Quines

Florian Pommerening

19. Juli 2009

Seminar: Selbstbezüglichkeit
Arbeitsgruppe für Grundlagen der Künstlichen Intelligenz
Institut für Informatik
Universität Freiburg
Sommersemester 2009

1 Einleitung

Als *Quines* werden Programme bezeichnet, die ihren eigenen Quelltext ausgeben. Diese Aufgabe wirkt auf den ersten Blick in den meisten Programmiersprachen unmöglich. Eine Konsequenz aus Kleenes Rekursionstheorem ist jedoch, dass es in jeder Turing-vollständigen Sprache Quines geben muss.

Im Folgenden wird der Begriff „Quine“ eindeutiger definiert und die Existenz von Quines bewiesen. Der Beweis ist konstruktiv und ergibt daher eine Konstruktionsvorschrift für Quines in beliebigen, Turing-vollständigen Sprachen. Im darauf folgenden Teil werden praktische Probleme bei der Umsetzung dieser allgemeinen Konstruktion beschrieben und verschiedene Lösungsansätze vorgestellt. Die Eigenschaften von Quines werden schließlich anhand einer Analogie zur Molekularbiologie diskutiert.

1.1 Definition

Der Name „Quine“ leitet sich von Willard Van Orman Quine, einem amerikanischen Philosophen und Logiker ab. Quine beschäftigte sich unter anderem mit indirekten Selbstreferenzen, wie sie zum Beispiel in dem folgendem, vom ihm aufgestellten Paradoxon vorkommen:

> *“yields falsehood when preceded by its quotation” yields falsehood when preceded by its quotation*

Dieser Satz ist eine indirekte Selbstreferenz, da er sich auf sich selbst bezieht ohne dabei Konstruktionen wie „dieser Satz“ zu verwenden.

Eine exakte Definition von Quines ist schwierig, da einige Techniken als Mogelei angesehen werden, auch wenn sie selbstreplizierende Programme erzeugen. Die folgende Definition ist daher eher als Richtlinie anzusehen.

Definition 1. *Ein* Quine *ist der nichtleere Quelltext q eines Programms, das bei seiner Ausführung exakt q ausgibt ohne auf die Quelltextdatei zuzugreifen, externe Programme oder komplexe Bibliotheksfunktionen zu benutzen.*

Die Einschränkung auf nichtleere Programme wurde nach dem *International Obfuscated C Code Contest 1994* gemacht. Eine 0 Byte große Datei wurde damals als kleinstes Quine der Welt abgegeben und erhielt einen Preis für „Worst Abuse of the Rules“.

Auf die eigene Quelltextdatei oder spezialisierte externe Programme zuzugreifen würde die Aufgabe in den meisten Fällen trivial machen und wird daher auch ausgeschlossen. Die Verwendung von Bibliotheksfunktionen wurde ausgeschlossen, da in einigen Sprachen Bibliotheken existieren, die den Zugriff auf den Quelltext erlauben. Erlaubt sind natürlich Funktionen, die für die Ausgabe von Strings nötig sind.

2 Kleenes Rekursionstheorem

Die Entwicklung eines Quines scheint nur mit hohen Programmierfähigkeiten möglich zu sein. In manchen Sprachen, erscheint ein Quine sogar vollkommen unmöglich. Die esoterische Programmiersprache *Shakespeare* benötigt zum Beispiel eine Zeile Quelltext für die Definition eines Zeichens.

Kleene [2] postulierte jedoch 1938 zwei Rekursionstheoreme, von denen das zweite als Sonderfall die Existenz von Quines in Turing-vollständigen Sprachen garantiert. Das erste Rekursionstheorem beschreibt die Existenz von kleinsten Fixpunkten bei *enumeration operators* und wird hier nicht weiter besprochen. Um das zweite Theorem zu besprechen, benötigen wir zunächst etwas Notation.

