Berechenbarkeit und Formale Sprachen

Abschnitte 1.6 bis 1.8

(Halteproblem, Satz von Rice,
rekursive Aufzählbarkeit)

Rolf Wanka
Universität Erlangen-Nürnberg

rwanka@cs.fau.de

7. November 2008

1.6 Unentscheidbare Probleme

Gibt es Grenzen für das, was Turingmaschinen berechnen können? Wir werden sehen, daß es solche Grenzen gibt, die sehr wichtige Probleme betreffen. Genauer gesagt: Wir werden zeigen, daß Probleme, von denen wir uns wünschten, daß sie lösbar wären, algorithmisch nicht lösbar sind!

Die Gödelnummer einer Turingmaschine M besteht nur aus Buchstaben aus $\{0,1,\#\}$. Durch die Abbildung

$$\begin{array}{rcl} 0 & \mapsto & 00 \\ 1 & \mapsto & 01 \\ \# & \mapsto & 11 \end{array}$$

dieser Zeichen kommen wir sogar nur mit dem Alphabet $\{0,1\}$ aus. Wir können also jede 1-Band-Turingmaschine M eindeutig durch eine Zeichenkette $\langle M\rangle \in \{0,1\}^*$ kodieren. Da der gleiche Kodierungstrick für beliebige Bandalphabete Σ funktioniert, gehen wir im folgenden davon aus, daß für alle Turingmaschinen, die uns begegnen, $\Sigma = \{0,1\}$ ist.

1.13 Definition:
Die Sprache

$$H := \{\langle M\rangle w \mid M \text{ ist eine deterministische 1-Band-Turingmaschine, die, gestartet mit } w \text{, hält}\}$$

ist das (allgemeine) *Halteproblem*.

> Beachten Sie, daß wir hier eine Menge von Zeichenfolgen (ein anderes Wort für „Menge von Zeichenfolgen“ ist ja *Sprache*) als *Problem* bezeichnen. Das ist eine Konvention, hinter der steckt, daß wir letztlich das zugehörige *Wortproblem* oder auch, in äquivalenter Bedeutung, *Entscheidungsproblem* meinen. Wir haben also die Menge H definiert und meinen im Hinterkopf für alle $w \in \{0,1\}^*$ die (maschinelle) Beantwortung der Frage „$w \in H$?“. Trotzdem: Nur die Menge H ist das Halteproblem. Im übrigen gilt auch für H: $H \subseteq \{0,1\}^*$.

1.14 Satz: (Turing, 1936)
H ist unentscheidbar (eine andere, aber äquivalente Formulierung lautet: *H ist nicht rekursiv).*

Beweis:
Wir nehmen an, daß es eine Turingmaschine M_H gibt, die H entscheidet. D. h. M_H hält auf *jeder* Eingabe x, und man kann am erreichten Zustand erkennen, ob $x \in H$ ist oder nicht!

Diese Annahme werden wir nun auf einen Widerspruch führen.

Wenn es M_H gibt, dann gibt es auch die folgende Turingmaschine M_{schlau}:

Turingmaschine M_{schlau}

(1) die Eingabe sei y
(2) falls $y = \langle M\rangle$, dann

(3) entscheide mittels M_H, ob $\langle M\rangle\langle M\rangle \in H$
(4) falls ja: schreibe nach rechts endlos viele 1en auf das Band
(5) falls nein: bleibe stehen

Die Turingmaschine M_{schlau} können wir explizit hinschreiben („programmieren“), falls uns jemand die δ-Tabelle von M_H gibt (die als „Unterprogramm“ aufgerufen würde).

Was passiert, wenn wir M_{schlau} mit der Eingabe $y = \langle M_{\text{schlau}}\rangle$ starten? Nun, es gibt nur zwei Möglichkeiten, nämlich die, daß die Maschine hält („Fall (a)“), und die, daß sie nicht hält („Fall (b)“).

(a) Wenn M_{schlau}, gestartet mit $\langle M_{\text{schlau}}\rangle$, hält, heißt das, daß die Abfrage in (3) die Antwort „nein“ ergeben hatte, also $\langle M_{\text{schlau}}\rangle\langle M_{\text{schlau}}\rangle \notin H$. D. h. wiederum, daß M_{schlau}, gestartet mit $\langle M_{\text{schlau}}\rangle$, nicht hält, im Widerspruch zum Anfang der Argumentation!

Also kann Fall (a) nicht gelten. Nun betrachten wir Fall (b).

(b) Wenn M_{schlau}, gestartet mit $\langle M_{\text{schlau}}\rangle$, endlos läuft, muß sich die Maschine in Zeile (4) befinden. Dorthin kommt sie nur, wenn die Abfrage in (3) die Antwort „ja“ ergeben hatte, also $\langle M_{\text{schlau}}\rangle\langle M_{\text{schlau}}\rangle \in H$. D. h. wiederum, daß M_{schlau} gestartet mit $\langle M_{\text{schlau}}\rangle$ hält, im Widerspruch zum Anfang der Argumentation!

Also kann keiner der beiden Fälle eintreten, wir müssen somit irgendwo von etwas ausgegangen sein, das falsch ist. Unsere gesamte Argumentation hat aber nur eine Schwachstelle, nämlich die Annahme, daß es M_H gibt. M_H gibt es also gar nicht, und als Konsequenz ist H nicht entscheidbar! □

1.15 Satz:
(a) H ist rekursiv aufzählbar.
(b) Das Komplement von H, d. h. $\overline{H} = \{0,1\}^* \setminus H$, ist nicht rekursiv aufzählbar.

