Theoretische Informatik 3. Semester

Prof. Dr. Martin Plümicke

29. Februar 2024

Inhaltsverzeichnis

1 Berechenbarkeit und rekursive Funktionen

Wir werden uns nun mit den Möglichkeiten eines Computers beschäftigen. Dabei werden wir sehen, dass alle Computer grundsätzlich die gleichen Fähigkeiten haben. Sie unterscheiden sich nur in der Geschwindigkeit und bei den möglichen Ein- und Ausgabearten. Berechnungen sind als partielle Funktionen $f : \{0,1\}^n \dashrightarrow \{0,1\}^m$, darstellbar. Wenn man die Zahlen vom 2er–System ins 10er–System umrechnet, so erhält man eine Funktion

$$f : \mathbb{N}^{n'} \dashrightarrow \mathbb{N}^{m'}.$$

Beispiel. Man kann eine Funktion

$$add : \{0,1\}^{16} \to \{0,1\}^8$$

betrachten als

$$add : \{0,1\}^8 \times \{0,1\}^8 \to \{0,1\}^8.$$

Wenn man der Berechnung jeweils 8-Bit Zahlen zu Grunde legt, so kann man die Funktion als

$$add : \mathbb{N}^2 \to \mathbb{N}$$

ansehen.

Wir werden nun Schritt für Schritt die Menge aller von Computern berechenbaren Funktionen herleiten.

1.1 Primitive rekursive Funktionen

Definition 1.1 (Grundfunktionen)**.** Die Menge der **Grundfunktionen** enthält folgende totale Funktionen.

1. $C_0^0 :\to \mathbb{N}$ mit $C_0^0 = 0$ **(Nullstellige Nullfunktion)**
2. $C_0^1 : \mathbb{N} \to \mathbb{N}$ mit $C_0^1(n) = 0$ **(einstellige Nullfunktion)**
3. $\Pi_i^k : \mathbb{N}^k \to \mathbb{N}$ mit $\Pi_i^k(n_1, \ldots, n_k) = n_i$ $(i \leq k)$ **(Projektionsfunktionen)**
4. $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = n + 1$ **(Nachfolger)**

Beispiel. Folgende Beispiele veranschaulichen die abstrakten Definitionen

- $\Pi_1^2(7, 22) = 7$
- $\Pi_3^5(1, 2, 4, 3, 5) = 4$
- $s(\,3\,) = 4$

Definition 1.2 (Komposition)**.** Seien $r \in \mathbb{N}_0$ und $s \in \mathbb{N} \setminus \{0\}$ und seien $g : \mathbb{N}^s \dashrightarrow \mathbb{N}$, sowie $h_i : \mathbb{N}^r \dashrightarrow \mathbb{N}$ für $1 \leq i \leq s$ partielle Funktionen, so heißt $f : \mathbb{N}^r \dashrightarrow \mathbb{N}$ mit

$$f(n_1, \ldots, n_r) = \begin{cases} g(h_1(n_1, \ldots, n_r), \ldots, h_s(n_1, \ldots, n_r)) & \text{falls definiert} \\ \perp \text{ (undefiniert)} & \text{sonst} \end{cases}$$

die **Komposition** von g und $h_1, \ldots, h_s$.

Beispiel. Seien folgende Beispiele gegeben:

1. r = 1, s = 2
$$\left.\begin{array}{l} g(n_1, n_2) = \Pi_1^2(n_1, n_2) \\ h_1(n) = s(n) \\ h_2(n) = C_0^1(n) \end{array}\right\} f(n) = g(h_1(n), h_2(n)) = \Pi_1^2(s(n), C_0^1(n)) = s(n)$$
2. r = 1, s = 1
$$\left.\begin{array}{l} g(n) = s(n) \\ h_1(n) = C_0^1(n) \end{array}\right\} f(n) = g(h_1(n)) = s(C_0^1(n)) = s(0) = 1$$

Definition 1.3 (Primitives Rekursionsschema)**.** Seien $k \in \mathbb{N}_0$, $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ eine totale Funktion, dann ist durch

1. $f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$
2. $f(x_1, \ldots, x_k, s(y)) = h(x_1, \ldots, x_k, y, f(x_1, \ldots, x_k, y))$

das zu g und h gehörige primitive Rekursionsschema gegeben. Man sagt f erfüllt das primitive Rekursionsschema, wenn f den Gleichungen 1 und 2 genügt. Das heißt, wir haben ein Gleichungssystem mit einer unbekannten Funktion f.

Herleitung der Funktion, die dem primitiven Rekursionsschema genügt:

Wir leiten nun die Funktion Schritt für Schritt her. Beginnend an der Stelle 0 wird in jedem Schritt eine Funktion bestimmt, die für eine weitere Zahl der gesuchten Funktion entspricht.