2.1 Notation

Definition 2. *Eine partielle Funktion f heißt genau dann* berechenbar*, wenn es eine Turing-Maschine*[1] *gibt, die für alle Eingaben $x \in Def(f)$ mit der Ausgabe $f(x)$ hält und die für alle Eingaben $y \notin Def(f)$ nicht hält.*

Insbesondere existiert damit für eine totale, berechenbare Funktion eine Turing-Maschine, die auf allen Eingaben terminiert. Wir betrachten hier berechenbare Funktionen als Funktionen $f : \mathbb{N}^k \to \mathbb{N}$. Computerprogramme, die andere Argumente einlesen, sind in dieser Definition mit eingeschlossen, da die Eingaben auch als binär codierte Zahlen angesehen werden können.

Definition 3. *Zwei partielle Funktionen f und g sind genau dann* streng gleich *(in Zeichen: $f \simeq g$), wenn $Def(f) = Def(g)$ und für alle $\vec{x} \in Def(f)$ gilt: $f(\vec{x}) = g(\vec{x})$*

Zwei Turing-Maschinen, die streng gleichen Funktionen entsprechen, terminieren auf einer Eingabe also entweder beide nicht, oder sie halten mit der gleichen Ausgabe.

Im Folgenden verwenden wir eine Gödel-Nummerierung der berechenbaren Funktionen, also ein Verfahren, das jeder berechenbaren Funktion eine eindeutige Nummer zuordnet. Eine Möglichkeit dies zu erreichen ist, jede Funktion jeweils mit der kleinsten Turing-Maschine zu indizieren, die diese Funktion berechnet. Als Indizes werden natürliche Zahlen benutzt, was ähnlich wie bei den Argumenten möglich ist, wenn die Codierung als binär codierte Zahl aufgefasst wird.

[1] Dies ist nur eine von vielen äquivalenten Definitionen. Andere Definitionen verwenden andere Berechenbarkeitsmodelle wie zum Beispiel den Lambda-Kalkül.

Definition 4. *Mit ϕ_i wird die Funktion mit der Gödel-Nummer i bezeichnet (bezüglich einer beliebigen aber festen Gödel-Nummerierung).*

Für eine konkrete Programmiersprache kann man sich den Index i als Quelltext vorstellen und die Funktion ϕ_i als die Funktion, die von diesem Quelltext beschreiben wird.

Mit diesen Definitionen kann nun Kleenes Theorem definiert werden.

2.2 Das Rekursionstheorem

Theorem 2.1 (Zweites Rekursionstheorem)**.** *Für jede totale, berechenbare Funktion f gibt es einen Index q, so dass*

$$\phi_q \simeq \phi_{f(q)}.$$

Die im Theorem vorkommende Funktion f bezeichnen wir als Quelltexttransformation, da sie zu einem Index einen neuen Index berechnet. Ein Beispiel dafür ist ein Programm, das einen Quelltext als Parameter erhält und in diesem while-Schleifen durch for-Schleifen ersetzt.

Das Theorem besagt, dass für jede Quelltexttransformation f ein Quelltext q existiert, dessen Semantik (ϕ_q) durch f nicht verändert wird. Zum Beispiel verändert eine Transformation, die ihrem Parameter eine Kommentarzeile hinzufügt, nie dessen Semantik. Der Index q wird daher auch als *semantischer Fixpunkt* und das Theorem auch als *Fixpunkttheorem* bezeichnet.

Definiert man die Transformation f auf allen Eingaben x als

$$f(x) := \texttt{print } x\texttt{;}$$

dann existiert nach dem Rekursionstheorem ein Quelltext q, dessen Semantik die gleiche ist wie die von $f(q)$ also `print` q`;`. Die Semantik von q ist daher q auszugeben: q ein Quine.

Das Rekursionstheorem garantiert semantische Fixpunkte auch für deutlich komplexere Transformationen. So gibt es zum Beispiel Programme, die ihren Quelltext rückwärts ausgeben oder ihre eigene Checksumme berechnen und ausgeben. Die Konstruktion solcher Programme verläuft analog zu den in Abschnitt 3 beschriebenen Verfahren, wird hier aber nicht näher besprochen.

2.3 Beweis

Der Beweis von Kleenes Theorem ist konstruktiv und ermöglicht die Konstruktion eines Quines in jeder Turing-vollständigen Sprache. Wir werden ihn daher in diesem Abschnitt detailliert nachvollziehen.