Beweis:
(a) Die universelle Turingmaschine M_0 aus Satz 1.14 ist der rekursive Aufzähler von H, d. h.

$$\langle M\rangle w \in H \iff M_0,\ \text{gestartet mit } \langle M\rangle w, \text{ hält}$$

(b) Wäre $\overline{H}$ rekursiv aufzählbar (mittels einer Turingmaschine $\tilde{M}$), könnte man folgende Turingmaschine programmieren:

Entscheider für H

die Eingabe sei $\langle M\rangle w$
führe *gleichzeitig* aus und stoppe, wenn eine stoppt:
– starte mit $\langle M\rangle w$ auf Band 1 die Turingmaschine M_0 , die die Wörter aus H akzeptiert (aus Teil (a))
– starte mit $\langle M\rangle w$ auf Band 2 die Turingmaschine $\tilde{M}$, die die Wörter aus $\overline{H}$ akzeptiert
hält M_0, halte akzeptierend, hält $\tilde{M}$, halte verwerfend.

Diese Turingmaschine hält für jede Eingabe, denn eine der beiden Untermaschinen, M_0 oder $\tilde{M}$, muß ja halten! Also entscheidet diese Turingmaschine das Halteproblem. Folglich (wegen Satz 1.14) gibt es $\tilde{M}$ nicht. $\square$

Mit der schlechten Nachricht der Unentscheidbarkeit des Halteproblems beginnt nun eine ganze Folge von derartigen schlechten Nachrichten.

1.16 Definition:
Die Sprache

$$H_\varepsilon := \{\langle M\rangle \mid M \text{ ist det. 1-Band-TM, die, gestartet mit leerem Band, hält}\}$$

heißt *initiales Halteproblem*.

> Hier ist die Namensgebung in den einschlägigen Lehrbüchern nicht ganz eindeutig. Das macht jeder Autor beinahe nach Lust und Laune, aber immer begründet. Manchmal wird H_ε auch als *spezielles Halteproblem* bezeichnet oder – etwas phantasielos – als *Halteproblem mit leerem Band*.

1.17 Satz:
H_ε ist nicht entscheidbar.

Beweis:
Wir zeigen, daß man einen Entscheidungsalgorithmus für H programmieren könnte, wenn H_ε entscheidbar wäre. Dazu programmieren wir um die Eingabe zum Halteproblem H so herum, daß wir eine Eingabe des initialen Halteproblems H_ε bekommen.

Gegeben sei also die Eingabe $\langle M\rangle w$ und damit die Frage „$\langle M\rangle w \in H$?“. Wie kann man diese Frage beantworten, wenn man die Frage „$\langle M'\rangle \in H_\varepsilon$?“ für alle $\langle M'\rangle$ beantworten könnte?

Betrachten Sie zu $\langle M\rangle w$ folgende Turingmaschine:

feste_Maschine$_{\langle M\rangle w}$
(1) die Eingabe sei x
(2) starte M mit w $\quad$ ($*$ das ist kein Tippfehler $*$)
(3) falls $x = \varepsilon$, dann halte.

Nun nehmen wir an, daß wir H_ε mittels der Turingmaschine $\tilde{M}$ entscheiden können. Dann haben wir:

(a) Wenn $\tilde{M}$ bei Eingabe $\langle$feste_Maschine$_{\langle M\rangle w}\rangle$ akzeptierend hält, muß feste_Maschine$_{\langle M\rangle w}$ bis Zeile (3) kommen, was nur möglich ist, wenn M, gestartet mit w, hält. Also ist $\langle M\rangle w \in H$.

(b) Wenn $\tilde{M}$ bei Eingabe $\langle$feste_Maschine$_{\langle M\rangle w}\rangle$ verwerfend hält, kann feste_Maschine$_{\langle M\rangle w}$ nicht bis Zeile (3) kommen, was nur möglich ist, wenn M, gestartet mit w, nicht hält. Also ist $\langle M\rangle w \notin H$.

D. h.: Die Antwort von $\tilde{M}$ auf die Eingabe $\langle\text{feste\_Maschine}_{\langle M\rangle w}\rangle$ ist die Antwort auf die Frage „$\langle M\rangle w \in H$?“!
Und damit kann es $\tilde{M}$ nicht geben, mithin ist H_ε nicht entscheidbar. □

Den Trick, das Halteproblem (oder dann weiter andere unentscheidbare Probleme) durch Drumherumprogrammieren zu maskieren, kann man immer wieder anwenden. Er heißt *Reduktion* und wird im weiteren ausführlich behandelt.

> Eine permanente Quelle für Mißverständnisse bei Newcomern auf diesem Gebiet ist, wer auf wen reduziert wird. Hier im Beweis wurde das Halteproblem H auf das initiale Halteproblem H_ε reduziert. Im Fall der Unentscheidbarkeit wird also das Problem, von dem man bereits die Unentscheidbarkeit weiß, auf das Problem, von dem man es noch nicht weiß, reduziert. Machen Sie sich bitte unbedingt mit dieser (in der Tat verstehbaren) Sprechweise vertraut. Und: Üben Sie Reduktionen!

