Leben in der rekursiven Welt
Selbstreproduzierende Automaten und Programme

Andreas Schwill

Eines der zentralen Elemente biologischer Lebensformen und zugleich notwendige Bedingung für Weiterentwicklung und Anpassung an geänderte Umweltbedingungen ist ihre Fähigkeit, sich zu reproduzieren und Nachkommen zu erzeugen, die vermöge Mutation und Auslese besser geeignet sind, sich in der Umwelt zu behaupten, als ihre Eltern und damit das Überleben und die Ausbreitung der Art sichern.
Versucht man Leben durch Maschinen nachzubilden oder auf dem Computer zu simulieren, so ist Selbstreproduktion eine der wichtigsten Fähigkeiten, die man in der Maschine anlegen muß. Wir wollen uns in diesem Artikel einige praktische und theoretische Aspekte der Selbstreproduktion überlegen.

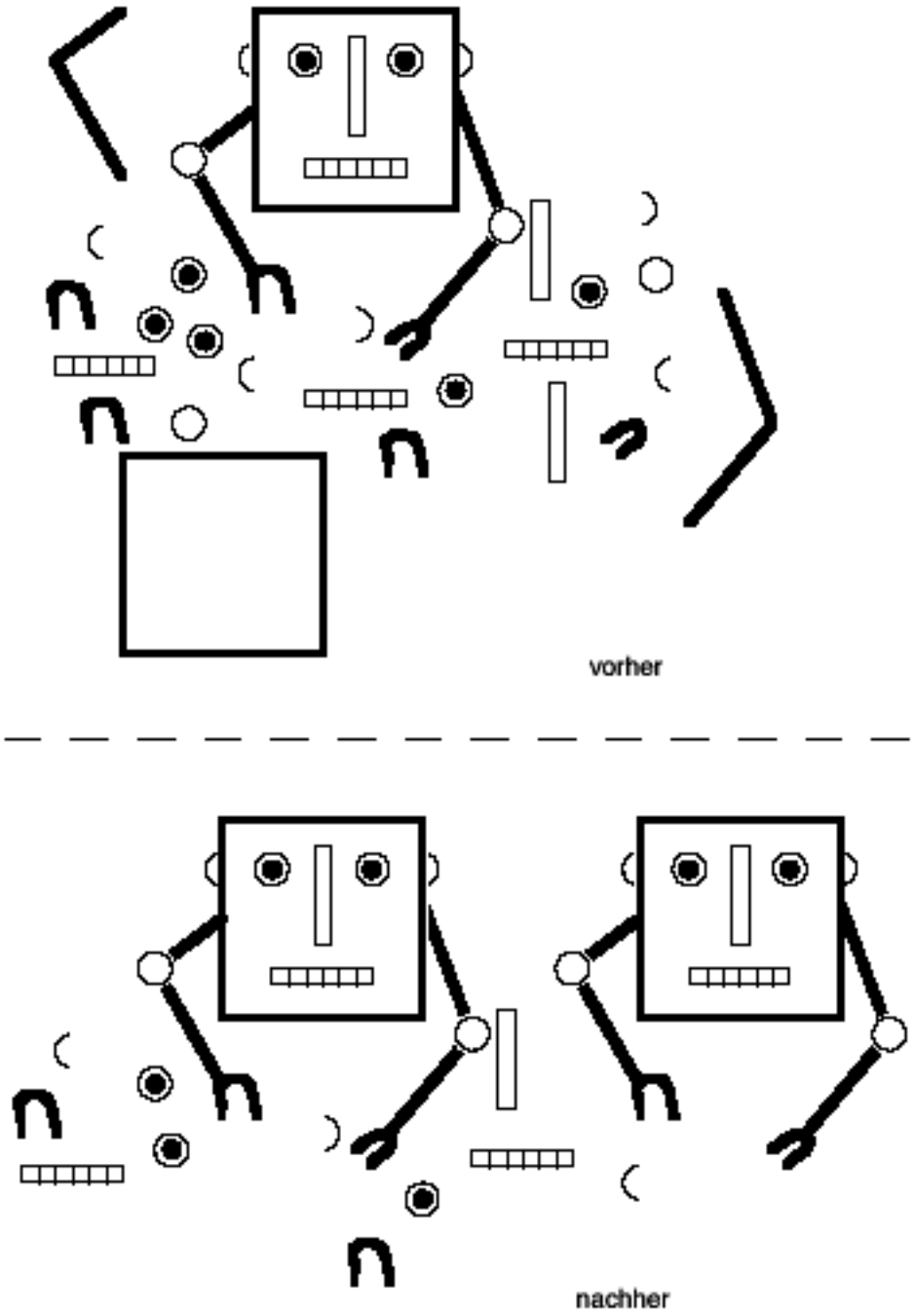

Abb. 1: Prinzip der Selbstreproduktion

Abb. 2: John von Neumann

Selbstreproduzierende Automaten

Kann man einen Automaten herstellen, der bei Zuführung einer hinreichend großen Menge von Rohstoffen eine exakte Kopie von sich selbst anfertigt (Abb. 1)? Diese Frage hat sich John von Neumann (1903-1957) (Abb. 2), einer der Pioniere der Informatik, bereits Mitte des letzten Jahrhunderts gestellt [1]. Seine Motivation war vor allem, biologische Prozesse nachzubilden und etwas über die prinzipiellen Möglichkeiten von Maschinen, die zu seiner Zeit noch sehr beschränkt waren, herauszubekommen.