An der Stelle 0 ist $f(\, x_1, \ldots, x_k, 0\,)$ durch $g(\, x_1, \ldots, x_k\,)$ bestimmt. Wir nennen die Funktion, die an der Stelle 0 der Funktion f entspricht und allen anderen Stellen undefiniert ist f_0:

$$f_0(x_1, \ldots, x_k, y) = \begin{cases} g(x_1, \ldots, x_k) \text{ für } y = 0 \\ \perp \text{ sonst} \end{cases}$$

Da die Funktion f an der Stelle 1 durch die Funktion h und durch f an der Stelle 0 definiert ist, lässt sich die Funktion f_1, die an den Stellen 0 und 1 der Funktion f entspricht und sonst undefiniert ist, wie folgt darstellen:

$$f_1(x_1, \ldots, x_k, y) = \begin{cases} f_0(x_1, \ldots, x_k, y) \text{ für } y < 1 \\ h(x_1, \ldots, x_k, y-1, f_0(x_1, \ldots, x_k, y-1)) \text{ für } y = 1 \\ \perp \text{ sonst} \end{cases}$$

Analog folgt f_2:

$$f_2(x_1, \ldots, x_k, y) = \begin{cases} f_1(x_1, \ldots, x_k, y) \text{ für } y < 2 \\ h(x_1, \ldots, x_k, y-1, f_1(x_1, \ldots, x_k, y-1)) \text{ für } y = 2 \\ \perp \text{ sonst} \end{cases}$$

Sei weiterhin:

$$f_n(x_1, \ldots, x_k, y) = \begin{cases} f_{n-1}(x_1, \ldots, x_k, y) \text{ für } y < n \\ h(x_1, \ldots, x_k, y-1, f_{n-1}(x_1, \ldots, x_k, y-1)) \text{ für } y = n \\ \perp \text{ sonst} \end{cases}$$

Die Funktion f_n erfüllt das primitive Rekursionsschema für alle $y \leq n$

Die Lösung für das primitive Rekursionsschema ist dann:

$$f = \lim_{n \to \infty} f_n$$

Beispiel. Sei

$$g : \mathbb{N} \to \mathbb{N} \text{ mit } g(x) = x = \Pi_1^1(x)$$

und $h : \mathbb{N}^3 \to \mathbb{N}$ mit

$$h(x_1, x_2, x_3) = x_3 + 1 = s(\, \Pi_3^3(x_1, x_2, x_3)\,).$$

Daraus ergibt sich folgendes primitives Rekursionsschema:

$f(x,0) = g(x) = x$
$f(x,s(y)) = h(x,y,f(x,y)) = f(x,y) + 1$

Herleitung der Lösung:

$$f_0(x,y) = \begin{cases} x, & y = 0 \\ \bot, & \text{sonst} \end{cases}$$

$$f_1(x,y) = \begin{cases} f_0(x,0) = x, & y = 0 \\ f_0(x,0) + 1 = x + 1, & y = 1 \\ \bot, & \text{sonst} \end{cases}$$

$$f_2(x,y) = \begin{cases} f_1(x,0) = x, & y = 0 \\ f_1(x,0) + 1 = x + 1, & y = 1 \\ f_1(x,1) + 1 = x + 2, & y = 2 \\ \bot, & \text{sonst} \end{cases}$$

$$f_3(x,y) = \begin{cases} f_2(x,0) = x, & y = 0 \\ f_2(x,0) + 1 = x + 1, & y = 1 \\ f_2(x,1) + 1 = x + 2, & y = 2 \\ f_2(x,2) + 1 = x + 3, & y = 3 \\ \bot, & \text{sonst} \end{cases}$$

Als Lösung ergibt sich dann:

$$f(x,y) = \lim_{n \to \infty} f_n(x,y) = x + y$$

Das primitive Rekursionsschema kann man direkt in Programmiersprachen übernehmen. Die Additionsfunktion lässt sich beispielsweise in Haskell implementieren:

```
add:: (Int, Int) -> Int

add(x, 0) = x
add(x, n) = add(x, n - 1) + 1
```

Auch in Java lässt sich die Additionsfunktion rekursiv implementieren:

```
class Add {
   int add(int x, int y) {
       if(y == 0) return x;
       else if (y > 0) return (add(x,y-1)+1);
   }
```

}

Satz 1.4. Seien $k \in \mathbb{N}\setminus\{0\}, g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ als totale Funktionen gegeben, so existiert genau eine totale Funktion: $f : \mathbb{N}^{k+1} \to \mathbb{N}$, die das primitive Rekursionsschema erfüllt.

Definition 1.5 (Primitiv rekursive Funktionen)**.** Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **primitiv rekursiv**, wenn sie entweder eine Grundfunktion ist, oder aus diesen in endlich vielen Schritten durch Komposition und durch das primitive Rekursionsschema gebildet wurde.

Satz 1.6. Jede primitive rekursive Funktion ist total.

Satz 1.7. Folgende Funktionen sind primitiv rekursiv:

1. $f(x, y) = x + y$
2. $f_k(x) = k \cdot x$, für alle $k \in \mathbb{N}$
3. $f(x, y) = x \cdot y$
4. $f(x, y) = x^y$
5. $f(x, y) = x^{x^{x^{\cdots}}}$ (y - mal)
6. $f(x, y) = x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & \text{sonst} \end{cases}$
7. $f(x, y) = |x - y|$

1.2 LOOP–Programme

Nun geben wir eine kleine Programmiersprache an, mit der man genau die primitiv rekursiven Funktionen programmieren kann.

Definition 1.8 (LOOP–Programme)**.** Die Programmiersprache LOOP umfasst:

- Variablen:
  $x_0, x_1, \ldots; y_1, y_2, \ldots; z_1, z_2, \ldots$
- Konstanten;
  $0, 1, 2, 3, 4, 5, \ldots$

- Trennsymbole:
  ";" und "="
- Operationszeichen:
  "+" und "-"
- Schlüsselwörter:
  `LOOP`, `DO`, `END`

LOOP–Programme werden wie folgt definiert:

1. $x_i = x_j + c$ ist ein LOOP–Programm.
2. $x_i = x_j - c$ ist ein LOOP–Programm.
3. Seien P_1 und P_2 LOOP–Programme, dann ist auch

   $$P_1; P_2$$

   ein LOOP–Programm.
4. Falls P ein LOOP–Programm ist, so ist auch

   $$\texttt{LOOP}\ x_i\ \texttt{DO}\ P\ \texttt{END}$$

   ein LOOP–Programm.