1.7 Reduktionen und der Satz von Rice

Bislang haben wir uns bei Turingmaschinen noch gar nicht dafür interessiert, was nach dem Halten auf dem Band steht. Im Vordergrund stand nur, in welchem Zustand sie stoppen, bzw., ob sie überhaupt stoppen. Wenn wir das, was auf dem Band steht, als *Ausgabe* in Abhängigkeit vom Eingabewort bezeichnen, haben wir den Schritt zur Berechnung von Funktionen vollzogen.
Formal können wir damit den Ausdruck „M berechnet eine Funktion f“ wie folgt definieren, wobei die *Berechenbarkeit* eine Eigenschaft von f sein soll, wie z. B. in der Analysis die Monotonie, Stetigkeit oder Differenzierbarkeit sehr wichtige Eigenschaften von Funktionen sind.

1.18 Definition:
Eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ heißt *berechenbar*, wenn es eine (1-Band-)Turingmaschine M_f gibt, für die mit $x \in \{0,1\}^*$ gilt:

- Ist $f(x)$ definiert, so hält M_f, gestartet mit x, und $f(x)$ steht dann auf dem Band.
- Ist $f(x)$ nicht definiert, so hält M_f, gestartet mit x, nicht.

Ist f total, d. h. für alle $x \in \{0,1\}^*$ definiert, und berechenbar, so heißt f *total berechenbar* oder auch *total rekursiv*. Diese beiden Begriffe sind synonym.

Insbesondere können wir mit dieser Definition auch noch einmal die Begriffe *entscheidbar* und *rekursiv aufzählbar* beschreiben:

- Eine Sprache L ist genau dann entscheidbar, wenn die sog. *charakteristische* Funktion $\chi_L : \{0,1\}^* \to \{0,1\}$ mit
$$\chi_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{falls } x \notin L \end{cases}$$
total berechenbar ist.

- Eine Sprache L ist genau dann rekursiv aufzählbar, wenn die (partielle) Funktion

$$\chi'_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ \text{undef} & \text{falls } x \notin L \end{cases}$$

  berechenbar ist.

Sehen wir uns nun den Beweis von Satz 1.17 noch einmal an. Wir können ihn in mehrere Teile gliedern. Was wir dort gemacht haben, ist, daß wir mit Hilfe eines (nicht existierenden) Entscheiders für H_ε einen Entscheider für H programmiert haben. Dabei bekommt der H_ε-Entscheider eine Turingmaschine in Form ihrer Gödelnummer (ihres Programms) als Eingabe, so daß die Antwort des H_ε-Entscheiders direkt (ohne Modifikationen) die Antwort auf die Frage „$\langle M\rangle w \in H$?" ist.
Das wollen wir im folgenden abstrahieren.

1.19 Definition:
Seien L_1 und L_2 Sprachen über dem Alphabet $\{0,1\}$. Eine Reduktion ist eine total berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ für die gilt:

$$x \in L_1 \iff f(x) \in L_2$$

Wir schreiben dann „$L_1 \leq L_2$" und sagen, „L_1 wird (mittels f) auf L_2 reduziert".

> Beachten Sie, daß f total sein muß. Das wird bei Reduktionen von Newcomern gerne übersehen. Eine weitere, sprachliche Fehlerquelle ist die, wer auf wen reduziert wird. Eine Eselsbrücke dafür, sich das zu merken, ist folgendes: Man spricht gegen die Pfeilrichtung „$\leq$". „$\leq$" können Sie (ungenau, aber hilfreich) als „ist leichter als" interpretieren.
>
> Beachten Sie auch, daß diese Definition fordert, daß f tatsächlich total berechenbar sein muß, und man nicht von irgendwelchen Annahmen ausgehen darf („Angenommen, L wäre entscheidbar. Dann sei f folgende Funktion ...").
>
> Und beachten Sie, daß $(x \in L_1 \iff f(x) \in L_2)$ gefordert ist. Manchmal ist es einfacher, eine Funktion f anzugeben, für die $(x \in L_1 \iff f(x) \notin L_2)$ gilt. Die ist dann keine Reduktion im Sinn der obigen Definition.
>
> Eine kleine Diskussion von Schwierigkeiten beim Verstehen und Anwenden des Reduktionskonzepts finden Sie am Ende dieses Abschnitts auf S. 9.

Die folgende total berechenbare Funktion f ist eine solche *Reduktionsfunktion*, die zeigt, daß $H \leq H_\varepsilon$ ist. $\langle \text{feste\_Maschine}_{\langle M\rangle w}\rangle$ ist dabei die Gödelnummer (das Programm) der im Beweis von Satz 1.17 programmierten Turingmaschine.

$$f(x) = \begin{cases} \langle \text{feste\_Maschine}_{\langle M\rangle w}\rangle & \text{falls } x \text{ von der Form } \langle M\rangle w \\ 0 & \text{sonst} \end{cases}$$

Ganz wichtig ist hierbei, daß die Eigenschaft „x von der Form $\langle M\rangle w$" entscheidbar ist! Dadurch kann nämlich f durch die folgende Turingmaschine berechnet werden:

$H\_auf\_H_\varepsilon\_Reduzierer$ (* berechnet die Reduktionsfunktion f *)
die Eingabe sei x
falls x von der Form $\langle M\rangle w$ ist: gib $\langle feste\_Maschine_{\langle M\rangle w}\rangle$ aus
anderenfalls: gib 0 aus