Versetzen wir uns in seine Zeit und versuchen, seine Gedankengänge nachzuvollziehen. Auf der Grundlage der damals noch recht beschränkten Fähigkeiten von Maschinen kann man rasch zu dem Schluß kommen, selbstreproduzierende Automaten könne es nicht geben. Denn damals wie heute kennt man im wesentlichen nur die Situation, daß eine produzierende Maschine stets (wesentlich) komplexer sein muß als das produzierte Produkt. So benötigt man, um eine einfache Pinwand-Nadel herzustellen, schon eine kleinere Fertigungsstraße; um diese Fertigungsstrecke herzustellen, ist schon eine Reihe ganz unterschiedlicher Automaten erforderlich, für die man selbst dann eine ganze Fabrik zur Herstellung benötigt. Um diese Fabrik wiederum aufzubauen, ist man auf eine ganze Reihe von Fabriken angewiesen, angefangen von einer Betonfabrik bis hin zu einer Fabrik für die Knöpfe an den Kitteln der Arbeiter (Abb. 3). Und am Ende dieser Kette steht immer der Mensch als das komplexeste Individuum, das die Fabriken entwickelt und angefertigt hat.

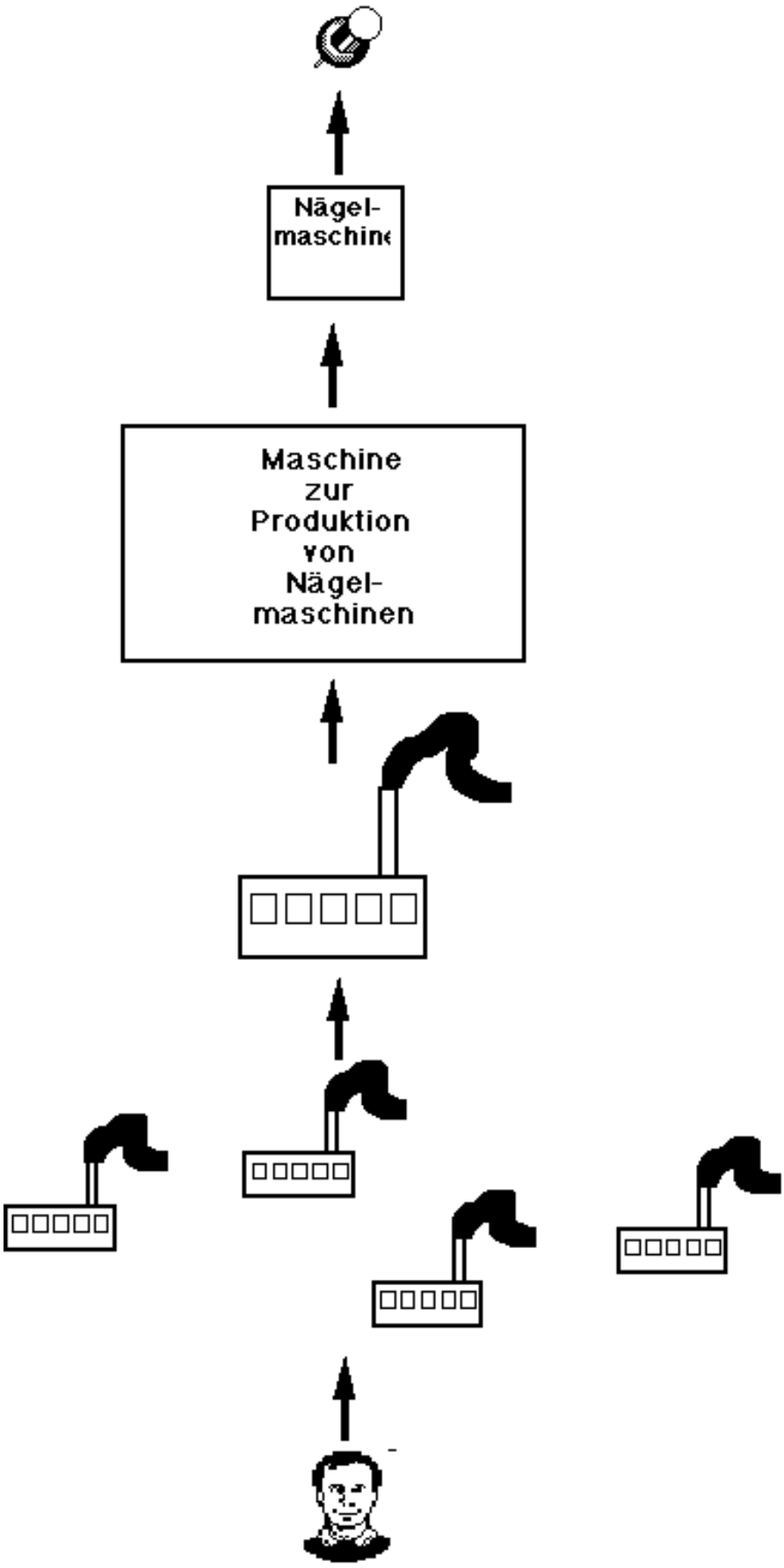
Nägel-
maschine
Maschine
zur
Produktion
von
Nägel-
maschinen

Abb. 3: Komplexität von Maschinen aus der Erfahrungswelt

Läßt man sich von diesen Erfahrungen leiten, fühlt man sich bestätigt, daß selbstreproduzierende Maschinen nicht möglich sind. Warum funktioniert es dann aber bei biologischen Systemen?