- Die Bedeutung (Semantik) eines LOOP–Programms P ist dadurch gegeben, dass P eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ berechnen soll. Zunächst gilt für den Aufruf $f(n_1, \ldots, n_k)$, dass die Variablen $x_1 = n_1, \ldots, x_k = n_k$ enthalten. $x_i = 0$ für $i > k$
- Die Wertzuweisungen haben die Bedeutung wie in Java / C.
  Ausnahme: Für $x_i = x_j - c$ und $x_j < c$ gilt $x_i = 0$
- $P_1; P_2$ heißt: Führe zunächst P_1 und dann P_2 aus.
- Die `LOOP`–Schleife hat folgende Bedeutung: `LOOP` x_i `DO` P `END` heißt, P wird sooft ausgeführt wie in x_i zu Beginn der LOOP–Schleife steht. Änderungen von x_i während der Schleife haben keine Auswirkung.
- Das Ergebnis steht am Ende in x_0

Anmerkung. Es ist offensichtlich, dass keine Endlosschleifen mit LOOP programmiert werden können.

Lemma 1.9. Die Grundfunktionen (Def. 1.1) lassen sich durch LOOP–Programme programmieren.

Beweis.

1. $C_0^0 :\to \mathbb{N}$ mit $C_0^0 = 0$
   LOOP–Programm: $x_0 = 0$
2. $C_0^1 : \mathbb{N} \to \mathbb{N}$ mit $C_0^1(x) = 0$
   LOOP–Programm: $x_0 = 0$
3. $\Pi_i^k : \mathbb{N}^k \to \mathbb{N}$ mit $\Pi_i^k(n_1, \ldots, n_k) = n_i$
   LOOP–Programm: $x_0 = x_i$
4. $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = n + 1$
   LOOP–Programm: $x_0 = x_1 + 1$

□

Der nächste Schritt wäre die Überlegung, ob man die Komposition von gegebenen Funktionen ebenfalls durch LOOP–Programme programmieren kann.

Lemma 1.10. Seien $h_i : \mathbb{N}^k \to \mathbb{N}$ für $1 \leq i \leq s$ primitiv rekursive Funktionen, die durch LOOP–Programme $P_1, \ldots, P_s$ programmiert sind. Sei weiter: $g : \mathbb{N}^s \to \mathbb{N}$ eine primitiv rekursive Funktion, die durch das LOOP–Programm P_g programmiert ist. Dann gibt es ein LOOP–Programm P_f welches $f : \mathbb{N}^k \to \mathbb{N}$ mit $f(x_1, \ldots, x_k) = g(h(x_1, \ldots, x_k), \ldots, h_s(x_1, \ldots, x_k))$ programmiert.

Beweis. Es wird jetzt das LOOP–Programm P_f angegeben:

$z_1 = x_1; z_2 = x_2; \ldots; z_k = x_k$;//Eingaben werden zwischengespeichert
P_1;
$y_1 = x_0$; //Ergebnis von P_1 wird zwischengespeichert
$x_1 = z_1; \ldots; x_k = z_k$;
P_2;
$y_2 = x_0$; //Ergebnis von P_2 wird zwischengespeichert
$x_1 = z_1; \ldots; x_k = z_k$;
P_3;
$y_3 = x_0$; //Ergebnis von P_3 wird zwischengespeichert
...
$x_1 = z_1; \ldots; x_k = z_k$;
P_s;

$y_s = x_0$;
$x_1 = y_1; x_2 = y_2; \ldots; x_s = y_s$;
P_g □

Beispiel. (vgl. Beispiel 1.1) Seien $g : \mathbb{N} \to \mathbb{N}$ mit $g(x) = s(x)$ und $h_1 : \mathbb{N} \to \mathbb{N}$ mit $h_1(x) = C_0^1(x)$ gegeben. Seien weiterhin die zugehörigen LOOP–Programme gegeben durch:

$$P_g \stackrel{\text{def}}{=} x_0 = x_1 + 1 \qquad \text{und} \qquad P_1 \stackrel{\text{def}}{=} x_0 = 0.$$

Dann ergibt sich für P_f:

$z_1 = x_1$;
$x_0 = 0$; //Programm P_1
$y_1 = x_0$;
$x_1 = y_1$;
$x_0 = x_1 + 1$ //Programm P_g

Lemma 1.11. Seien für $k \geq 0$, $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ primitiv rekursive Funktionen die durch LOOP–Programme P_g und P_h programmiert sind. Dann gibt es für die primitiv rekursive Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$, die das primitive Rekursionsschema

1. $f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$
2. $f(x_1, \ldots, x_k, s(y)) = h(x_1, \ldots . x_k, y, f(x_1, \ldots, x_k, y))$

erfüllt, ein LOOP–Programm P_f, das f programmiert.