Die durch die Turingmaschine $H\_auf\_H_\varepsilon\_Reduzierer$ berechnete Funktion f ist total und berechenbar. Außerdem gilt

$$x \in H \quad\Rightarrow\quad x = \langle M\rangle w \text{ und } M\text{, gestartet mit } w\text{, hält} \Rightarrow f(x) = \langle feste\_Maschine_{\langle M\rangle w}\rangle \in H_\varepsilon$$

$$x \notin H \quad\Rightarrow\quad \begin{cases} x = \langle M\rangle w \text{ und } M\text{, gestartet mit } w\text{, hält nicht} \\ \qquad\qquad\qquad \Rightarrow f(x) = \langle feste\_Maschine_{\langle M\rangle w}\rangle \notin H_\varepsilon \\ x \text{ nicht von der Form } \langle M\rangle w \Rightarrow f(x) = 0 \notin H_\varepsilon \end{cases}$$

also zusammen $\big(x \in H \iff f(x) \in H_\varepsilon\big)$ und damit insgesamt $H \leq H_\varepsilon$.

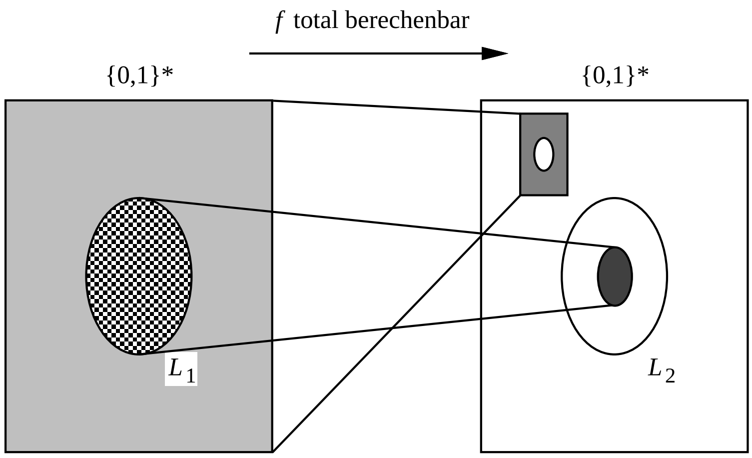

Abbildung 1.1: Wirkungsweise einer Reduktion (das Spiegelei-Bild)

Abbildung 1.1 (unser *Spiegelei-Bild* ☺) zeigt, was die Reduktionsfunktion f macht. Für jedes Wort links muß es ein Wort rechts geben (aber nicht unbedingt umgekehrt). Jedes Wort aus L_1 wird dabei in die Menge L_2 abgebildet, ebenso wird jedes Wort aus $\overline{L_1} = \{0,1\}^* \setminus L_1$ in die Menge $\overline{L_2}$ abgebildet. Die Antworten auf die Fragen „$x \in L_1$?" und „$f(x) \in L_2$?" sind deswegen identisch.

Reduktionen an sich kann man auch als Programmieraufgabe interpretieren: Wie würden Sie ein Programm schreiben, das die Sprache L_1 entscheidet, wenn Sie in einer externen Library eine Prozedur finden würden, die L_2 entscheiden kann? Konkret in unserem Beispiel Satz 1.17 kann man „$x \in H$?" entscheiden, wenn man den Entscheider für H_ε, den man in der Library gefunden hat, auf die Ausgabe von $H\_auf\_H_\varepsilon\_Reduzierer$, gestartet mit x, anwendet.

Im folgenden wird die beschriebene Intuition formaler untersucht, nämlich wie die Spracheigenschaften „entscheidbar/nicht entscheidbar" und „rekursiv aufzählbar/nicht rekursiv aufzählbar" durch die Eigenschaft $L_1 \leq L_2$ weitergegeben werden.

1.20 Satz:
Seien L_1 und L_2 Sprachen, und sei $L_1 \leq L_2$ mittels f.

(a) *Ist L_2 entscheidbar, dann ist auch L_1 entscheidbar.*

(b) *Ist L_2 rekursiv aufzählbar, dann ist es auch L_1.*

Beweis:
Der Beweis ist jetzt noch eine einfache Programmieraufgabe.

(a) Gegeben sei die Frage „$x \in L_1$?“ und eine Entscheidungsturingmaschine M_{L_2} für L_2. Der Entscheidungsalgorithmus für L_1 besteht darin, $f(x)$ zu berechnen und als Eingabe für M_{L_2} zu benutzen. Die Antwort von M_{L_2} auf die Frage „$f(x) \in L_2$?“ ist die Antwort auf die ursprüngliche Frage.

(b) geht analog, nur daß wir jetzt die Antwort „nein“ gar nicht mehr benötigen, sondern uns nur noch für Halten und Nichthalten interessieren. □

Die folgende Konsequenz bekommen Sie direkt aus Satz 1.20, indem Sie die Kontrapositionen der Aussagen bilden[1].

1.21 Korollar:
Seien L_1 und L_2 Sprachen, und sei $L_1 \leq L_2$.

(a) *Ist L_1 unentscheidbar, dann ist auch L_2 unentscheidbar.*

(b) *Ist L_1 nicht rekursiv aufzählbar, dann ist auch L_2 nicht rekursiv aufzählbar.*

Was wir als „Drumherumprogrammieren“ bezeichnet hatten, läßt sich derart verallgemeinern, daß man einen sehr mächtigen Satz beweisen kann, der, etwas ungenau formuliert, besagt, daß alle nichttrivialen semantischen[2] Eigenschaften von Turingmaschinen- und damit Computer-Programmen unentscheidbar sind.