Tatsächlich ist diese Vermutung falsch. Hintergrund der Fehleinschätzung ist vermutlich die Tatsache, daß man in der Lebenswelt nur mit relativ einfachen Maschinen konfrontiert ist und nur eine sehr ungenaue Vorstellung über die prinzipiellen Fähigkeiten *sehr* komplexer Automaten entwickeln kann. Diese letzteren vorstellbaren Maschinen sind dann tatsächlich in der Lage, sich selbst zu reproduzieren.

Wir wollen im folgenden die logischen Schlüsse von Neumanns nachvollziehen. Zunächst ganz allgemein: Gesucht ist eine Maschine M, die eine andere Maschine M' herstellen kann. M benötigt zur Produktion von M' offenbar:

- einen Bauplan von M', der exakt die Bauteile und ihr Zusammenwirken beschreibt,
- einen großen Vorrat an Bauteilen (z.B. Schrauben, Räder, Zahnräder, Bleche, Magnetbänder zum Speichern usw.)
- Greifarme zur Montage der Bauteile etc.
- eine Steuereinheit, die den Bauplan interpretiert und in Steuerbefehle für die Greifarme etc. umsetzt.

Von Neumanns Idee bestand darin, drei unabhängige Maschinen (gedanklich) zu konstruieren, die geeignet zusammengeschaltet werden und insgesamt M ergeben. Die Maschinen sind

- ein Universalkonstruktor U,
- ein Magnetbandkopierer D (zu von Neumanns Zeiten verwendete man noch Magnetbänder zur Speicherung) und
- ein Kontrollautomat C.

Die Maschinen im einzelnen:

- Der Universalkonstruktor U erhält als Eingabe einen auf Magnetband gespeicherten Bauplan b(M) einer Maschine M und eine Menge von Bauteilen und produziert die Maschine M (Abb. 4). U entspricht heute etwa einer automatischen Fabrik.
- Der Magnetbandkopierer D erhält ein beschriebenes und ein leeres Magnetband und kopiert das beschriebene auf das leere. Anschließend verfügt man über zwei identische Magnetbänder (Abb. 5).
- Der Kontrollautomat C überwacht die Arbeitweise von U und D; er kann U und D zu beliebigen Zeiten stoppen und mit Magnetbändern versorgen.

Nun baut man die drei Maschinen U, D und C zusammen und erhält eine sehr komplexe Maschine R (Abb. 7). R arbeitet wie folgt:

1. R erhält einen Bauplan b(M) einer Maschine M auf Magnetband.
2. C spannt das Band in den Kopierer D ein. D kopiert das Band; Ergebnis: zwei identische Kopien von b(M).
3. C gibt eine der beiden Kopien an U. U fertigt aus b(M) die Maschine M.
4. C gibt der Maschine M das Band b(M) mit auf den Weg und liefert M und b(M) aus.

Insgesamt hat R aus b(M) also M und eine Kopie von b(M) produziert. Schreibweise:

$$b(M) \rightarrow_R M+b(M).$$

Diese Konstruktion ist insoweit allgemeingültig, als mit ihr keine Annahmen oder Einschränkungen an die Maschine M verbunden sind. Man kann also – und hierin liegt der rekursive Gedankensprung – ohne weiteres M=R setzen. Dann fertigt R aus einem Bauplan b(R) sich selbst mit einer weiteren Beschreibung b(R) an, also

$$b(R) \rightarrow_R R+b(R).$$

R ist genau die gesuchte selbstreproduzierende Maschine.

Hierbei ist noch zu bemerken, daß M nur aus endlich vielen Bauteilen bestehen kann, die nur auf endlich viele verschiedene Arten kombiniert werden können, so daß Baupläne b(M) stets nur endlich lang sind.

Gegenwärtig wäre die Menschheit beinahe in der Lage, eine selbstreproduzierende Maschine nach diesem Vorbild herzustellen: sie bestände in etwa aus der Zusammenfassung und elektronischen Vernetzung aller automatischen Fabriken der Welt, eine gigantische, aber durchaus vorstellbare Maschine, deren komplexestes Element wohl der Universalkonstruktor ist.