Beweis. Es wird jetzt das LOOP–Programm P_f angegeben:

$z_1 = x_1, z_2 = x_2, \ldots, z_k = x_k, z_{k+1} = x_{k+1}$; //Eingaben werden zwischengespeichert
P_g;
$y = 1$;
`LOOP` z_{k+1} `DO`
    $x_1 = z_1; \ldots; x_k = z_k$; //Entfällt bei $k = 0$
    $x_{k+1} = y - 1$;
    $x_{k+2} = x_0$;
    P_h;
    $y = y + 1$
`END` □

Beispiel. (vgl. Beispiel 1.1) Seien für $k = 1$ die Funktionen $g : \mathbb{N} \to \mathbb{N}$ mit $g(x) = x$ und $h : \mathbb{N}^3 \to \mathbb{N}$ mit $h(x_1, x_2, x_3) = x_3 + 1$ gegeben. Dann ergibt sich das primitive Rekursionsschema:

1. $f(x, 0) = g(x) = x$
2. $f(x, s(y)) = h(x, y, f(x, y)) = f(x, y) + 1$.

Weiterhin seien die zugehörigen LOOP–Programme

$$P_g \overset{\text{def}}{=} x_0 = x_1; \qquad \text{und} \qquad P_h \overset{\text{def}}{=} x_0 = x_3 + 1;$$

gegeben.

Daraus ergibt sich dann für P_f:

$z_1 = x_1; z_2 = x_2;$
$x_0 = x_1;$ //Programm P_g
$y = 1;$
`LOOP` z_2 `DO`
    $x_1 = z_1;$
    $x_2 = y - 1;$
    $x_3 = x_0;$
    $x_0 = x_3 + 1;$ //Programm P_h
    $y = y + 1;$
`END`

Satz 1.12. Die Menge der primitiv rekursiven Funktionen entspricht der Menge der LOOP–Programme, d.h.

1. Es gibt zu jeder primitiv rekursiven Funktion ein LOOP–Programm, welches diese programmiert (Lemmata 1.9, 1.10, 1.11).
2. Zu jedem LOOP–Programm gibt es eine primitiv rekursive Funktion, die dessen Bedeutung beschreibt. (ÜA)

Anmerkung. Dieser Satz bedeutet, dass man jedes *primitiv rekursive Problem* sowohl als rekursive Funktion, wie auch als Schleife implementieren kann.

1.3 μ–Rekursive Funktionen

Es stellt sich nun die Frage, ob alle Programme, die auf Computern implementierbar sind, primitiv rekursive Programme sind.

Betrachten wir zunächst die Endlosschleife.

Beispiel. Das einfachste Programm einer beliebigen Programmiersprache mit Schleifen, das die Endlosschleife repräsentiert, ist gegeben durch:

```
while (true){;}
```

Dieses Programm lässt sich nicht als primitiv rekursive Funktion darstellen, da alle primitiv rekursiven Funktionen total sind (Wenn ein Programm für eine bestimmte Eingabe keine Ausgabe liefert, so ist die Funktion, die dieses Programm programmiert an der Stelle nicht definiert, also ist die Funktion nicht total).

Neben den nicht totalen Funktionen gibt es noch eine weitere Klasse von Funktionen, die zwar total, aber trotzdem nicht primitiv rekursiv sind. Betrachten wir dazu einmal den Aufbau der Arithmetik (Addition, Multiplikation, Potenz, …):

1. Addition wird zurückgeführt auf s–Funktion (Nachfolger)
2. Multiplikation wird zurückgeführt auf die Addition.
3. Potenz wird zurückgeführt auf die Multiplikation.

   ...

Wir bilden eine Funktion, deren eines Argument angibt, wieviele Schritte in der eben aufgezeigten Reihe gegangen werden soll. Diese Idee führt zur Ackermannfunktion.

Definition 1.13 (Ackermannfunktion)**.** Die **Ackermannfunktion** $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist durch folgende Gleichung definiert:
$A(0, y) = y + 1$
$A(x, 0) = A(x - 1, 1)$
$A(x, y) = A(x - 1, A(x, y - 1))$

Lemma 1.14. Die Ackermannfunktion ist nicht primitiv rekursiv. Die Ackermannfunktion ist auf dem Rechner implementierbar.

Beweis. Als Beweis geben wir die Java–Funktion `Ackermann` an:

```
int A(int x, int y)
{
   if(x == 0) return(y+1);
       else if(y == 0) return A(x-1,1);
   else { return A(x-1, A(x,y-1)); }
}
```

□

Zusammenfassung:

1. Die Endlosschleife ist implementierbar, aber nicht primitiv rekursiv.
2. Die Ackermannfunktion ist implementierbar aber nicht primitiv rekursiv.

Die Klasse der implementierbaren Funktionen ist also "größer" als die Klasse der primitiv rekursiven Funktionen.

Definition 1.15 (μ–Operator)**.** Sei $h : \mathbb{N} \times \mathbb{N}^k \dashrightarrow \mathbb{N}$ eine partielle Funktion. Dann ist der μ–**Operator** definiert durch:

$$(\mu y)[h(y, x_1, \ldots, x_k) = 0] = min\{y \mid h(y, x_1, \ldots, x_k) = 0\} =: z,$$

wobei $h(y, x_1, \ldots, x_k)$ für alle $y < z$ definiert ist (also $\neq \perp$).

Wenn kein solches Minimum existiert, ist der μ–Operator undefiniert.

Definition 1.16 (Erfüllt das μ–Operator - Schema)**.** Man sagt $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ **erfüllt das** μ–**Operator–Schema**, wenn

$$f(x_1, \ldots, x_k) = (\mu y)[h(y, x_1, \ldots, x_k) = 0].$$

Beispiel. Die beiden folgenden Beispiele zeigen, wie mit Hilfe des μ–Operator–Schemas die Endlosschleife und die Ackermannfunktion beschrieben werden kann.