Dazu bezeichnen wir im folgenden mit

$$\mathcal{R} = \{f \mid f : \{0,1\}^* \to \{0,1\}^* \text{ ist berechenbar}\}$$

die Menge der berechenbaren Funktionen.

Zu einer Teilmenge S, $S \subseteq \mathcal{R}$, bezeichne

$$\mathrm{Prog}(S) = \{\langle M\rangle \mid M \text{ berechnet eine Funktion } f \in S\}$$

die Menge *aller* (! ganz wichtig) Programme, d. h. Gödelnummern von Turingmaschinen, die die Funktionen aus S berechnen.

Die Mengen $\mathrm{Prog}(\emptyset)$ und $\mathrm{Prog}(\mathcal{R})$ sind entscheidbar durch einfache Syntaxanalyse, wie wir sie schon kennengelernt hatten. Die Eigenschaften der Wörter dieser beiden Mengen werden als *trivial* bezeichnet. Wir zeigen nun, daß alle anderen Mengen Prog(.) unentscheidbar sind!

[1] Sie wissen schon, die Aussage $(A \Rightarrow B)$ ist äquivalent zu $(\neg B \Rightarrow \neg A)$

[2] Die Semantik eines Programms ist seine Bedeutung; für jemanden ohne Wissen über Turingmaschinen ist die Gödelnummer einer Turingmaschine lediglich eine nichtssagende 0-1-Folge.

1.22 Satz: (Rice, 1953)
Sei S, $S \subseteq \mathcal{R}$, mit $\emptyset \neq S \neq \mathcal{R}$. Dann ist $\mathrm{Prog}(S)$ nicht entscheidbar.

Beweis:
Sei u die für keine Eingabe definierte Funktion. u ist berechenbar im Sinne der Definition 1.18, z. B. durch die (ganz konkrete) Turingmaschine M_u, die sofort in eine Endlosschleife geht. Mit anderen Worten: $u \in \mathcal{R}$ und $\langle M_u \rangle \in \mathrm{Prog}(\mathcal{R})$. Beachten Sie, daß schon $\mathrm{Prog}(\{u\})$ unendlich viele Programme enthält.

Entweder ist $u \notin S$ („Fall (a)“) oder $u \in S$ („Fall (b)“).

Fall (a): $u \notin S$ und damit auch $\langle M_u \rangle \notin \mathrm{Prog}(S)$. Wir zeigen: $H_\varepsilon \leq \mathrm{Prog}(S)$.

Da $S \neq \emptyset$, gibt[3] es eine Funktion $s \in S$, die durch eine (wieder ganz konkrete) (1-Band-)Turingmaschine M_s mit $\langle M_s \rangle \in \mathrm{Prog}(S)$ berechnet werden kann. Nun schreiben wir ein neues Programm:

Drumherum$_{\langle M \rangle}$

die Eingabe sei y
starte M mit leerem Band auf einem Hilfsband
starte M_s mit y

Als Reduktionsfunktion nehmen wir

$$f(x) = \begin{cases} \langle \text{Drumherum}_{\langle M \rangle} \rangle & \text{falls } x = \langle M \rangle \\ \langle M_u \rangle & \text{falls } x \neq \langle M \rangle \quad \text{(der „sonst“-Fall)} \end{cases}$$

f ist total berechenbar, eine Eigenschaft, die eine Reduktionsfunktion zu erfüllen hat, und über die man zumindest nachzudenken hat. Nun gilt:

$$\begin{aligned} x \in H_\varepsilon \quad &\Rightarrow \quad x = \langle M \rangle \text{ und } M \text{, gestartet mit } \varepsilon \text{, hält} \\ &\Rightarrow \quad f(x) = \langle \text{Drumherum}_{\langle M \rangle} \rangle \text{ und Drumherum}_{\langle M \rangle} \text{ berechnet } s \in S \\ &\Rightarrow \quad f(x) \in \mathrm{Prog}(S) \end{aligned}$$

$$x \notin H_\varepsilon \quad \Rightarrow \quad \begin{cases} x = \langle M \rangle \text{ und } M \text{, gestartet mit } \varepsilon \text{, hält nicht} \\ \qquad \Rightarrow f(x) = \langle \text{Drumherum}_{\langle M \rangle} \rangle \text{ und Drumherum}_{\langle M \rangle} \text{ berechnet } u \notin S \\ \qquad \Rightarrow f(x) \notin \mathrm{Prog}(S) \\ x \text{ nicht von der Form } \langle M \rangle \Rightarrow f(x) = \langle M_u \rangle \notin \mathrm{Prog}(S) \end{cases}$$

Fall (b): $u \in S$. Wir zeigen diesmal: $\overline{H_\varepsilon} \leq \mathrm{Prog}(S)$.

Interessanterweise können wir die Reduktionsfunktion aus Fall (a) ganz übernehmen!

[3] An dieser Stelle wird der Beweis *nichtkonstruktiv*. Niemand berechnet für uns dieses s, es fällt beinahe vom Himmel.

