Berechenbarkeit und Komplexität

Mächtigkeit von Programmiersprachen: WHILE- und LOOP-Programme

Prof. Berthold Vöcking
präsentiert von Prof. Joost-Pieter Katoen

21. November 2008

Turing-mächtige Programmiersprachen

Definition

Eine Programmiersprache wird als *Turing-mächtig* bezeichnet, wenn jede Funktion, die durch eine TM berechnet werden kann, auch durch ein Programm in dieser Programmiersprache berechnet werden kann.

Die Programmiersprache WHILE – Syntax

Elemente eines WHILE-Programms

- Variablen x_0 x_1 x_2 ...
- Konstanten -1 0 1
- Symbole ; $:=$ $+$ $\neq$
- Schlüsselwörter WHILE DO END

Induktive Definition – Induktionsanfang

Zuweisung

Für jedes $c \in \{-1, 0, 1\}$ ist die Zuweisung

$$x_i := x_j + c$$

ein WHILE-Programm.

Die Programmiersprache WHILE – Syntax

Induktive Definition – Induktionsschritte:

Hintereinanderausführung

Falls P_1 und P_2 WHILE-Programme sind, dann ist auch

$$P_1; P_2$$

ein WHILE-Programm.

WHILE-Konstrukt

Falls P ein WHILE-Programm ist, dann ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

ein WHILE-Programm.

Die Programmiersprache WHILE – Semantik

Ein While-Programm P berechnet eine k-stellige Funktionen der Form $f : \mathbb{N}^k \to \mathbb{N}$.

- Die Eingabe ist in den Variablen $x_1, \ldots, x_k$ enthalten.
- Alle anderen Variablen werden mit 0 initialisiert.
- Das Resultat eines WHILE-Programms ist die Zahl, die sich am Ende der Rechnung in der Variable x_0 ergibt.

- Programme der Form $x_i := x_j + c$ sind Zuweisungen des Wertes $x_j + c$ an die Variable x_i.
- In einem WHILE-Programm P_1; P_2 wird zunächst P_1 und dann P_2 ausgeführt.
- Das Programm WHILE $x_i \neq 0$ DO P END hat die Bedeutung, dass P solange ausgeführt wird, bis x_i den Wert 0 erreicht.

Beispiel eines WHILE-Programms

Was berechnet dieses WHILE-Programm?

WHILE $x_2 \neq 0$ DO
    $x_1 := x_1 + 1;$
    $x_2 := x_2 - 1$
END;
$x_0 := x_1$

Die Programmiersprache WHILE – Mächtigkeit

Satz

Die Programmiersprache WHILE ist Turing-mächtig.

Beweis: Es ist nicht schwierig zu zeigen, dass eine TM durch eine RAM mit konstant vielen Registern und eingeschränktem Befehlssatz

$$\text{LOAD, CLOAD, STORE, CADD, CSUB,}$$
$$\text{GOTO, IF } c(0) \neq 0 \text{ GOTO, END}$$

simuliert werden kann.

Wir müssen also nur noch zeigen, dass jede Funktion, die durch eine eingeschränkte RAM berechnet werden kann, auch durch ein WHILE-Programm berechnet werden kann.

Beweis Turing-Mächtigkeit von WHILE-Programmen

Sei Π ein beliebiges RAM-Programm mit eingeschränktem Befehlssatz, das aus ℓ Zeilen besteht und k Register für natürliche Zahlen benutzt.

Wir speichern den Inhalt von Register $c(i)$, für $0 \leq i \leq k$, in der Variable x_i des WHILE-Programms.

In der Variable x_{k+1} speichern wir zudem den Befehlszähler b der RAM ab.

Die Variable x_{k+2} verwenden wir, um eine Variable zu haben, die immer den initial gesetzen Wert 0 enthält.

Beweis Turing-Mächtigkeit von WHILE-Programmen

Die oben aufgelisteten RAM-Befehle werden nun in Form von konstant vielen Zuweisungen der Form $x_i := x_j + c$ mit $c \in \{-1, 0, 1\}$ implementiert.

