

# Quines

Florian Pommerening

Seminar: Selbst-Referenz  
Arbeitsgruppe für Grundlagen der Künstlichen Intelligenz  
Albert-Ludwigs-Universität Freiburg

28. Juli 2009

# Motivation

## Quine in natürlicher Sprache

*Schreibe den folgenden Satz zweimal auf; beim zweiten Mal in Anführungszeichen: „Schreibe den folgenden Satz zweimal auf; beim zweiten Mal in Anführungszeichen: “*

- Anweisung erzeugt ihre eigene Kopie
  - Auch in Programmiersprachen möglich?
  - Zufall, Geschick oder grundsätzliche Regel?
  - Typische Techniken

# Inhalt

- 1 Definitionen
  - Quine
  - Daten, Code und Introns
- 2 Kleenes Rekursionstheorem
  - Intuition
  - Beweis
- 3 Techniken
  - Konstruktiver Ansatz
  - Weitere Techniken

# Was ist ein Quine?

## Definition

Ein Quine ist ein Computerprogramm, das eine Kopie seines Quelltextes als Ausgabe schreibt.

Benannt nach Willard Van Orman Quine (1908 - 2000)

- Beschäftigte sich mit indirekter Selbst-Referenz
  - Aussage eines Satzes über sich selbst
  - Kein Demonstrativpronomen („Dieser Satz ...“)

## Quines Paradoxon

„yields falsehood when preceded by its quotation“ yields falsehood when preceded by its quotation

# Was ist *kein* Quine?

Nicht erlaubt sind

- Zugriff auf externe Programme

## Bash

```
#!/bin/cat
```

- Zugriff auf die Quelltext-Datei

## Python

```
import sys
sourcecode_file = open(sys.argv[0])
for line in sourcecode_file:
    print line,
```

# Was ist *kein* Quine? (2)

Nicht gerne gesehen sind

- Leere Dateien

C, Bash, ...

- Komplexe Funktionen aus Bibliotheken

Python

```
from inspect import getsourcelines
from sys import modules
source = getsourcelines(modules[__name__])[0]
for line in source:
    print line,
```

# Daten, Code und Introns

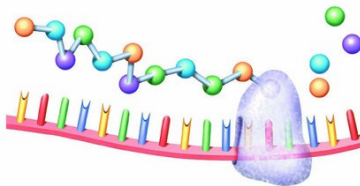
- Daten
  - Passiver Teil des Quines
  - Enthält Kopie des Codes
- Code
  - Aktiver Teil des Quines
  - Benutzt die Daten um die Daten auszugeben
  - Benutzt die Daten um den Code auszugeben
- Introns
  - Teil der Daten
  - Veränderung erhält Quine





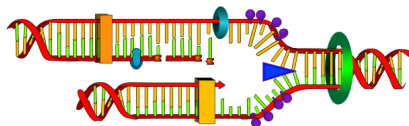
# Analogie zur Molekularbiologie (2)

- Translation der RNA durch Ribosomen (Interpreter)
  - Interpretation als **Programm**
  - Verkettung von Aminosäuren
  - Herstellung von Proteinen (insb. Enzymen)



# Analogie zur Molekularbiologie (3)

- Auftrennen des DNA-Strangs durch hergestellte Enzyme
- Kopie der Stränge
  - Enzyme interpretieren **DNA als Daten**
  - Introns werden mit kopiert



# Inhalt

- 1 Definitionen
  - Quine
  - Daten, Code und Introns
- 2 Kleenes Rekursionstheorem
  - Intuition
  - Beweis
- 3 Techniken
  - Konstruktiver Ansatz
  - Weitere Techniken

# Grundlagen der Berechenbarkeitstheorie

- Programm als Funktion  $f : \mathbb{N}^n \rightarrow \mathbb{N}$ 
  - Natürliche Zahlen als Input/Output
  - Programm terminiert immer  $\Leftrightarrow$  Totale Funktion
  - Enthält Endlosschleifen  $\Leftrightarrow$  Partielle Funktion
- Äquivalenz von Programmen  $f \simeq g$ 
  - Gleicher Definitionsbereich
    - $\Rightarrow$  Terminieren auf gleichem Input
  - Gleiche Rückgabewerte im Definitionsbereich

# Grundlagen der Berechenbarkeitstheorie (2)

- Auflistung aller Programme  $\phi_e, \forall e \in \mathbb{N}$ 
  - Zugriff über Index  $e$  („Quelltext“)
  - Ausführung/Rückgabewert eines Programms:  $\phi_e(\vec{y})$
- Funktionen die Quelltext bearbeiten
  - Beispiel: Ersetze for-Schleifen mit while-Schleifen
  - Neuer Index:  $F(e)$
  - Ausführung des veränderten Quelltexts:  $\phi_{F(e)}(\vec{y})$