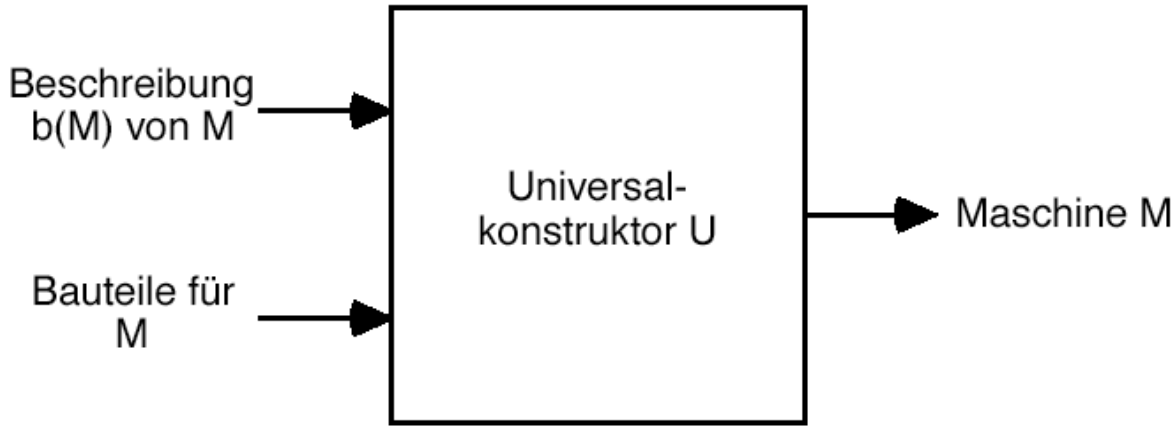

Abb. 4: Universalkonstruktor U

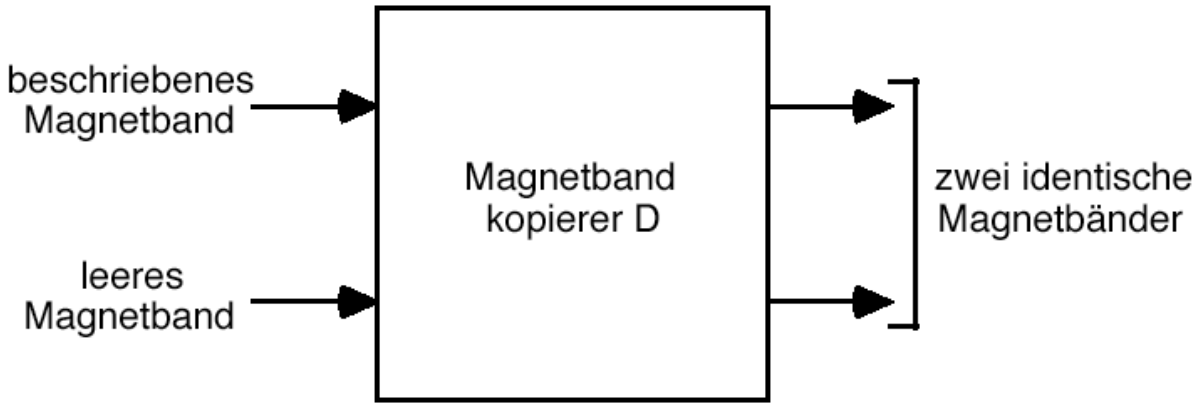

Abb. 5: Magnetbandkopierer D

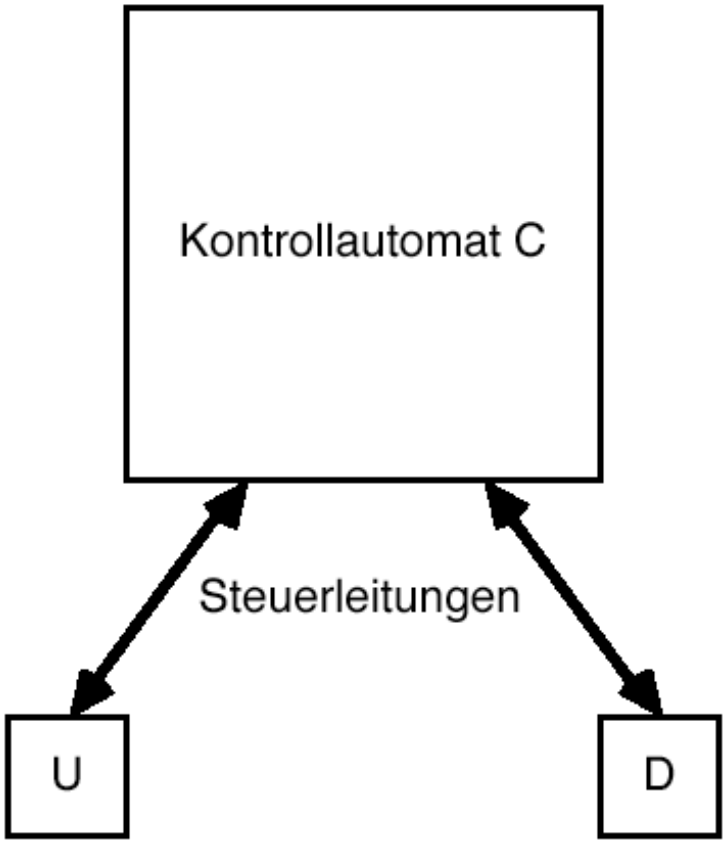

Abb. 6: Kontrollautomat C

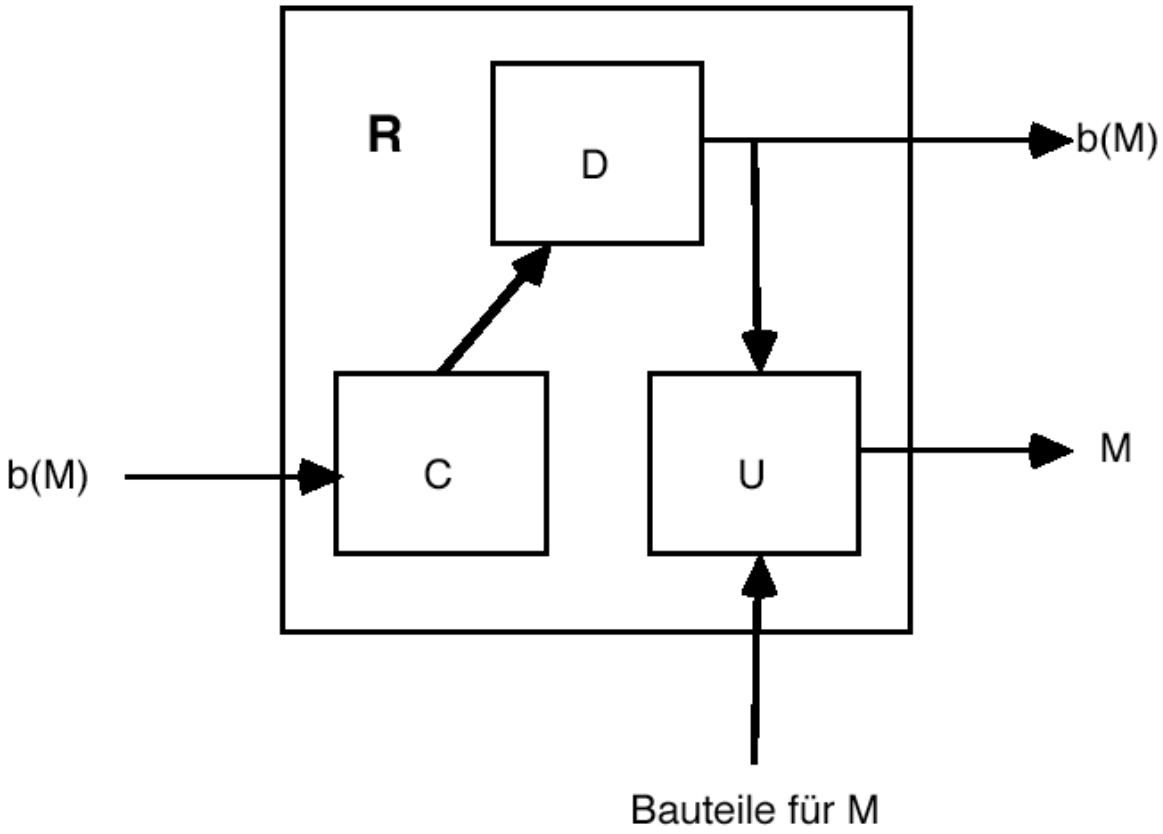

Abb. 7: Maschine M

Selbstreproduzierende Programme

In der Informatik sind wir weniger daran interessiert, konkrete Maschinen zu erstellen, vielmehr begnügen wir uns mit den Beschreibungen von Handlungsfolgen (Programmen), von denen wir wissen, daß eine Maschine sie ausführen kann, und identifizieren Programme und Maschinen. Folglich befassen wir uns in diesem Abschnitt mit der Entwicklung von selbstreproduzierenden Programmen. Ein Programm heißt *selbstreproduzierend*, wenn es keine Eingabe besitzt und bei Ablauf seinen eigenen Programmtext ausgibt. Auch hier wieder die Paradoxie: Wenn ein Programm w ein programm w' ausgeben soll, muß doch in w mehr Information und Komplexität stecken als in w' ...

Die Fähigkeit zur Selbstreproduktion benötigen z.B. Computerviren, die bei Ablauf möglichst viele Kopien von sich anfertigen und über das Internet verbreiten sollen.

Versuchen wir einen einfachen Ansatz, ein selbstreproduzierendes Programm in PASCAL zu schreiben: Man beginnt z.B. mit

```
(1)   program repro(output);
(2)   begin
```

und ergänzt nacheinander Anweisungen zur Ausgabe des Programmtextes:

(3) writeln(‘program repro(output);’);

um die erste Zeile auszugeben,

(4) writeln(‘begin’);

um die zweite Zeile auszugeben. Um die dritte und vierte Zeile auszugeben, muß man die auszugebenden Hochkommas verdoppeln:

(5) writeln(‘writeln(’’program repro(output);’’);’);

(6) writeln(‘writeln(’’begin’’);’);

Um fünfte und sechste Zeile auszugeben, sind die auszugebenden Hochkommas erneut zu verdoppeln:

(7) writeln(‘writeln(’’writeln(’’’’program repro(output);’’’’);’’);’);

(8) writeln(‘writeln(’’writeln(’’’’begin’’’’);’’);’)

…

end.

Die fortlaufende Verdopplung der Hochkommas ist eines der Probleme, die bei diesem einfachen Ansatz zu lösen sind. Eine Möglichkeit besteht darin, statt des Hochkommas die ASCII-Codierung zu verwenden: Das Hochkomma besitzt die ASCII-Codierung 39, so daß man stattdessen überall chr(39) schreiben kann.

Ein schwierigeres Problem folgt aus der Beobachtung, daß man nach dem obigen Schema an jeder Stelle immer nur Programmtext ausgibt, der vor der Zeile steht, die gerade ausgeführt wird. Konkret gibt man in obigem Programm in Zeile n die Programmzeile n-2 aus. Am Schluß des Programms unmittelbar vor dem end muß jedoch bereits der gesamte Programmtext (incl. end) ausgegeben sein. Folglich muß der ausgegebene Text irgendwann den gerade bearbeiteten Text „überholen“. Nach dem obigen Ansatz ist das nicht möglich, denn in jeder Zeile wird ein Stück Programmtext ausgeführt, das länger ist als der jeweils ausgegebene Text. Es muß aber irgendwann gerade umgekehrt sein: Man muß mit einem kleinen Stück Programmtext ein großes Stück Programmtext ausgeben, damit die Ausgabe die Verarbeitung überholt. Dieses „Überhol-Problem“ kann man z.B. auf zwei Arten angehen, durch Textkonstanten oder durch Prozeduren bzw. Funktionen. Hat man eine Textkonstante t oder eine geeignete Prozedur p definiert, so reicht ein kurzes Stück Programmtext write(t) oder p(...) aus, um ein langes Stück Programmtext auszugeben. Ob dies zur Entwicklung eines selbstreproduzierenden Programms reicht, werden wir später sehen. Zunächst beschäftigen wir uns mit einigen theoretischen Überlegungen.