Der RAM-Befehl LOAD i wird beispielsweise ersetzt durch

$$x_0 := x_i + 0;\ x_{k+1} := x_{k+1} + 1$$

Der RAM-Befehl CLOAD i wird analog ersetzt durch

$$x_0 := x_{k+2} + 0;\ \underbrace{x_0 := x_0 + 1;\ \ldots;\ x_0 := x_0 + 1;}_{i \text{ mal}}\ x_{k+1} := x_{k+1} + 1$$

Die RAM-Befehle STORE, CADD, CSUB und GOTO lassen sich leicht auf ähnliche Art realisieren.

Beweis Turing-Mächtigkeit von WHILE-Programmen

Der RAM-Befehl IF $c(0) \neq 0$ GOTO j ersetzen wir durch das WHILE-Programm:

$x_{k+1} := x_{k+1} + 1;$ $\quad (b := b + 1)$
$x_{k+3} := x_0 + 0;$ $\quad (help := c(0))$
WHILE $x_{k+3} \neq 0$ DO $\quad$ (while $help \neq 0$)
$\quad x_{k+1} := x_{k+2} + 0;$ $\underbrace{x_{k+1} := x_{k+1} + 1; \cdots + 1;}_{j \text{ mal}}$ $\quad (b := j)$
$\quad x_{k+3} := x_{k+2} + 0$ $\quad (help := 0)$
END $\quad$ (end of while)

Den RAM-Befehl END ersetzen wir durch das WHILE-Programm

$$x_{k+1} = 0 \ .$$

Beweis Turing-Mächtigkeit von WHILE-Programmen

Jede Zeile des RAM-Programms wird nun wie oben beschrieben in ein WHILE-Programm transformiert. Das WHILE-Programm für Zeile i bezeichnen wir mit P_i.

Wir betten P_i in ein WHILE-Programm P_i' mit der folgenden Semantik ein:

Falls $x_{k+1} = i$ dann führe P_i aus.

Wie kann man P_i' implementieren? – Hausaufgabe

Beweis Turing-Mächtigkeit von WHILE-Programmen

Nun fügen wir die WHILE-Programme $P'_1, \ldots, P'_\ell$ zu einem WHILE-Programm P zusammen:

$x_{k+1} := 1;$

WHILE $x_{k+1} \neq 0$ DO

$P'_1; \ldots; P'_\ell$

END

P berechnet dieselbe Funktion wie Π. $\square$

Die Programmiersprache LOOP – Erläuterung

Syntax

Änderung im Vergleich zu WHILE-Programmen:

Wir ersetzen das WHILE-Konstrukt durch ein LOOP-Konstrukt der folgenden Form:

$$\text{LOOP } x_i \text{ DO } P \text{ END} \quad ,$$

wobei die Variable x_i nicht in P vorkommen darf.

Semantik

Das Programm P wird x_i mal hintereinander ausgeführt.

Die Programmiersprache LOOP – Mächigkeit

Definition

Die durch LOOP-Programme berechenbaren Funktionen werden als *primitiv-rekursiv* bezeichnet.

Vermutung von Hilbert (1926): Die Klasse der primitiv rekursiven Funktionen stimmt mit der Klasse der rekursiven (berechenbaren) Funktionen überein.

Ackermann (1929): Diese Vermutung stimmt nicht!

Die Ackermann-Funktion – Definition

Definition

Die Ackermannfunktion $A : \mathbb{N}^2 \to \mathbb{N}$ ist folgendermaßen definiert:

$$
\begin{array}{lcll}
A(0,n) & = & n+1 & \text{für } n \geq 0 \\
A(m+1,0) & = & A(m,1) & \text{für } m \geq 0 \\
A(m+1,n+1) & = & A(m,A(m+1,n)) & \text{für } m,n \geq 0
\end{array}
$$

Die Ackermann-Funktion – Eigenschaften

Monotonie

- $A(m+1,n) > A(m,n)$
- $A(m,n+1) > A(m,n)$
- $A(m+1,n-1) \geq A(m,n)$ (Übungsaufgabe)

Wenn man den ersten Parameter fixiert ...