Der Beweis stützt sich auf drei Theoreme, die hier nur informell besprochen und nicht bewiesen werden

Theorem 2.2. *Die Verkettung von zwei berechenbaren Funktionen ist eine berechenbare Funktion.*

Beweisskizze. Für jede der Funktionen existiert eine Turing-Maschine. Diese können miteinander verknüpft werden um eine Turing-Maschine zu erhalten, die die Verkettung berechnet. □

Korollar 2.3. *Die Verkettung von zwei totalen, berechenbaren Funktionen ist eine totale, berechenbare Funktion.*

Theorem 2.4 (UTM-Theorem)**.** *Für jedes $k \in \mathbb{N}$ ist die partielle Funktion $u_\phi^k : \mathbb{N}^{1+k} \to \mathbb{N}$ berechenbar, die für alle Indizes i von berechenbaren Funktionen $\phi_i : \mathbb{N}^k \to \mathbb{N}$ und für alle $\vec{x} \in Def(\phi_i)$ folgendermaßen definiert ist*

$$u_\phi^k(i, \vec{x}) := \phi_i(\vec{x}).$$

Das UTM-Theorem beschreibt die Existenz von *universellen Turing-Maschinen*, die in der Lage sind, andere Turing-Maschinen zu simulieren. In einer Programmiersprache entspricht den Funktionen u_ϕ^k ein Interpreter für die Sprache, der in der Sprache selbst geschrieben wurde.

Theorem 2.5 (s_m^n-Theorem)**.** *Sei i der Index der berechenbaren Funktion $\phi_i : \mathbb{N}^{m+n} \to \mathbb{N}$. Dann gibt es eine totale, berechenbare Funktion $s_m^n : \mathbb{N}^{1+m} \to \mathbb{N}$, für die gilt*

$$\phi_{s_m^n(i,\vec{x})} \simeq (\lambda \vec{y}.\, \phi_i(\vec{x}, \vec{y})) \qquad \text{für } \vec{x} \in \mathbb{N}^m, \vec{y} \in \mathbb{N}^n.$$

Mit der Funktion s_m^n können die ersten m Parameter einer berechenbaren Funktion mit $m + n$ Parametern auf feste Werte gesetzt werden. Dies lässt sich in den meisten Programmiersprachen durch eine zusätzliche Funktion realisieren, die den eigentlichen Funktionsaufruf mit den fest programmierten Parametern macht.

Für den Beweis des Rekursionstheorems sei nun f eine beliebige aber totale, berechenbare Funktion. Sei außerdem

$$G(x, \vec{y}) := \phi_{\phi_x(x)}(\vec{y}). \tag{1}$$

Die Funktion G interpretiert ihren ersten Parameter x als Quelltexttransformation, die auf sich selbst angewendet einen neuen Index („Quelltext“) ergibt. Dieser wird mit den restlichen Parametern $\vec{y}$ aufgerufen.

Nach dem UTM-Theorem sind die Funktionen $u_\phi^1(x, x) = \phi_x(x)$ und $u_\phi^k(i, \vec{y}) = \phi_i(\vec{y})$ berechenbar, wenn x (resp. i) der Index einer berechenbaren Funktion ist. Nach Theorem 2.2 ist damit auch die Verkettung $\phi_{\phi_x(x)}(\vec{y}) = G(x, \vec{y})$ berechenbar, es gibt also einen Index g mit $G \simeq \phi_g$.

Nach dem s_m^n-Theorem ist daher die Funktion $h(x) := s_n^1(g, x)$ total, berechenbar und es gilt

$$\phi_{h(x)} \simeq \lambda\vec{y}.\phi_g(x, \vec{y}) \overset{(1)}{\simeq} \phi_{\phi_x(x)}. \tag{2}$$

Intuitiv beschreibt h eine Quelltexttransformation, die zu einem gegebenen Quelltext x einen Quelltext generiert, der x auf sich selbst simuliert, und das Ergebnis dann auf der Eingabe $\vec{y}$ simuliert. Wichtig dabei ist, dass h das Programm x nicht simuliert, sondern nur Quelltext erstellt, der dies macht.

