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 rellen Zahlen gilt das übrigens nicht. Warum funktioniert das Argument in diesem Fall nicht?*

Übungsaufgabe 4.14 *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: $\ulcorner\varphi\urcorner \in A \iff \exists n\colon (\ulcorner\varphi\urcorner, n) \in \bar{A}$ in der Notation von Definition 3.19. Betrachten Sie die Theorie $T' = \{\mathsf{A}^{n+1}\varphi^{n+1} \mid (\ulcorner\varphi\urcorner, n) \in \bar{A}\}$.)*

Übungsaufgabe 4.15 *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.*