Theorie selbstreproduzierender Programme

Noch bevor man sich der nicht ganz leichten Ausgabe widmet, ein selbstreproduzierendes Programm zu konstruieren, sollte man mit mathematischen Mitteln versuchen nachzuweisen, ob solch ein Programm überhaupt existiert oder nicht.
Das geeignete Mittel hierfür ist das vielfältig verwendbare Rekursionstheorem von Stephen C. Kleene (1909-1994) aus der Theorie berechenbarer Funktionen. Das Theorem ermöglicht mit relativ wenig Aufwand die unmittelbare Entscheidung darüber, ob ein Problem/eine Funktion berechenbar ist, also durch ein Programm gelöst werden kann, oder nicht. Das Rekursionstheorem, von dem es zwei Versionen gibt, stellt anschaulich eine Beziehung her zwischen den Eingaben eines beliebigen Programms und den von diesen Eingaben berechneten Funktionen, wenn man die Eingaben als Texte betrachtet, die selbst wieder als Programme interpretiert werden können.
Einige wenige grundlegende Begriffe genügen zur formalen Definition des Rekursionstheorems. Sei A ein beliebiges *Alphabet*, z.B. die Zeichen des ASCII-Codes, und A* die Menge aller endlichen Texte, die man mit Zeichen aus A bilden kann. Dann bezeichnen wir mit

- $P^i=\{f: (A^*)^i \rightarrow A^* \mid f \text{ ist berechenbar}\}$ die Menge aller berechenbaren Funktionen mit i Eingabewerten und einem Ausgabewert, die jeweils Texte über A sind. Programme, die diese Funktionen berechnen, haben also in PASCAL etwa folgende Gestalt:

  var $w_1,w_2,...,w_i,y$: text;
  read$(w_1,w_2,...,w_i)$; ...; write(y).

  P^0 ist dann die Menge aller konstanten Funktionen (keine Eingaben) mit Werten in A*; man kann P^0 mit A* identifizieren.
- Für $f \in P^{i+1}$ sei $f_u \in P^i$ definiert durch $f_u(w_1,w_2,...,w_i)=f(u,w_1,w_2,...,w_i)$. f_u bezeichnet die Funktion, die der Text u, als Programm interpretiert, berechnet. Ist u kein gültiger Programmtext, so ist f_u undefiniert.

  Beispiel: Ist u der Programmtext für ein Primzahlprogramm, so ist

  $f_u: A^* \rightarrow A^*$

  die von u berechnete Funktion mit

  $$f_u(x) = \begin{cases} \text{'ja', falls x eine Primzahl darstellt,} \\ \text{'nein', sonst} \end{cases}$$
- $e \in P^{i+1}$ sei eine *universelle Funktion* für P^i, d.h., eine Funktion, die beliebige andere Funktionen simulieren kann und sich bei Eingabe eines Programms u und von Eingabewerten $w_1,w_2,...,w_i$ für u so verhält, wie es u mit den Eingabewerten tun würde (Abb. 8). Ein Computer zusammen mit seinem Betriebssystem, dem Übersetzer und dem Laufzeitsystem für PASCAL realisiert solch eine universelle Funktion e. Statt ein

Programm u direkt auf dem Computer zu implementieren, übergibt man es mit den Eingabewerten an e, und e simuliert es und verhält sich so wie u.

Nun kann man auch formal definieren:

Definition

Ein Programm $u \in A^*$ heißt *selbstreproduzierend*, wenn gilt:

- u berechnet eine Funktion aus $P^0=A^*$, also eine konstante Funktion,
- $e(u)=e_u=u$, d.h. u gibt sich selbst aus.

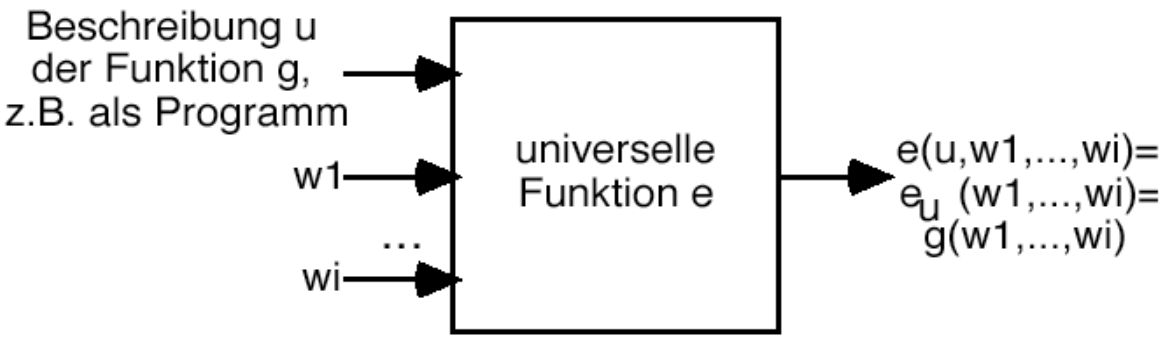

Abb. 8: Wirkungsweise einer universellen Funktion

Das Rekursionstheorem (Version 1) formuliert nun folgende Aussage.