Da $S \neq \mathcal{R}$, gibt es eine Funktion $s \in \mathcal{R} \setminus S$, die durch eine (ganz konkrete) (1-Band-)Turingmaschine M_s mit $\langle M_s \rangle \notin \text{Prog}(S)$ berechnet werden kann.
Als Reduktionsfunktion nehmen wir wieder

$$f(x) = \begin{cases} \langle \text{Drumherum}_{\langle M \rangle} \rangle & \text{falls } x = \langle M \rangle \\ \langle M_u \rangle & \text{falls } x \neq \langle M \rangle \end{cases}$$

Mit dieser total berechenbaren Funktion f zeigen Sie bitte nun als Fingerübung einmal selbst: $x \in \overline{H_\varepsilon} \iff f(x) \in \text{Prog}(S)$ □

Zum besseren Verständnis des Beweis dieses Satzes sollten Sie sich noch einmal das Spiegelei-Bild (Abbildung 1.1) vornehmen und ganz konkret die beiden Fälle des Beweises untersuchen, indem Sie die vorkommenden Turingmaschinen einzeichnen.

Der Satz von Rice hat sehr weitreichende Konsequenzen. In einem konkreten Beispiel angewandt besagt er folgendes: Angenommen, es geht um die Menge *aller* Navigationsprogramme, die in Landkarten den kürzesten Weg zwischen zwei Orten berechnen. Dann gibt es kein Korrektheitsüberprüfungsprogramm, das für jedes mögliche Navigationsprogramm dessen Korrektheit überprüft. Damit wird der schwarze Peter an Sie zurückgegeben: Wenn Sie ein Navigationsprogramm schreiben, müssen Sie das selbst so systematisch machen, daß Sie Ihrem Kunden im Zweifelsfall die Korrektheit Ihres *einen, ganz konkreten* Programms beweisen können.

Machen Sie sich mit der genauen Bedeutung des Satzes von Rice vertraut. Newcomer wenden ihn häufig zu großzügig an. Insbesondere, daß es in ihm um *alle* Programme geht, die die Funktionen in S berechnen, wird leider allzu gern übersehen. Schränkt man die Programme (nicht die Funktionen!), um die es geht, ein, kann das Problem durchaus entscheidbar werden.

Einige Bemerkungen zu den Reduktionen: Wir haben in den obigen Ausführungen immer zwei Richtungen gezeigt, nämlich $(x \in L_1 \Rightarrow f(x) \in L_2)$ und $(x \notin L_1 \Rightarrow f(x) \notin L_2)$. Newcomer schreiben gerne die erste Richtung hin und machen dann einfach aus den „$\Rightarrow$"-Zeichen „$\Leftrightarrow$"-Zeichen. Wenn aber die ausgedachte „Reduktionsfunktion" nicht korrekt ist, kommt der Fehler leider meist in der „$\Leftarrow$"-Richtung vor und bleibt somit unentdeckt. Um auf der sicheren Seite zu sein, sollte man in der Anfangszeit immer beide Richtungen getrennt beweisen.

Ein Hindernis für Newcomer beim Verstehen des Reduktionskonzepts ist die verwirrende(?!?) Vielzahl der vorkommenden Turingmaschinen(-Programme): Es gibt drei Ebenen, auf denen welche auftauchen:

- Ebene 1: „Könnte man L_2 entscheiden, dann auch L_1." Hier wird eine Maschine für L_1 angegeben. Sie taucht im Beweis von Satz 1.20 auf.
- Ebene 2: „Man kann L_1 auf L_2 mittels f reduzieren." Hier wird eine Maschine für die Berechnung der Reduktionsfunktion f programmiert.
- Ebene 3: „$x \in L_1 \iff f(x) \in L_2$." Hier sind die x und $f(x)$ ganz häufig Maschinen.

Hier hilft leider nur die intensive Beschäftigung mit diesen drei Ebenen, um Klarheit in und Vertrautheit mit diesem Gebiet zu bekommen.

1.8 Rekursive Aufzählbarkeit

Bislang wurde die Mengeneigenschaft *rekursiv aufzählbar* einfach nur als abstrakte Bezeichnung benutzt. Im folgenden werden wir wenigstens den Bestandteil *aufzählbar* erklären.

1.23 Satz:
Sei L eine unendliche Sprache. Dann gilt:
L ist rekursiv aufzählbar $\iff$ *es gibt eine total berechenbare surjektive Funktion* $g : \{0,1\}^* \to L$

Beweis:
„$\Leftarrow$“:
Sei $g : \{0,1\}^* \to L$ eine total berechenbare surjektive Funktion, die von der 1-Band-Turingmaschine M_g berechnet werden möge, und die wir sozusagen wieder in einer Programmlibrary zur Verfügung gestellt bekommen haben. Ziel ist es, eine Turingmaschine M zu programmieren, die, gestartet mit $x \in \{0,1\}^*$, für genau die $x \in L$ hält und dabei M_g als Unterprogramm benutzt.