# Kleenes Rekursionstheorem

## Zweites Rekursionstheorem / Fixpunktsatz

Für jede totale berechenbare Funktion  $F$  gibt es einen Index  $q$  mit

$$\phi_q \simeq \phi_{F(q)}$$

- Interpretation der Funktion  $F$ 
  - Änderungen am Quelltext
- Interpretation des Index  $q$ 
  - Quelltext, der durch  $F$  inhaltlich nicht verändert wird
  - Gleiche Funktion wie  $F(q)$
  - *Semantischer Fixpunkt*
- Für Quines:  $F(q) := \mathbf{print\ } q$

# Beweis

- Definiere Funktion  $h(x)$ 
  - Auf Eingabe  $x$  gebe den folgenden Quelltext zurück:
    - Simuliere Programm  $\phi_x$  auf  $x$
    - Simuliere das Ergebnis  $\phi_x(x)$
    - Gebe das Ergebnis  $\phi_{\phi_x(x)}$  zurück
  - $\phi_{h(x)} \simeq \phi_{\phi_x(x)}$
- Achtung!
  - $h(x)$  generiert Quelltext
  - Keine Simulation
- Ergebnis der Berechenbarkeitstheorie
  - Simulation von Programmen möglich (Interpreter)
  - $h$  ist totale, berechenbare Funktion

# Beweis (2)

- Definiere Index  $e$  („Quelltext“)
  - $\phi_e \simeq (F \circ h)$
  - Berechne zuerst Funktion  $h$
  - Benutze das Ergebnis als Input für  $F$
- Ergebnis der Berechenbarkeitstheorie
  - Verkettung von zwei totalen, berechenbaren Funktionen

⇒ Ebenfalls totale, berechenbare Funktion



## Beweis (3)

## Definitionen

$$\begin{aligned}\phi_{h(x)} &\simeq \phi_{\phi_x(x)} \\ \phi_e &\simeq (F \circ h)\end{aligned}$$

- Betrachte Ausführung von  $h(e)$

$$\phi_{h(e)} \simeq \phi_{\phi_e(e)} \quad \text{Definition von } h$$

## Beweis (3)

## Definitionen

$$\begin{aligned}\phi_{h(x)} &\simeq \phi_{\phi_x(x)} \\ \phi_e &\simeq (F \circ h)\end{aligned}$$

- Betrachte Ausführung von  $h(e)$

$$\begin{aligned}\phi_{h(e)} &\simeq \phi_{\phi_e(e)} && \text{Definition von } h \\ &\simeq \phi_{(F \circ h)(e)} && \text{Definition von } e\end{aligned}$$

## Beweis (3)

## Definitionen

$$\begin{aligned}\phi_{h(x)} &\simeq \phi_{\phi_x(x)} \\ \phi_e &\simeq (F \circ h)\end{aligned}$$

- Betrachte Ausführung von  $h(e)$

$$\begin{aligned}\phi_{h(e)} &\simeq \phi_{\phi_e(e)} && \text{Definition von } h \\ &\simeq \phi_{(F \circ h)(e)} && \text{Definition von } e \\ &\simeq \phi_{F(h(e))} && \text{Verkettung von Funktionen}\end{aligned}$$

## Beweis (3)

## Definitionen

$$\begin{aligned}\phi_{h(x)} &\simeq \phi_{\phi_x(x)} \\ \phi_e &\simeq (F \circ h)\end{aligned}$$

- Betrachte Ausführung von  $h(e)$

$$\begin{aligned}\phi_{h(e)} &\simeq \phi_{\phi_e(e)} && \text{Definition von } h \\ &\simeq \phi_{(F \circ h)(e)} && \text{Definition von } e \\ &\simeq \phi_{F(h(e))} && \text{Verkettung von Funktionen}\end{aligned}$$

- $q = h(e)$  ist der gesuchte Index

# Inhalt

- 1 Definitionen
  - Quine
  - Daten, Code und Introns
- 2 Kleenes Rekursionstheorem
  - Intuition
  - Beweis
- 3 Techniken
  - Konstruktiver Ansatz
  - Weitere Techniken

# Quine mit Kleenes Theorem

- Beweis ist konstruktiv
  - Ermöglicht Konstruktion von Quines
  - In *jeder turing-vollständigen Sprache*
- Praktische Probleme
  - $e$  muss die Funktion von  $h$  codieren
    - Vollständiger Interpreter für die Sprache
    - In der Sprache selbst geschrieben
  - $h(e) = \phi_e(e)$  enthält Programm im Programm
    - In den meisten Sprachen Probleme mit Quotes

# Quine mit Kleenes Theorem (2)

- Interpreter muss nicht vollständig sein
  - Nur Simulation von  $\phi_e(e)$  und  $F$
  - $F$  ist trivial (**print**)
- $q = h(e)$  enthält **e als Code** und **e als Daten**
- **Interpreter** benutzt **Daten**
  - um **Code** auszugeben
    - vgl. RNA produziert Proteine
  - um **Daten** auszugeben
    - vgl. Proteine kopieren DNA

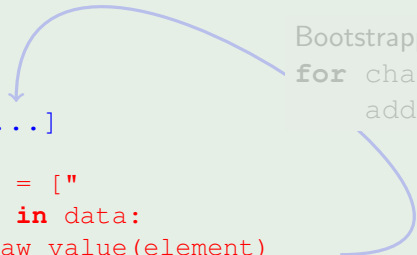
# Quine mit Kleenes Theorem (3)