Satz (Rekursionstheorem, Version 1)

Zu jedem $i \geq 0$ und zu jeder berechenbaren Funktion $f \in P^{i+1}$ gibt es ein Wort $w \in A^*$, so daß für alle $w_1,w_2,...,w_i \in A^*$ gilt:

$$f(w,w_1,w_2,...,w_i)=e_w(w_1,w_2,...,w_i).$$

Nehmen wir i=1, dann gibt es also zu allen berechenbaren Funktionen f eine erste Eingabe w, die man als Programmtext interpretieren kann, so daß sich f auf w und jeder beliebigen zweiten Eingabe w_1 genauso verhält, wie das Programm w nur auf die Eingabe w_1:

$$f(w,w_1)=e_w(w_1).$$

Beispiel: f sei die Funktion, die aus dem Text w der ersten Eingabe das w_1-te Zeichen liefert, sofern w_1 eine natürliche Zahl ist, und sonst undefiniert ist. Wählt man nun w so, daß w das w_1-te Zeichen seines eigenen Textes liefert, so gilt offenbar:

$$f(w,w_1)=„w_1\text{-tes Zeichen von } w“=e_w(w_1) \text{ für alle } w_1 \in A^*.$$

Als Folgerung aus dem Rekursionstheorem (Version 1) ergibt sich die Existenz eines selbstreproduzierenden Programms.

Folgerung
Es gibt ein selbstreproduzierendes Programm.

Beweis
Sei $f: A^* \to A^*$ die Identität, also

$f(x)=x$ für alle $x \in A^*$.

f ist berechenbar und erfüllt die Voraussetzungen des Rekursionstheorems mit i=0. Dann gibt es ein Wort $w \in A^*$, so daß gilt:

$f(w)=e_w$.

Andererseits folgt aus der Definition von f

$f(w)=w$

und daher

$e_w=w$.

w ist das gesuchte selbstreproduzierende Programm.

Interessant und für (leider meist bösartige) Anwendungen relevant ist die Tatsache, daß man jedes beliebige Programm w in ein Programm w' umwandeln kann, daß sich selbstreproduziert, sich ansonsten aber so verhält wie w. Man kann also z.B. ein Primzahlprogramm schreiben, daß zunächst die übliche Primzahltestfunktion realisiert und sich kurz vor seiner Beendigung noch selbst ausgibt, auf Wunsch auf mehrfach. Auch dies zeigt man leicht mit dem Rekursionstheorem.

Folgerung
Zu jedem Programm w (hier) mit einem Eingabe- und einem Ausgabewert kann man ein Programm w' konstruieren, das zunächst dieselbe Funktion wie w berechnet und sich anschließend zweimal selbst ausgibt.

Beweis
Sei w ein beliebiges Programm mit einem Eingabe- und einem Ausgabewert. $g: A^* \to A^*$ sei die durch w berechnete Funktion. Wir definieren eine Funktion $f \in P^2$ mit

$f: A^* \times A^* \to A^*$ durch

$f(x,y)=g(y)xx$.

Da g nach Voraussetzung berechenbar ist, ist offenbar auch f berechenbar. Nach dem Rekursionstheorem gibt es dann ein Wort $u \in A^*$, so daß für alle $y \in A^*$ gilt:

$$f(u,y) = e_u(y)$$

und daher

$$e_u(y) = g(y)uu.$$

u ist das gesuchte selbstreproduzierende Programm. Es berechnet zunächst die Funktion g(y) und gibt sich anschließend zweimal aus.

Auch die Umwandlung eines beliebigen Programms in ein selbstreproduzierendes nach obigem Schema kann selbst wieder maschinell durch ein Programm erfolgen.

Konstruktion eines selbstreproduzierenden Programms

Man kann ein selbstreproduzierendes Programm wenig aufwendig schematisch schreiben, wobei man das eingangs erwähnte Problem der Hochkommas und das Überholproblem bewältigen muß.

Man definiert für das zu schreibende Programm eine Textkonstante T, die das gesamte Programm enthält und aus der man Zeichen für Zeichen ausgibt. Da die Textkonstante sich nicht selbst enthalten kann, läßt man genau sie in T fehlen. Wenn man mit der zeichenweisen Ausgabe an die Stelle kommt, wo die Textkonstante selbst auszugeben wäre, unterbricht man die zeichenweise Ausgabe und gibt die Textkonstante en bloc aus; anschließend setzt man mit der zeichenweisen Ausgabe fort.

Ansatz:

```
program repro(output);
const T='program repro(output); const T=;var i: integer; begin for i:=1 to X
do begin if i=Y then write(chr(39),T,chr(39)) else write(T[i]) end end.';
var i: integer;
begin
    for i:=1 to X do
    begin
        if i=Y then write(chr(39),T,chr(39)) else write(T[i])
    end
end.
```

Man erkennt: In der Definition von T heißt es '... const T=; var ...', d.h. hinter dem Gleichheitszeichen fehlt genau der Text der Konstanten T. An die Stelle von X setzt man die Länge des Programmtextes einschl. Leerzeichen und Zeilenumbrüchen, jedoch ohne die Länge der Konstanten T, Y ist die Programmposition, an der der Inhalt von T beginnt. Wenn sich der Autor nicht verzählt hat, ist X=124 (Länge von X ist drei und muß mitge-

zählt werden) und Y=30. Man beachte, daß das ausgegebene Programm nicht die gleiche Zeilenstruktur besitzt wie das Originalprogramm, was man aber leicht durch zusätzliche Anweisungen sicherstellen kann.