Sei nun e der Index von $f \circ h$. Dieser Index existiert da f, h und damit nach Korollar 2.3 auch $f \circ h$ totale, berechenbare Funktionen sind.

$$\phi_e \simeq f \circ h \tag{3}$$

Der Index e beschreibt intuitiv gesehen den Quelltext eines Programms, das zu einer Eingabe x den Wert $f(h(x))$ berechnet.

Dann gilt für $q := h(e)$:

$$\phi_q \simeq \phi_{h(e)} \overset{(2)}{\simeq} \phi_{\phi_e(e)} \overset{(3)}{\simeq} \phi_{(f\circ h)(e)} \simeq \phi_{f(h(e))} \simeq \phi_{f(q)} \tag{4}$$

Insgesamt bildet das gewählte q einen semantischen Fixpunkt und belegt damit das Rekursionstheorem. Sowohl e als auch h sind totale, berechenbare Funktionen, daher kann q tatsächlich in jeder Turing-vollständigen Sprache konstruiert werden.

3 Konstruktion

Im Folgenden werden Techniken zur Implementierung von Quines besprochen. Wir beginnen mit einem Ansatz, der sich direkt aus dem Beweis ableiten lässt, jedoch zu sehr langen Quines führen kann.

3.1 Direkte Konstruktion

Für die direkte Umsetzung des Beweises werden die Funktionen h und e benötigt. Da e von h abhängt, muss zuerst h konstruiert werden.

Konstruktion von h. Die Funktion h stellt eine Quelltexttransformation dar, deren Ausgabe x auf x simuliert, und das Ergebnis dann auf der Eingabe $\vec{y}$ simuliert. Listing 3.1 zeigt ein Beispiel dafür.

Wir gehen hier davon aus, dass es in der Sprache eine Funktion namens `simulate(i, `$\vec{y}$`)` gibt, die den Quelltext `i` mit den Parametern $\vec{y}$ simuliert.

```
1 (λ x.
2     return "(λy⃗. x_on_x = simulate(" + x + ", " + x + ");"
3                                 + "return simulate(x_on_x, y⃗);)")
```

Listing 3.1: Quelltext von h.

Eine weitere Annahme ist, in der verwendeten Sprache anonyme Funktionen möglich sind. Diese Annahme wurden hier und auch bei der Konstruktion von e gemacht, um die Lesbarkeit zu erhöhen. Im Anschluss wird jedoch gezeigt wie man die Funktionen auch ohne diese Annahmen definieren kann.

Konstruktion von e. Der Quelltext e soll ein Programm beschreiben, das $f \circ h$ berechnet. Das Ergebnis der Funktion f ist aber bei einem Quine immer der Quelltext, das Argument von f ausgibt und kann daher in den meisten Sprachen durch ein einfaches `"print"` realisiert werden, wie in Listing 3.2 demonstriert. Der Eintrag `h(x)` muss dabei mit dem, von der Funktion h berechneten, Quelltext $h(x)$ ersetzt werden.

```
1 (λ x. return "print " + h(x);)
```

Listing 3.2: Definition von e.

Das eigentliche Quine erhält man dann durch die Berechnung von `h(e)`. Dieser Prozess wird auch Bootstrapping genannt.

Praktische Probleme. Die oben beschriebene Konstruktion enthält drei Annahmen, die bei der praktischen Umsetzung zu Problemen führen können.

- *Die Existenz der Interpreterfunktion* `simulate` *wird vorausgesetzt.*
  Aus dem UTM-Theorem folgt, dass es möglich ist eine solche Funktion zu programmieren. Trotzdem ergibt sich das Problem, dass die Funktion nicht Teil der Sprache ist, sondern im Quine enthalten sein muss. Dies lässt sich jedoch durch eine Erweiterung der Funktion f lösen, wie sie in Listing 3.3 demonstriert wird. Da die Definition von `simulate(i,` $\vec{y}$`)` vollkommen unabhängig von h und e ist kann sie vor dem eigentlichen Quelltext des Quines definiert werden. Diese Technik kann auch dazu verwendet werden um andere Funktionen, wie zum Beispiel h, auszulagern.