- $A(1,n) = n+2,$
- $A(2,n) = 2n+3,$
- $A(3,n) = 8 \cdot 2^n - 3,$
- $A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+2 \text{ viele Potenzen}} - 3,$

Bereits $A(4,2) = 2^{65536} - 3$ ist größer als die geschätzte Anzahl der Atome im Weltraum.

Wachstum der Variableninhalte in einem LOOP-Programm

Definition der Funktion F_P

- Sei P ein LOOP-Programm
- Seien $x_0, x_1, \ldots, x_k$ die Variablen in P.
- Wenn die Variablen initial die Werte $a = (a_0, \ldots, a_k) \in \mathbb{N}^{k+1}$ haben, dann sei $f_P(a)$ das $(k+1)$-Tupel der Variablenwerte nach Ausführung von P.
- Sei $|f_P(a)|$ die Summe der Einträge im $(k+1)$-Tupel $f_P(a)$.
- Wir definieren nun die Funktion $F_P : \mathbb{N} \to \mathbb{N}$ durch

$$F_P(n) = \max \left\{ |f_P(a)| \,\middle|\, a \in \mathbb{N}^{k+1} \text{ mit } \sum_{i=0}^{k} a_i \leq n \right\} .$$

Intuitiv beschreibt die Funktion F_P das maximale Wachstum der Variablenwerte im LOOP-Programm P.

Ackermannfunktion versus F_P

Wir zeigen nun, dass $F_P(n)$ für alle $n \in \mathbb{N}$ echt kleiner ist als $A(m, n)$, wenn der Parameter m genügend groß in Abhängigkeit von P gewählt wird.

Lemma

Für jedes LOOP-Programm P gibt es eine natürliche Zahl m, so dass für alle n gilt: $F_P(n) < A(m, n)$.

Beachte, für ein festes Programm P ist der Parameter m eine Konstante.

Beweis durch Strukturelle Induktion (Überblick)

Induktionsanfang

- Sei P von der Form $x_i := x_j + c$ für $c \in \{-1, 0, 1\}$.
- Wir werden zeigen: $F_P(n) < A(2, n)$.

Induktionsschritt (1. Art)

- Sei P von der Form $P_1; P_2$.
- Induktionsannahme: $\exists q \in \mathbb{N} : F_{P_1}(\ell) < A(q, \ell)$ und $F_{P_2}(\ell) < A(q, \ell)$.
- Wir werden zeigen: $F_P(n) < A(q + 1, n)$.

Induktionsschritt (2. Art)

- Sei P von der Form LOOP x_i DO Q END.
- Induktionsannahme: $\exists q \in \mathbb{N} : F_Q(\ell) < A(q, \ell)$.
- Wir werden zeigen: $F_P(n) < A(q + 1, n)$.

Beweis des Lemmas

Der Induktionsanfang

- Sei P von der Form $x_i := x_j + c$ für $c \in \{-1, 0, 1\}$.
- Dann gilt $F_P(n) \leq 2n + 1$.
- Somit folgt $F_P(n) < A(2, n)$.

Erläuterung: Vor Ausführung von P könnte gelten $x_j = n$ und alle anderen Variablen haben den Wert 0. Ferner könnte c den Wert 1 haben. Nach Ausführung von P gilt somit $x_i = n + 1$ und somit ist die Summe der Variableninhalte $x_i + x_j = 2n + 1$. Ein größeres Wachstum der Variableninhalte ist nicht möglich.