## Pseudocode

```
data = [...]
```

Bootstrapping:  
**for** char **in** code:  
    add(ord(char))

```
print "data = ["  
for element in data:  
    print raw_value(element)  
print "]\n"  
for element in data:  
    print character_value(element)
```





# Quine mit Kleenes Theorem (3)

## Pseudocode

```
data = [...]
```

```
print "data = ["  
for element in data:  
    print raw_value(element)  
print "]\n"  
for element in data:  
    print character_value(element)
```

Bootstrapping:

```
for char in code:  
    add(ord(char))
```

# Weitere Techniken

- Konstruktiver Ansatz immer möglich
  - Auch in Sprachen ohne Strings
- Sportlicher Ehrgeiz
  - Finde möglichst kurzes Quine
  - Finde elegantes Quine
  - Finde Multi-Quines
    - Sammlung von  $n$  Quines
    - Aufruf ohne Parameter  $\Rightarrow$  Eigenen Quelltext ausgeben
    - Aufruf mit Parameter  $i \Rightarrow$  gebe Quine  $i$  aus
  - Finde mehrsprachige Quines

# Beispiele

## Scheme

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

## Lisp

```
( (lambda (x) (list x (list 'quote x)))  
' (lambda (x) (list x (list 'quote x))) )
```

## $\lambda$ -Kalkül

```
( $\lambda x.xx$ )( $\lambda x.xx$ )
```

- Anonyme Funktionen
- „Verdoppeln“ auf sich selbst angewendet
- Sprache unterstützt Daten als Code

## Beispiele (2)

### C

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

### PHP

```
<? printf($a='<? printf($a=%c%s%c,39,$a,39);',  
39,$a,39);
```

- printf zur Verdopplung
  - Benutzt 0x22 als " bzw. 39 als '

# Beispiele (3)

dc

6581840dnP

hq9+





q

Shakespeare

Zu groß für die Folie (112 MB)

# Fazit

- Quines reproduzieren sich selbst
  - Unterscheidung zwischen **Code** und **Daten**
  - Introns können verändert werden (vgl. DNA)
- Kleenes Rekursionstheorem
  - Für alle  $F$  gibt es  $q$  mit  $\phi_q \simeq \phi_{F(q)}$
  - Beweist die Existenz von Quines
  - Ermöglicht konstruktiven Ansatz
- Sportlicher Aspekt
  - Kleinere, schönere oder mächtigere Quines

-  Hofstadter, D. R. (2004).  
*Gödel, Escher, Bach. Ein endlos geflochtenes Band (Aufl. 10)*.  
Klett Cotta im Deutschen Taschenbuch Verlag, Stuttgart.
-  Madore, D. (2009).  
Quines (self-replicating programs).  
[Online; accessed 29-May-2009].
-  Wikipedia (2009a).  
Kleene's recursion theorem — wikipedia, the free  
encyclopedia.  
[Online; accessed 3-January-2009].
-  Wikipedia (2009b).  
Quine (computing) — wikipedia, the free encyclopedia.  
[Online; accessed 29-May-2009].

# Beispiele (4)

## Multiquine

```
data_L1 = [...]; data_L2 = [...]  
if parameter == magic_number:  
    print "data_L2 = ["  
    print_L2_data(data_L2)  
    print "]; data_L1 = ["  
    print_L2_data(data_L1)  
    print "]\n"  
    print_L2_code(data_L2)  
else:  
    print "data_L1 = ["  
    print_L1_data(data_L1)  
    print "]; data_L2 = ["  
    print_L1_data(data_L2)  
    print "]\n"  
    print_L1_code(data_L1)
```



# Beispiele (5)

## Python

```
File "quine.py", line 1
  File "quine.py", line 1
    ^
IndentationError: unexpected indent
```

- Abhängig vom Dateinamen
- Kein echtes Quine

# Shakespeare Programming Language (SPL)

## SPL

```
[Enter Romeo and Juliet]
Romeo:  You are as beautiful as the
        clearest summer's day.
Juliet:  You evil lying fatherless bastard.
Romeo:  Are you worse than a plague? If
        so, let us return to scene II.
Romeo:  Remember the difference between
        the king and his wolf.
Juliet:  Recall the self-reference seminar.
Romeo:  Open your heart.
Juliet:  Speak your mind.
[Exeunt]
```

## Pseudocode

```
J = 2
R = -8
if (J < -1):
    goto II
J.push(2)

R.pop()
print J
print chr(R)
```

# Shakespeare Quine

- Idee
  - Eindeutigen Satz für jeden ASCII-Wert
  - SPL-Funktion, die diese Sätze generieren kann
- Datendefinition
  - Stack-Variable auf die der Code geschichtet wird
  - Für jeden Buchstaben im Code

`Remember the sum of ...`

- Code
  - Gebe Präfix aus
  - Für jedes Element der Daten
    - Generiere Remember-Satz
  - Für jedes Element der Daten
    - Speak your mind.