- *Die Existenz von anonymen Funktionen wird vorausgesetzt.*
  Auch diese Einschränkung lässt sich mit einem eigenen Interpreter um-

```
1 def simulate(i, y⃗):
2     # Source code of simulate
3
4 (λ x.
5     value = h(x)
6     return "print " + <Definition of simulate> +
7            "print " + value)
```

Listing 3.3: Definition von e erweitert um einen Interpreter.

gehen, indem man diesen mächtig genug macht, um anonyme Funktionen zu simulieren. Das UTM-Theorem garantiert unter anderem die Existenz eines solchen Interpreters.

- *Die Berechnung von `h(e)` ergibt ineinander geschachtelte Programme.* Das kann zum Problem werden, wenn die Programme, wie hier angedeutet, als Strings codiert werden und es nur ein Begrenzungszeichen für Strings gibt. Die Lösung sind zwei zusätzliche Funktionen, die dem Quine mit der oben beschriebenen Technik hinzugefügt werden. Eine dieser Funktionen ist dafür zuständig, einen String zu escapen, also alle problematischen Zeichen durch im String gültige Zeichen zu ersetzen. Die zweite Funktion macht diese Ersetzung rückgängig. Diese beiden Funktionen müssen an allen Stellen in e, h und `simulate` eingesetzt werden, an denen Strings erstellt bzw. interpretiert werden.

Appendix A.1 zeigt ein Beispiel, in dem diese Techniken umgesetzt wurden.

3.2 Einsparungen am Interpreter

Der komplexeste Teil der eben beschriebenen Methode ist die Entwicklung eines Interpreters. Diese Arbeit erscheint zum Großteil überflüssig, da im eigentlichen Programm nur ein kleiner Ausschnitt der Sprache verwendet wird. Tatsächlich werden im Beispiel von Appendix A.1 nur die Befehle `simulate`, `program_output`, `data_output` und `h` benutzt. Der Interpreter muss also nur diese vier Funktionen, die Verkettung und die Verschachtelung von Funktionen unterstützen, was sich in den meisten Sprachen leicht mit einer `switch/case` Konstruktion und einigen Stringfunktionen realisieren lässt.

Bei der weiteren Analyse des konstruierten Quines fällt ein Prinzip auf, das sich in vielen Quines wiederfindet: Der Quelltext `e` enthält das vollständige, restliche Programm, als Daten. Durch die Simulation von `h` wird `e` auf sich selbst ausgewertet und dadurch verdoppelt. Das entstehende Quine enthält das ursprüngliche Programm einmal als Code und einmal als Daten.

Dieses Prinzip kann ausgenutzt werden um ein kompakteres Quine zu konstruieren, das ohne einen eigenen Interpreter auskommt. Die folgende Konstruktion basiert nicht mehr direkt auf Kleenes Beweis, weshalb nicht garantiert ist, dass sie in jeder Turing-vollständigen Sprache möglich ist. Sie ist jedoch in allen gängigen Hochsprachen und auch in einigen esoterischen Sprachen (u.a. Shakespeare) möglich.

```
1 data = [ ... ]
2 print "data = ["
3 for datum in data:
4     print interpret_as_data(datum)
5 for datum in data:
6     print interpret_as_code(datum)
```

Listing 3.4: Quine ohne Interpreter.

Listing 3.4 zeigt unvollständigen Pseudocode für diesen Ansatz. Der Quelltext besteht aus drei Teilen:

- Ein *Präfix* enthält den Quelltext der nötig ist, um eine Sequenz zu definieren und eventuell zusätzliche Befehle, die das Schreiben in diese Sequenz initialisieren. (Hier: `data = [`)
- Die *Datendefinition* enthält den dritten Teil des Programms codiert als Daten. Hierfür bietet sich zum Beispiel die Codierung als ASCII-Werte der Zeichen an. In der Datendefinition werden diese codierten Werte der im Präfix definierten Sequenz hinzugefügt. (Hier ausgelassen)
- Der *Code* besteht wiederum aus drei Teilen:

  Zuerst wird das Schreiben in die Sequenz abgeschlossen (oft mit einer schließenden Klammer) und der Präfix ausgegeben. Danach werden die Daten benutzt um die Datendefinition zu generieren. Wenn die ASCII-Werte benutzt wurden, reicht es häufig die betrachtete Zahl von einem Komma gefolgt auszugeben. Schließlich werden die Daten benutzt um den Code zu generieren. Wurden ASCII-Werte benutzt, reicht es hier den zum ASCII-Wert passenden Buchstaben auszugeben.