Beweis des Lemmas

Der Induktionsschritt (1. Art)

- Sei P von der Form $P_1; P_2$.
- Induktionsannahme: $\exists q \in \mathbb{N} : F_{P_1}(\ell) < A(q, \ell)$ und $F_{P_2}(\ell) < A(q, \ell)$.
- Somit gilt

$$F_P(n) \;\leq\; F_{P_2}(F_{P_1}(n)) \;<\; A(q, A(q, n)) \;\; .$$

- Wir verwenden die Abschätzung $A(q, n) \leq A(q+1, n-1)$.
- Es folgt

$$F_P(n) \;<\; A(q, A(q+1, n-1)) \;=\; A(q+1, n) \;\; .$$

Beweis des Lemmas

Der Induktionsschritt (2. Art)

- Sei P von der Form LOOP x_i DO Q END.
- Induktionsannahme: $\exists q \in \mathbb{N} : F_Q(\ell) < A(q, \ell)$.
- Sei $\alpha = \alpha(n)$ derjenige Wert für x_i der $F_P(n)$ maximiert.
- Dann gilt

$$F_P(n) \;\leq\; F_Q(F_Q(\ldots F_Q(F_Q(n-\alpha))\ldots)) + \alpha \;\;,$$

wobei die Funktion $F_Q(\cdot)$ hier α-fach ineinander eingesetzt ist.

Der Induktionsschritt (2. Art) – Fortsetzung

- Bisher haben wir gezeigt

$$F_P(n) \leq F_Q(F_Q(\ldots F_Q(F_Q(n-\alpha))\ldots)) + \alpha \ ,$$

  wobei die Funktion $F_Q(\cdot)$ hier α-fach ineinander eingesetzt ist.
- Aus der Induktionsannahme folgt $F_Q(\ell) \leq A(q,\ell) - 1$.
- Dies wenden wir auf die äußerste Funktion F_Q an und erhalten

$$F_P(n) \leq A(q, F_Q(\ldots F_Q(F_Q(n-\alpha))\ldots)) + \alpha - 1 \ .$$

- Wiederholte Anwendung liefert

$$\begin{aligned} F_P(n) &\leq A(q, A(q, \ldots A(q, A(q, n-\alpha))\ldots)) \\ &\leq A(q, A(q, \ldots A(q, A(q+1, n-\alpha))\ldots)) \ . \end{aligned}$$

Beweis des Lemmas

Der Induktionsschritt (2. Art) – Fortsetzung

- Bisher haben wir gezeigt

$$F_P(n) \leq A(q, A(q, \ldots A(q, A(q+1, n-\alpha))\ldots)) \ .$$

- Der Definition der Ackermannfunktion entnehmen wir $A(q+1, y+1) = A(q, A(q+1, y))$.
- Auf die innere Verschachtelung angewendet ergibt sich

$$F_P(n) \leq A(q, A(q, \ldots A(q+1, n-\alpha+1)\ldots)) \ ,$$

  wobei die Schachtelungstiefe nur noch $\alpha - 1$ ist.
- Nach weiteren $\alpha - 2$ vielen Anwendungen, folgt

$$F_P(n) \leq A(q+1, n-1) < A(q+1, n) \ .$$

□

Ackermannfunktion nicht LOOP-berechenbar

Satz

Die Ackermannfunktion ist nicht primitiv-rekursiv.

Beweis:

- Angenommen die Ackermannfunktion ist durch ein LOOP-Programm berechenbar.
- Dann sei P ein LOOP-Programm für die Fkt $B(n) = A(n, n)$.
- Es gilt $B(n) \leq F_P(n)$.
- Aus dem Lemma folgt, es gibt $m \in \mathbb{N}$ mit $F_P(n) < A(m, n)$.
- Für $m = n$ gilt somit

$$B(n) \leq F_P(n) < A(m, n) = A(n, n) = B(n) .$$

- Widerspruch! Also folgt der Satz. □

Schlussfolgerung

Da die Ackermannfunktion durch eine TM berechenbar ist, folgt

Korollar

Die Klasse der primitiv-rekursiven Funktionen ist eine echte Teilmenge der rekursiven Funktionen.