1. Sei $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $h(y, x) = 1$. Dann ist durch $f(x) = (\mu y)[h(y, x) = 0]$ die Funktion $f : \mathbb{N} \dashrightarrow \mathbb{N}$ mit $f(x) = \perp$ gegeben.
2. Sei $h : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{N}$ mit $h(y, x_1, x_2) = \begin{cases} 0 \text{ falls } A(x_1, x_2) = y \\ 1 \text{ sonst} \end{cases}$

   Dann gilt $A(x_1, x_2) = (\mu y)[h(y, x_1, x_2) = 0]$

Anmerkung. Grundsätzlich: Die Entscheidung, ob die Anwendung der Funktion h den Wert 0 ergibt, ist primitiv rekursiv.

- Im 1. Beispiel ist das offensichtlich.
- Im 2. Beispiel bedeutet das, dass die Entscheidung, ob die Ackermannfunktion für eine gegebene Eingabe einen bestimmten Wert annimmt, *primitiv rekursiv ist.*

  Dies gilt, obwohl die Ackermannfunktion selbst *nicht primitiv rekursiv ist.*

  Idee: Dies kann man sich plausibel machen, da die Ackermannfunktion streng monoton steigend ist und somit die Frage, ob die Ackermannfunktion einen bestimmten Wert annimmt nach einer vorbestimmten Anzahl von Schritten entschieden ist. Also kann dies durch ein LOOP–Programm berechnet werden. Das wiederum heißt aber, dass die Entscheidungsfunktion primitiv rekursiv ist.

Definition 1.17 (Menge der rekursiven Funktionen)**.** Die Menge der **rekursiven Funktionen** besteht aus den Grundfunktionen und der Anwendung der Komposition, des primitiven Rekursionsschemas und der μ–Operator Bildung.

1.4 WHILE–Programme

Es bleibt jetzt noch die Erweiterung der LOOP–Programme, so dass auch der μ–Operator programmiert werden kann.

Definition 1.18 (WHILE–Programme)**.** Die Programmiersprache der **WHILE–Programme** umfasst Variablen, Konstanten, Trennsymbole, Operationssymbole, sowie alle Schlüsselwörter der LOOP–Programme.

Zusätzlich wird noch das Schlüsselwort „`WHILE`“ hinzugefügt.

Ein WHILE–Programm wird gebildet nach den Regeln 1. - 4. aus Definition 1.8 und der Regel:

5. Sei P ein WHILE–Programm und x_i eine Variable, so ist

   $$\texttt{WHILE}\ (x_i \neq 0)\ \texttt{DO}\ \ P\ \ \texttt{END}$$

   ein WHILE–Programm.

Bedeutung bzw. Semantik: P wird solange ausgeführt, wie $x_i \neq 0$ ist.

Lemma 1.19. Sei $h : \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ eine Funktion aus der Menge der rekursiven Funktionen, die durch das WHILE–Programm P_h programmiert wird. Dann gibt es ein WHILE–Programm P_f, das

$$f(x_1, \ldots, x_k) = (\mu y)[h(y, x_1, \ldots, x_k) = 0]$$

programmiert.

Beweis. Es wird jetzt das WHILE–Programm P_f angegeben:

$x_{k+1} = x_k, x_k = x_{k-1}, \ldots, x_2 = x_1;$
$z_{k+1} = x_{k+1}, z_k = x_k, \ldots, z_2 = x_2;$
$y = 0;$
$x_1 = y;$
$P_h;$
`WHILE` $(x_0 \neq 0)$ `DO`
    $y = y + 1;$
    $x_1 = y;$
    $x_{k+1} = z_{k+1}, x_k = z_k, \ldots, x_2 = z_2;$
    $P_h;$
`END`;
$x_0 = y;$

□

Satz 1.20. Die Menge der rekursiven Funktionen entspricht der Klasse der WHILE–Programme, d.h.:

1. es gibt zu jeder rekursiven Funktion ein WHILE–Programm, das diese programmiert.
2. zu jedem WHILE–Programm gibt es eine rekursive Funktion, die die Berechnung beschreibt.

Beweis. 1. Satz 1.12, Lemma 1.19.

2. Satz 1.12 und Übungs–Aufgabe.

□

Satz 1.21 (Halteproblem)**.** Es gibt kein WHILE–Programm das als Eingabe ein beliebiges WHILE–Programm W (in codierter Form) und eine Eingabe $(n_1, \ldots, n_k)$ für W nimmt und als Ausgabe 1 liefert, wenn die Ausführung von W mit der Eingabe $(n_1, \ldots, n_k)$ keine Endlosschleife ergibt und 0 liefert, wenn sich eine Endlosschleife ergibt.

Spezifikation des Halteproblems:

1. Deklarationen:
   Datentypen: $< \mathcal{W}; execute >$, wobei $\mathcal{W}$ die Menge der codierten WHILE–Programme ist und $execute : \mathcal{W} \times \mathbb{N}^k \dashrightarrow \mathbb{N}$ das Programm mit der gegebenen Eingabe ausführt.

2. Eingabe:
   $p \in \mathcal{W}, (n_1, \ldots, n_k) \in \mathbb{N}^k$

3. Ausgabe:
   $x \in \{0, 1\}$

4. Nachbedingung:
   $(((p, (n_1, \ldots, n_k)) \in Defexecute) \rightarrow x_0 = 1)\ xor$
   $(((p, (n_1, \ldots, n_k)) \notin Defexecute) \rightarrow x_0 = 0)$
   (mit *Def* bezeichnet man den Definitionsbereich)

Das heißt es gibt Probleme, die man zwar spezifizieren aber nicht programmieren kann.