3.3 Weitere Techniken

Die im letzen Abschnitt beschriebene Konstruktion führt in vielen Sprachen zwar zu einem Quine, das sehr lang werden kann. Die Entwicklung von be-

sonders kurzen[2] oder eleganten Quines hat sich daher zu einer Art Sport entwickelt und erfordert im Gegensatz zur reinen Konstruktion oft den Einsatz von sprachspezifischen Mitteln. Aus diesem Grund kann hier keine allgemein verwendbare Konstruktionsvorschrift angegeben werden. Stattdessen werden exemplarisch zwei Techniken besprochen, die häufig eingesetzt werden.

```
1 ((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))
```

Listing 3.5: Ein Quine in *Scheme*.

Listing 3.5 zeigt eine Technik, die besonders in Sprachen wie *Lisp* und *Scheme* eingesetzt wird, in denen die direkte Verwendung von Daten als Code möglich ist. Die hier definierte Funktion entspricht dem Lambda-Ausdruck $(\lambda x.xx)(\lambda x.xx)$, der eine Verdopplungsfunktion auf sich selbst anwendet.

```
1 int main(void){
2 char str[] = "int main(void){
3                char str[] = %c%s%c;
4                printf(str, 0x22, str, 0x22);}"
5 printf(str, 0x22, str, 0x22);}
```

Listing 3.6: Ein Quine in *C* geschrieben von Aldo Cortesi. Zeilenumbrüche wurden eingefügt um die Lesbarkeit zu erhöhen.

Listing 3.6 zeigt ein Quine in *C*, in dem man die Trennung von Daten und Code gut erkennen kann. Hier wird die Funktion `printf` verwendet um die Daten zu verdoppeln und so einmal als Daten und einmal als Code auszugeben. Ein weiterer oft verwendeter Trick ist die Verwendung von ASCII-Werten für die Anführungszeichen, die durch `printf` für die Platzhalter `%c` eingesetzt werden. Die ASCII-Werte können in den Daten vorkommen, so dass kein escapen notwendig ist.

4 Diskussion

Quines wurden von Douglas Hofstadter [1] mit sich selbst reproduzierenden Zellen verglichen. Diese Analogie geht über die reine Selbstreproduktion hinaus und zeigt deutliche Parallelen zu den hier vorgestellten Konstruktionsverfahren. Der interessante Teil der Selbstreproduktion ist dabei die Kopie der DNA innerhalb der Zelle. Den Aufbau der neuen Zelle werden wir hier

[2] Zum Beispiel `6581840dnP` im Unix desk calculator dc.

nicht betrachten, da die Zelle für den Kopiervorgang der DNA die Rolle einer Laufzeitumgebung oder eines Interpreters übernimmt.

4.1 Reproduktion von Zellen

Die DNA befindet sich im Zellkern jeder Zelle und interagiert dort mit bestimmten Enzymen. Das Enzym *RNA-Polymerase* ist dabei für die sogenannte *Transkription* zuständig. Die Polymerase kopiert Abschnitte der DNA (die Gene) auf eine neue Struktur, die *Messenger-RNA* (mRNA) genannt wird. Bei der Transkription unterscheidet man zwischen *Introns*, die keine relevante Information enthalten, und *Extrons*, die das „Programm“ der DNA beinhalten. Introns werden bei der Transkription übersprungen.

Die entstandene mRNA ist klein genug um aus dem Zellkern in die Zelle zu gelangen. Die *Ribosomen* (Komplexe aus Proteinen und RNA) „interpretieren“ dort die mRNA bei der sogenannten *Translation*. Je drei aufeinanderfolgende Nukleotide (ein *Codon*) codieren eine Aminosäure, die vom Ribosom aneinandergereiht werden. Die entstehenden Ketten bilden verschiedene Proteine, insbesondere Enzyme.

Einige der so hergestellten Enzyme (Helikase, DNA-Polymerasen, u.a.) gelangen zurück in den Zellkern und bewirken dort den eigentlichen Kopiervorgang der DNA. Die Helikase trennt die DNA in zwei Stränge auf, die von je einer Polymerase wieder zur vollen DNA Doppelhelix aufgefüllt werden.