Anläßlich eines studentischen Wettbewerbs an der Universität Dortmund vor vielen Jahren mit dem Ziel, das kürzeste selbstreproduzierende Programm zu ermitteln, gewann der Sieger mit folgendem Programm:

```
program repro2(output);
const  d = 39;
       b = ';begin writeln(c,chr(d),b,chr(d),chr(59));
            writeln(chr(67),chr(61),chr(d),c,chr(d),b) end.';
       c = 'program repro2(output); const d=39;b=';
begin
      writeln(c, chr(d), b, chr(d), chr(59));
      writeln(chr(67), chr(61), chr(d), c, chr(d), b)
end.
```

Das Rekursionstheorem und universelle Viren

Eine interessante, wenn auch nicht ganz realistische Anwendung besitzt das Rekursionstheorem in seiner 2. Fassung. Es identifiziert die Grenzen von berechenbaren Funktionen und besagt anschaulich, daß eine berechenbare Funktion nicht ein beliebig kompliziertes Ein-/Ausgabeverhalten besitzen kann. Denn man kann zu jeder beliebigen berechenbaren Funktion f: $A^* \rightarrow A^*$, die also einen Text einliest und einen Text als Ergebnis liefert, stets ein Programm w finden, das vor wie nach der Transformation durch f immer noch die gleiche Funktion berechnet. Die Funktion f_w, die w berechnet, ist also gewissermaßen *immun* gegen die Modifikation durch f.

Beispiele

1. Sei f z.B. die Funktion, die einen Text einliest und alle +-Zeichen durch *-Zeichen und alle Zeichen ‘x’ durch ‘y’ ersetzt. Dann gibt es ein Programm w, dessen Funktion auch nach der Transformation immer noch dieselbe ist. Offenbar könnte w ein Programm sein, in dem gar keine x und keine +-Zeichen vorkommen.
2. Sei f die Funktion, die aus dem eingegebenen Text das erste Zeichen entfernt. Offenbar zerstört f jedes Pascal-Programm w, da es aus dem ersten Wort ‘program’ den Text ‘rogram’ macht. Anschließend ist die berechnete Funktion von w undefiniert. Nur ein Programm, das bereits undefiniert ist, bleibt auch unter der Transformation von f undefiniert und berechnet daher vorher wie nachher dieselbe Funktion. Ein

pathologischer Fall, der aber die Grenzen des Rekursionstheorems aufzeigt und seine Aussage relativiert.

Satz (Rekursionstheorem, Version 2)
Zu jeder totalen, also überall definierten Funktion $f \in P^1$ und zu jedem $i \geq 0$ gibt es ein Wort $w \in A^*$, so daß für alle $w_1, w_2, ..., w_i \in A^*$ gilt:

$$e_{f(w)}(w_1, w_2, ..., w_i) = e_w(w_1, w_2, ..., w_i),$$

wobei e eine universelle Funktion für P^i ist.

Aus dem Rekursionstheorem 2 kann man folgern, daß es keine *universellen* Viren gibt, die *jedes* Individuum schädigen. Dazu folgendes Szenario:
Eine feindliche Macht hat Zugang zum zentralen Rechenzentrum des Gegners bekommen, in dem *alle* Programme

$$w_1, w_2, w_3, ...$$

aufbewahrt werden. Alle Programme mögen Funktionen von A* nach A* berechnen.
Diese Programme sollen unbrauchbar gemacht werden. Einfach alle Programme zu löschen, hilft nicht weiter, da der Gegner dann den Sabotageakt sofort merkt und entsprechende Gegenmaßnahmen einleiten kann. Daher möchte man einen Virus einschleusen, der alle diese Programme nach einem vorgegebenen Verfahren verändert, so daß nachher jedes Programm fehlerhafte Resultate errechnet. Hat das Unternehmen Aussicht auf Erfolg?
Sei α der eingeschleuste Virus, er berechne die Funktion

$$f_\alpha: A^* \rightarrow A^* \text{ mit}$$

$$f_\alpha(w) = \begin{cases} w_i', & \text{falls } w = w_i \text{ für ein } i \in IN, \\ \text{„read(x); write(x)“}, & \text{sonst.} \end{cases}$$

Nach dem Rekursionstheorem 2 gibt es dann ein $w \in A^*$, so daß für alle $x \in A^*$ gilt:

$$e_{f(w)}(x) = e_w(x),$$

wobei e eine universelle Funktion ist. Daraus folgt also: w und f(w) berechnen die gleiche Funktion. Bei w muß es sich offenbar um eines der Programme w_i handeln. Folglich berechnen das Originalprogramm w_i und das geänderte Programm $f(w_i) = w_i'$ die gleiche Funktion. w_i ist also gegen den eingeschleusten Virus resistent.
Es ist entscheidend, daß alle Programme, auch nutzlose, im Rechenzentrum gelagert sind, anderenfalls greift die Argumentation mit dem Rekursionstheorem nicht.

Der Sabotageakt, alle Programme systematisch zu verändern, also einen universellen Virus zu entwickeln, kann also nicht zum Erfolg führen, denn zu jedem Virus gibt es

mindestens ein resistentes Programm. Welches das ist und ob es sich dabei um ein nützliches oder nutzloses Programme handelt, weiß man vorher nicht. Folglich ist das Ergebnis nur ein schwacher Trost und auf keinen Fall beruhigend.

Literatur

[1] Von Neumann, John, *Theory of Self-Reproducing Automata*. Edited and completed by A. W. Burks. University of Illinois Press, Urbana, Illinois (1966)

[2] A. Oberschelp: Rekursionstheorie. BI Wissenschaftsverlag Mannheim 1993.

[3] Heise online: Zum 100. Geburtstag von John von Neumann http://www.heise.de/newsticker/data/bo-28.12.03-000 (geprüft: 13.01.2004)

Prof. Dr. Andreas Schwill
Lehrstuhl für Didaktik der Informatik
Universität Potsdam
August-Bebel-Str. 89
14482 Potsdam
Email: schwill@cs.uni-potsdam.de
WWW: http://www.informatikdidaktik.de