Korollar 1.22. Es gibt durch prädikatenlogische Formeln beschreibbare Funktionen, die nicht durch WHILE–Programme programmierbar sind, also auf Rechnern nicht implementierbar sind.

1.5 Einführung Turingmaschine

Turingmaschinen sind Automaten. Ein Automat ist ein *Objekt*, das aus einer Menge von Zuständen und einer Menge von Zustandsübergängen besteht. Ein Automat erwartet eine Eingabe und verarbeitet diese, indem er Schritt für Schritt unterschiedliche Zustände annimmt. Schließlich liefert der Automat eine Ausgabe.

In Turingmaschinen steuert eine zentrale Einheit mit endlich vielen Zuständen die Verarbeitung. Auf einem Ein-/Ausgabeband können endlich viele Symbole stehen. Beim Start steht ein Eingabewort auf dem Band. Ein Lese-/Schreibkopf (LSK) kann Zeichen vom Band lesen und auf das Band schreiben. Anfangs steht der LSK auf dem linkesten Zeichen (Abbildung 1.1).

Formal kann man eine Turingmaschine wie folgt definieren.

Definition 1.23 (Turingmaschine)**.** Eine Turingmaschine (TM) sei gegeben durch eine Struktur (Q, Σ, Γ, δ, q_0, B, F) mit

- Q ist eine endliche, nicht-leere Menge von Zuständen

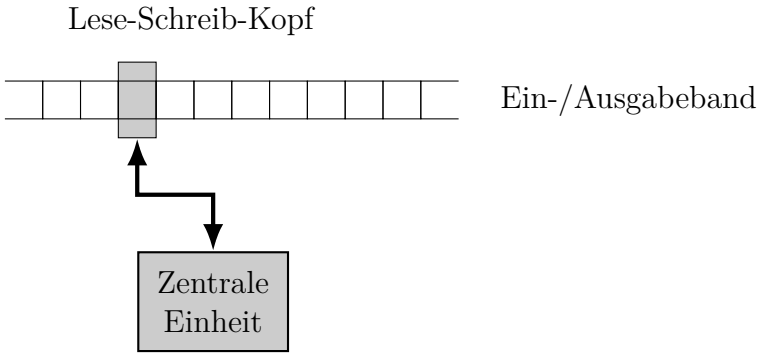

Abbildung 1.1: Turingmaschine

- Σ ist eine endliche Menge (Eingabealphabet)
- Γ ist eine endliche Menge (Symbole auf dem Band), d.h. $\Sigma \subset \Gamma$
- δ: Q $\times$ $\Gamma \rightarrow$ Q $\times$ Γ $\times$ D ist die Übergangsfunktion

  D = {L, R, N} gibt Werte für die Richtung der LSK-Bewegung an
- q_0 ist der Startzustand,
- B ist das Leerzeichen, d.h. das Zeichen, das außerhalb des Eingabewortes auf dem Band steht

  Es muss gelten: B $\in \Gamma$, aber B $\notin \Sigma$
- F $\subseteq$ Q ist die Menge der Endzustände.

Arbeitsweise der Turingmaschine:

- Turingmaschine beginnt im Startzustand, LSK zeigt auf das am weitesten links stehende Zeichen der Eingabe.
- Aktuelle Aktion (aktueller Schritt) hängt vom Zustand und vom gelesenen Zeichen ab.
- Schritt besteht aus drei Teilaktionen:
  - Wechsel des Zustandes
  - Schreiben eines Symbols auf das Band
  - Bewegen des LSK um eine Position nach rechts bzw. nach links oder stehenbleiben
- Die Turingmaschine hält an, wenn die Übergangsfunktion nicht definiert ist.

1.6 Turingmaschine als Automat zur Berechnung von rekursiven Funktionen

Man kann die Turingmaschine auch als Automat zur Berechnung von rekursiven Funktionen betrachten. Folgende Defintion beschreibt, wie die Berechnung erfolgt.

Definition 1.24 (Turing–berechenbar)**.** Eine n–stellige Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt *turing–berechenbar*, wenn es eine Turingmaschine $TM = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ B,\ F)$ gibt, für die folgendes gilt: Schreibt man die Eingabewerte $w_1, w_2, \ldots, w_n$ durch Blanks getrennt auf das Band

$$\ldots\, B\, B\, w_1\, B\, w_2\, B\, \ldots\, B\, w_n\, B\, B\, \ldots,$$

so hält die Turingmaschine TM in einem Finalzustand an und auf dem Band steht

$$\ldots\, B\, B\, f(\, w_1, w_2, \ldots, w_n\,)\, B\, B\, \ldots.$$

Anmerkung. Es muss eine Codierung der natürlichen Zahlen in das Eingabealphabet Σ der Turingmaschine geben.

Diese kann die Binärdarstellung der natürlichen Zahlen sein.

Oftmals wird aber auch die unäre Darstellung der natürlichen Zahlen (n wird durch n-mal 1 dargestellt) verwendet.

Satz 1.25 (Hauptsatz der Algorithmentheorie)**.** Die Menge der turing–berechenbaren Funktionen und die Menge der rekursiven Funktionen sind identisch.

Beweis.