M arbeitet wie folgt: Sie bekommt ein beliebiges $x \in \{0,1\}^*$ als Eingabe auf Band 1 und sucht nun ein $y \in \{0,1\}^*$ mit $g(y) = x$. Das macht sie, indem sie systematisch nacheinander auf Band 2 die möglichen y-Werte ε, 0, 1, 00, 01, 10, 11, 000, ... erzeugt. Dieser mögliche y-Wert wird auf Band 3 kopiert. Auf Band 3 wird M_g nun mit y gestartet. Da g total ist, hält M_g für jede Eingabe. Die Ausgabe von M_g, gestartet mit y, wird nun mit der Eingabe x auf Band 1 verglichen. Bei Gleichheit wird gestoppt, bei Ungleichheit wird der nächste y-Wert ausprobiert. Da g surjektiv ist, gibt es für jedes $x \in L$ ein y mit $g(y) = x$, d. h. ist die Eingabe in L, wird ein solches y auch gefunden. Anderenfalls versandet M in einer Endlosschleife.
Auf einem höheren Level läßt sich M algorithmisch so beschreiben:

> Die Turingmaschine M:
>
> die Eingabe sei x;
> $y := \varepsilon$;
> **while** M_g, gestartet mit y, nicht x ausgegeben hat **do** $y := \text{next}(y)$.

Insgesamt haben wir: M, gestartet mit x, hält $\iff x \in L$.

„$\Rightarrow$“:
Da L rekursiv aufzählbar ist, gibt es eine Turingmaschine M, die auf genau den Eingaben $x \in L$ hält. Wir müssen nun eine Turingmaschine M_g programmieren, die

(i) für *jede* Eingabe hält,
(ii) nur Wörter aus L ausgibt und
(iii) jedes Wort aus L ausgeben kann.

Die von M_g berechnete Funktion ist dann die gesuchte Funktion g.

Was bedeutet es u. a., daß M, gestartet mit x, hält? Es gibt ein $t \in \mathbb{N}$, so daß M, gestartet mit x, nach t Schritten stoppt. D. h. als Eingabe y für M_g werden beliebige Paare (x,t) (die müssen

geeignet kodiert werden!) verwendet. M_g führt dann t Schritte von M, gestartet mit x, aus. Wenn M dabei eine Endkonfiguration erreicht, wird x ausgegeben, ansonsten ein festes $x_{\text{fix}} \in L$.
Da M_g ja ein $y \in \{0,1\}^*$ als Eingabe bekommt, müssen wir uns nun überlegen, wie wir daraus ein Paar $(x,t) \in \{0,1\}^* \times \mathbb{N}$ machen. Das geht z. B. wie folgt:

$$\text{aus\_eins\_mach\_zwei}(y) = \begin{cases} (a_1 \ldots a_n, t) & \text{falls } y = a_1 \ldots a_n 0 \underbrace{1 \ldots 1}_{t \text{ viele}} \\ (0,1) & \text{sonst} \end{cases}$$

aus_eins_mach_zwei ist total berechenbar und surjektiv. Instruktive Beispiele sind:

$$\begin{aligned} &\text{aus\_eins\_mach\_zwei}(011010111) = (01101,3), \\ &\text{aus\_eins\_mach\_zwei}(011) = (\varepsilon,2), \\ &\text{aus\_eins\_mach\_zwei}(1111) = (0,1), \\ &\text{aus\_eins\_mach\_zwei}(0110) = (011,0) \end{aligned}$$

Man sieht, wir können sagen, daß wir in y die rechteste 0 als Komma interpretieren (und den Fall abfangen, daß y gar keine 0 enthält).
Nun ist es ein leichtes, das Programm für M_g anzugeben.

Die Turingmaschine M_g:

die Eingabe sei y;
$(x,t) :=$ aus_eins_mach_zwei(y);
simuliere auf einem Extra-Band t Schritte von M, gestartet mit x;
falls M eine Endkonfiguration erreicht hat, gib x aus, sonst x_{fix}

Ebenso leicht ist es, die Punkte (i) bis (iii) zu überprüfen. □

Eine interessante Programmieraufgabe ist es, die Konstruktion aus dem gerade geführten Beweis zu benutzen, um die Funktion g sogar bijektiv zu machen, d. h. ein Verfahren anzugeben, das für jede Eingabe y ein anderes Wort $x \in L$ ausgibt.

Insgesamt können wir nun ein Programm schreiben, das nacheinander $g(\varepsilon)$, $g(0)$, $g(1)$, $g(00)$, $g(01)$, $g(10)$, $g(11)$, $g(000)$, ... hinschreibt, also tatsächlich L aufzählt! Überlegen Sie sich, was das bedeutet: Wir wissen z. B., daß das initiale Halteproblem H_ε rekursiv aufzählbar ist. D. h. jetzt also, daß es ein Verfahren gibt, das nacheinander alle Programme $\langle M \rangle$ aufschreibt, die, gestartet mit dem leeren Band, halten.

Beachten Sie, daß es im allgemeinen bei der Reihenfolge der Ausgabe keine „schöne“ Ordnung geben kann, wenn man eine rekursiv aufzählbare Sprache L mit einer wie oben konstruierten Funktion g aufzählt, da L ansonsten bereits entscheidbar wäre.

Zum Abschluß dieses Kapitels der Vorlesung wollen wir noch ein paar nicht offensichtliche Eigenschaften rekursiv aufzählbarer Sprachen untersuchen.
Dazu führen wir ein paar Sprechweisen ein:

- Eine Menge $\mathcal{L}$ von Sprachen bezeichnet man als *Sprachklasse* oder auch als *Sprachfamilie*. Z. B. ist $\mathcal{E} = \{L \mid L \text{ ist entscheidbar}\}$ die Klasse der enscheidbaren Sprachen, und $\mathcal{L}_0 = \{L \mid L \text{ ist rekursiv aufzählbar}\}$ die Klasse der rekursiv aufzählbaren Sprachen[4].