4.2 Analogie zu Quines

Dieser, zugegebenermaßen sehr grobe, Überblick über die Kopie der DNA beinhaltet einige Parallelen zum Aufbau von Quines. Die auf die mRNA kopierte DNA wird von den Ribosomen bei der Translation als Programm interpretiert um die zur Reproduktion nötigen Enzyme herzustellen. Diese Enzyme interpretieren die DNA jedoch als Daten und kopieren sie ohne ihre Bedeutung zu verstehen. Der gleiche „Quelltext“ kommt hier wie bei Quines ($h(e)$ bzw. $\phi_e(e)$) einmal als Programm und einmal als Daten vor.

In diesem Zusammenhang wurde auch der Begriff des Introns in die Terminologie von Quines übernommen: Hier wird ein Teil der Daten Intron genannt, das beliebig verändert werden kann, so dass das geänderte Programm wieder ein Quine ist. Ein Intron ist wie bei der DNA auch ein Teil der Daten.

Wie das erste in dieser Arbeit konstruierte Quine, enthält die DNA die Codierung ihres eigenen Interpreters in Form von Nukleotidketten, die die verarbeitenden Enzyme codieren. Die Reproduktion der DNA geht sogar über die hier vorgestellten Quines hinaus, da sie nicht nur sich selbst, sondern auch ihren Interpreter, die Zelle vervielfältigt.

Literatur

[1] Douglas R. Hofstadter. *Gödel, Escher, Bach. Ein endlos geflochtenes Band (Aufl. 10)*. Klett Cotta im Deutschen Taschenbuch Verlag, Stuttgart, 2004.

[2] Stephen Cole Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):150–155, 1938.

[3] David Madore. Quines (self-replicating programs), 2009. [Online; accessed 29-May-2009].

A Appendix

A.1 Quelltext

Die Listings A.1 und A.2 zeigen den Quelltext eines Bootstrappers, der aus dem (im String `definitions` übergebenem) Quelltext ein Quine erstellt. Listing A.1 beinhaltet dabei eine (arbiträr gewählte) Methode um in String ungültige Zeichen zu codieren (vgl. Abschnitt 3.1).

```
1  # START OF <definitions>
2  def escape(s):
3      s = s.replace("!",  "!exclamation")
4      s = s.replace("\n", "!newline")
5      s = s.replace("\\", "!backslash")
6      s = s.replace("\"", "!quote")
7      return s
8
9  def unescape(s):
10     s = s.replace("!quote", "\"")
11     s = s.replace("!backslash", "\\")
12     s = s.replace("!newline", "\n")
13     s = s.replace("!exclamation", "!")
14     return s
```

Listing A.1: Bootstrapper für ein streng konstruiertes Quine in Python.

Durch die eigene Syntax für Funktionen (`parameter` und `simulate`), sind keine anonymen Funktionen mehr nötig. Listing A.2 benutzt zur eigentlichen Interpretation die Python-Funktion **`exec`**, die in einer anderen Sprache eventuell durch einen selbstgeschriebenen Interpreter ersetzt werden müsste.

```
16 def parameter(param):
17     return "!parameter!quote" + escape(param) + "!quote"
18
19 def simulate(command):
20     splitted = command.split("!parameter")
21     program = splitted[0]
22     input = splitted[1:]
23     for i in xrange(len(input)):
24         program = program.replace("{" + str(i) + "}", \
25                                   unescape(input[i]))
26     exec(program)
27
28 def program_output(x):
29     print unescape(x)
30
31 def data_output(x):
32     print "simulate(\"" + x + "\")"
33
34 def h(x):
35     return "simulate({0})" + parameter(
36                 "simulate({0})" + parameter(
37                      x + parameter(x)))
38 # END OF <definitions>
39
40 definitions = "..." # contains line 2 - 37
41
42 def e():
43     return "program_output({0});\
44             data_output(h({1}))" \
45              + parameter(escape(definitions))
46
47 simulate(h(e()))
```

Listing A.2: Bootstrapper für ein streng konstruiertes Quine in Python.