- Es ist zu beweisen, dass es für jede rekursive Funktion eine Turingmaschine gibt, die diese berechnet.
- Es ist zu beweisen, dass es zu jeder Turingmaschine eine rekursive Funktion gibt, die dessen Berechnung beschreibt.

Für den Beweis wollen wir auf die einschlägige Literatur verweisen. □

Wir wollen hier als Beispiele die Turingmaschinen zur Berechnung der Grundfunktionen angeben:

Beispiel.

1. Nullstellige Nullfunktion C_0^0:

$$TM = (\{q_0, q_1\},\ \{0,1\},\ \{0,1,\#\}, \{(q_0,\#) \mapsto (q_1, 0, N)\},\ q_0, \#, \{q_1\})$$

2. Einstellige Nullfunktion C_1^0:

   Idee der Übergangsfunktion:

   - Man liest die Eingabe Zeichen für Zeichen und ersetzt jede Ziffer durch ein Blank
   - Wenn man ein Blank liest, schreibt man eine 0 und geht in den Finalzustand.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_0, q_1\},\ \{0,1\},\ \{0,1,\#\}, \delta,\ q_0, \#, \{q_1\})$$

   mit

$$\begin{aligned}\delta = \{&(q_0, 0) \mapsto (q_0, \#, R),\\ &(q_0, 1) \mapsto (q_0, \#, R),\\ &(q_0, \#) \mapsto (q_1, 0, N)\}\end{aligned}$$

3. Projektionsfunktionen Π_i^n:

   Idee der Übergangsfunktion:

   - Man liest die Eingaben Zeichen für Zeichen und ersetzt jede Ziffer durch ein Blank bis zur Eingabe i-1.
   - Bei jedem gelesenen Blank kommt man in einen neuen Zustand, so kann man feststellen, bei welcher Eingabe man sich im Moment befindet.
   - Die i. Eingabe lässt man stehen.
   - Es werden alle Zeichen ab der Eingabe i+1 durch Blanks ersetzt.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_1, q_2, \ldots, q_i, q_{i+1}, q_{i+2}\},\ \{0,1\},\ \{0,1,\#\}, \delta,\ q_1, \#, \{q_{i+2}\})$$

mit

$$\begin{aligned}\delta = &\{(q_j, e) \mapsto (q_j, \#, R) \mid 1 \leq j \leq i-1,\ e \in \{0,1\}\}\\ &\cup \{(q_j, \#) \mapsto (q_{j+1}, \#, R) \mid 1 \leq j \leq i-1\}\\ &\cup \{(q_i, e) \mapsto (q_i, e, R) \mid e \in \{0,1\}\}\\ &\cup \{(q_i, \#) \mapsto (q_{i+1}, \#, R)\}\\ &\cup \{(q_{i+1}, e) \mapsto (q_{i+1}, \#, R) \mid e \in \{0,1\}\}\\ &\cup \{(q_{i+1}, \#) \mapsto (q_{i+2}, \#, R)\}\\ &\cup \{(q_{i+2}, 0) \mapsto (q_{i+1}, \#, R), (q_{i+2}, 1) \mapsto (q_{i+1}, \#, R)\}\end{aligned}$$

4. Nachfolger-Funktion:

   Idee der Übergangsfunktion:

   - Zunächst einmal muss man in einem Zustand q_0 von links nach rechts über das Eingabewort gehen, damit der LSK auf dem letzten Zeichen steht.
   - Danach wechselt man in den Zustand q_1. Nun kann man die Binärzahl nach links zurückgehen: Befindet sich eine 1 auf dem Band, wird sie durch eine 0 überschrieben und man verbleibt im Zustand q_1 ($1 + 1 = 0$ Übertrag 1).
   - Sobald man auf eine 0 oder das Blank- Zeichen stößt, schreibt man eine 1 aufs Band und wechselt in den akzeptierenden Zustand q_2.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,\#\}, \delta, q_0, \#, \{q_2\})$$

   mit der Übergangsfunktion δ dargestellt als Tabelle:

Zustand	**Eingabesymbol**		
	0	1	#
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \#, L)$
q_1	$(q_2, 1, N)$	$(q_1, 0, L)$	$(q_2, 1, N)$
q_2	-	-	-

Definition 1.26 (k–Band Turingmaschine)**.** Die Maschine $TM = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ B,\ F, k)$ heißt *k–Band Turingmaschine* mit

1. $\Sigma,\ \Gamma,\ q_0,\ B,\ F$ wie in Def. 1.23
2. $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times D^k$ mit $D = \{L, R, N\}$.

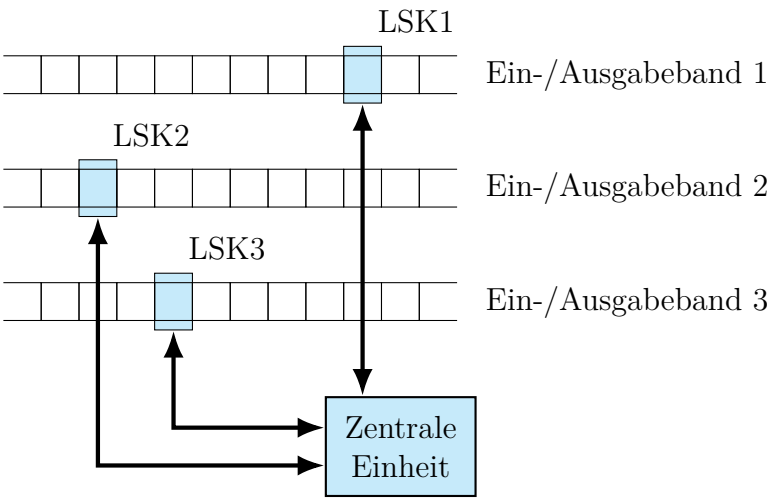

Abbildung 1.2: k-Band Turingmaschine

(vgl. Abbildung 1.2)

Am Anfang der Berechnung befindet sich die Eingabe auf dem ersten Band und die restlichen $k-1$ Bänder sind mit B besetzt. Am Ende der Berechnung befindet sich das Ergebnis auf dem ersten Band und die restlichen $k-1$ Bänder sind wiederum mit B besetzt.