- Wenn wir eine k-stellige Operation $\mathrm{op}(.,\ldots,.)$ auf Sprachen haben, also eine Operation, die aus k Sprachen eine neue Sprache macht, dann ist eine Sprachklasse $\mathcal{L}$ genau dann *gegen* op *abgeschlossen* (eine alternative Sprechweise ist: *unter* op *abgeschlossen*), wenn gilt: $L_1,\ldots,L_k \in \mathcal{L} \Rightarrow \mathrm{op}(L_1,\ldots,L_k) \in \mathcal{L}$

  Z. B. kann man sich ganz einfach überlegen, daß die entscheidbaren Sprachen gegen die Vereinigung abgeschlossen sind: $L_1, L_2 \in \mathcal{E} \Rightarrow L_1 \cup L_2 \in \mathcal{E}$. Auch wissen wir aus Satz 1.15, daß die rekursiv aufzählbaren Sprache nicht gegen Komplementbildung abgeschlossen sind, da $H \in \mathcal{L}_0 \Rightarrow \overline{H} \notin \mathcal{L}_0$.

Wir hatten gerade erwähnt, daß es einfach ist zu zeigen, daß die entscheidbaren Sprachen gegen die Vereinigung abgeschlossen sind. Ein Beweis besteht darin, für eine Eingabe x *zuerst* zu überprüfen, ob $x \in L_1$ ist. Wenn nicht, *dann* wird $x \in L_2$ überprüft. Dieses Nacheinanderüberprüfen ist möglich, da die entscheidenden Turingmaschinen ja immer halten.

Eine Schwierigkeit beim Beweis von Abschlußeigenschaften für rekursiv aufzählbare Sprachen besteht darin, daß man dieses Nacheinander nicht machen kann. Stellen Sie sich vor, $x \notin L_1$ und $x \in L_2$. Würde zuerst überprüft, ob x in L_1 ist, würde das Verfahren in eine Endlosschleife laufen, obwohl $x \in L_1 \cup L_2$ ist.

Zum Ziel führen hier zwei mögliche Programmiertechniken:

Zum einen könnte man beide Maschinen gleichzeitig (anderes Wort, das oft benutzt wird: parallel) auf zwei Bändern mit jeweils der Eingabe x laufen lassen. Sobald eine der beiden hält, hält die Maschine für die Vereinigung.

Zum anderen könnte man wie ein Ein-Prozessor-Computer im Multitasking-Betrieb arbeiten: Jede Maschine bekommt ein Zeitfenster, während dessen sie rechnen darf, danach wird die Konfiguration gespeichert, und nun ist die nächste Maschine dran, deren abgespeicherte Konfiguration geladen wird, so daß deren Rechnung für die Dauer des Zeitfensters weitergeführt werden kann. Die Kombination der Maschinen läßt man also kontrolliert laufen. Damit nicht eine Maschine endlos läuft, läßt man jede „ein bißchen“ laufen. Implizit ist dieser Trick auch im Beweis der Richtung „$\Rightarrow$“ von Satz 1.23 zu finden, nämlich in Form der Zeitschranke t.

1.24 Satz:
Seien L_1 und L_2 rekursiv aufzählbar. Dann gilt:

(i) $L_1 \cup L_2$ *ist rekursiv aufzählbar.*

(ii) $L_1 \cap L_2$ *ist rekursiv aufzählbar.*

[4] Warum sie mit $\mathcal{L}_0$ bezeichnet wird, werden wir später noch entdecken.

Beweis:
Nach der Vorstellung des Parallellaufens und des Zeitscheibentricks ist nun klar, wie die Maschinen arbeiten müssen.

Für die Vereinigung reicht es, wenn eine der beiden Maschinen hält, beim Durchschnitt müssen beide halten. □

Im Beweis von Satz 1.15 hatten wir durch Anwendung der parallelen Ausführung von Programmen für H und $\overline{H}$ gezeigt, daß $\overline{H}$ nicht rekursiv aufzählbar ist. Der gleiche Beweis geht natürlich für alle rekursiv aufzählbaren Sprachen.

1.25 Satz:
L ist entscheidbar $\iff$ L und $\overline{L}$ sind rekursiv aufzählbar.

Um für die rekursiv aufzählbaren Sprachen zu zeigen, daß sie gegen kompliziertere Operationen wie die Konkatenation „∘“ (auch Produkt genannt und definiert durch: $L_1 \circ L_2 := \{w \mid \exists u \in L_1 \exists v \in L_2\colon w = uv\}$) abgeschlossen sind, ist der Programmieraufwand zwar größer, aber die beiden genannten Tricks funktionieren noch immer.

Nachdem wir uns bislang mit dem Grenzbereich dessen, was man überhaupt mit Computern berechnen kann (und was nicht!), beschäftigt haben, wollen wir uns nun mit dem auseinandersetzen, was man mit Computern schnell berechnen kann (und was vermutlich leider nicht). Erstaunlicherweise sind auch dabei Turingmaschinen und Reduktionen extrem nützlich. Bei den Reduktionen werden wir im folgenden auch auf die Zeit achten, die man braucht, um die Reduktionsfunktion auszurechnen.