Satz 1.27. Für eine Funktion f sind folgende Aussagen äquivalent:

1. f ist durch eine (1–Band) Turingmaschine berechenbar
2. f ist durch eine k–Band Turingmaschine berechenbar

Anmerkung. Weitere berechnungsäquivalente Turingmaschinenmodelle sind

- Turingmaschinen mit (mehrdimensionalem) Speicher (Abb. 1.3)
- Turingmaschinen mit getrenntem Ein– und Ausgabeband
- Turingmaschinen mit k Bändern und h_i Köpfen auf Band i, $1 \leq i \leq k$ und $h_i \in \mathbb{N}$

Wenn wir uns nochmals den Beweis–Ansatz von Satz 1.25 vergegenwärtigen, wird uns bewusst, dass man für jede rekursive Funktion eine eigene Turingmaschine benötigt. Dies ist wenig sinnvoll, da die Turingmaschine als abstraktes Modell für den Computer dienen soll, würde das bedeuten, dass man für jedes Programm einen eigenen Computer bräuchte.

Dieses Problem wird durch die *universelle Turingmaschine* gelöst.

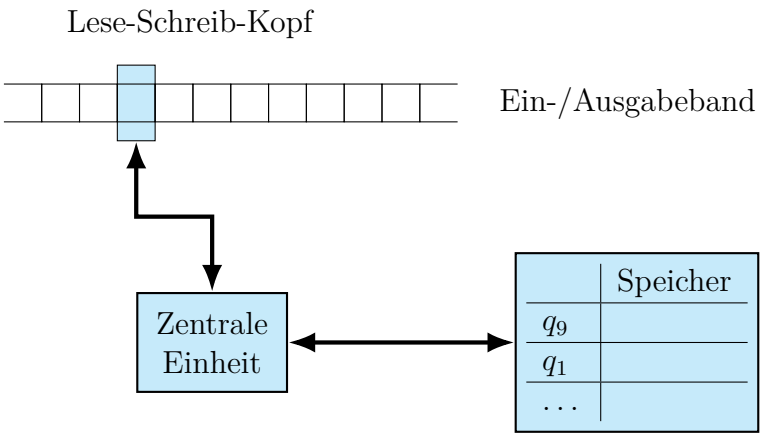

Abbildung 1.3: Modifizierte Turingmaschine: Speicher

Satz 1.28 (Satz über die universelle Turingmaschine)**.** Es gibt eine *universelle Turingmaschine* U, die bei codierter Eingabe des Programms einer beliebigen Turingmaschine T und einer beliebigen Dateneingabe w das gleiche Ergebnis liefert, wie die Turingmaschine T angewendet auf w.

Beweisidee:

- Benutzung einer 3–Band Turingmaschine
- Eingabe w auf Band 1
- Codierung der Funktion δ_T von T auf Band 2
- aktueller Zustand auf Band 3
- Funktion δ_U bearbeitet codiertes δ_T (Band 2) und verändert dabei Band 1 (entspricht dem Band von T) und Band 3 (Zustand von T).

Zusammenfassung

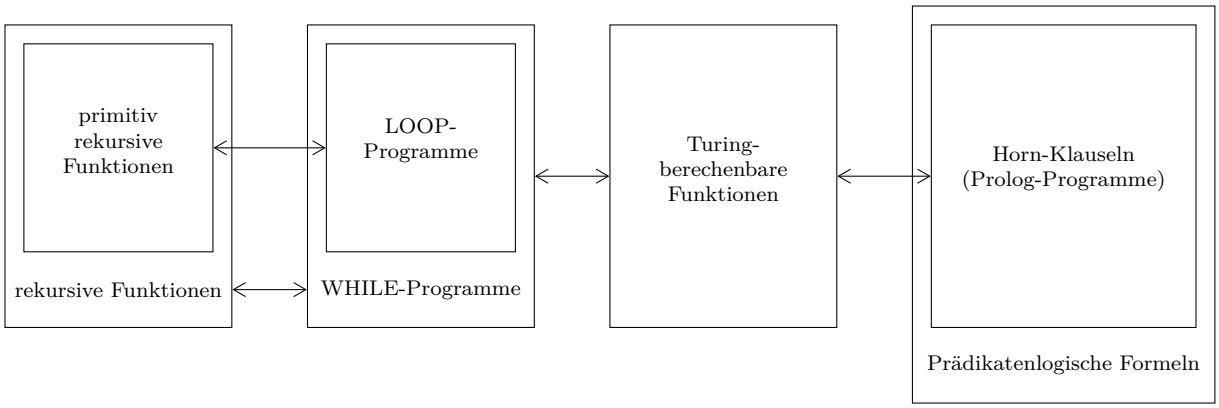

Abbildung 1.4: Hauptsatz der Algorithmentheorie