Vorlesung 11
WHILE-Programme

Wdh.: Das Postsche Korrespondenzproblem

Das Postsche Korrespondenzproblem (PKP) ist eine Art Puzzle aus Dominos.
Eine Instanz ist zum Beispiel

$$K = \left\{ \begin{bmatrix} b \\ \hline ca \end{bmatrix}, \begin{bmatrix} a \\ \hline ab \end{bmatrix}, \begin{bmatrix} ca \\ \hline a \end{bmatrix}, \begin{bmatrix} abc \\ \hline c \end{bmatrix} \right\} .$$

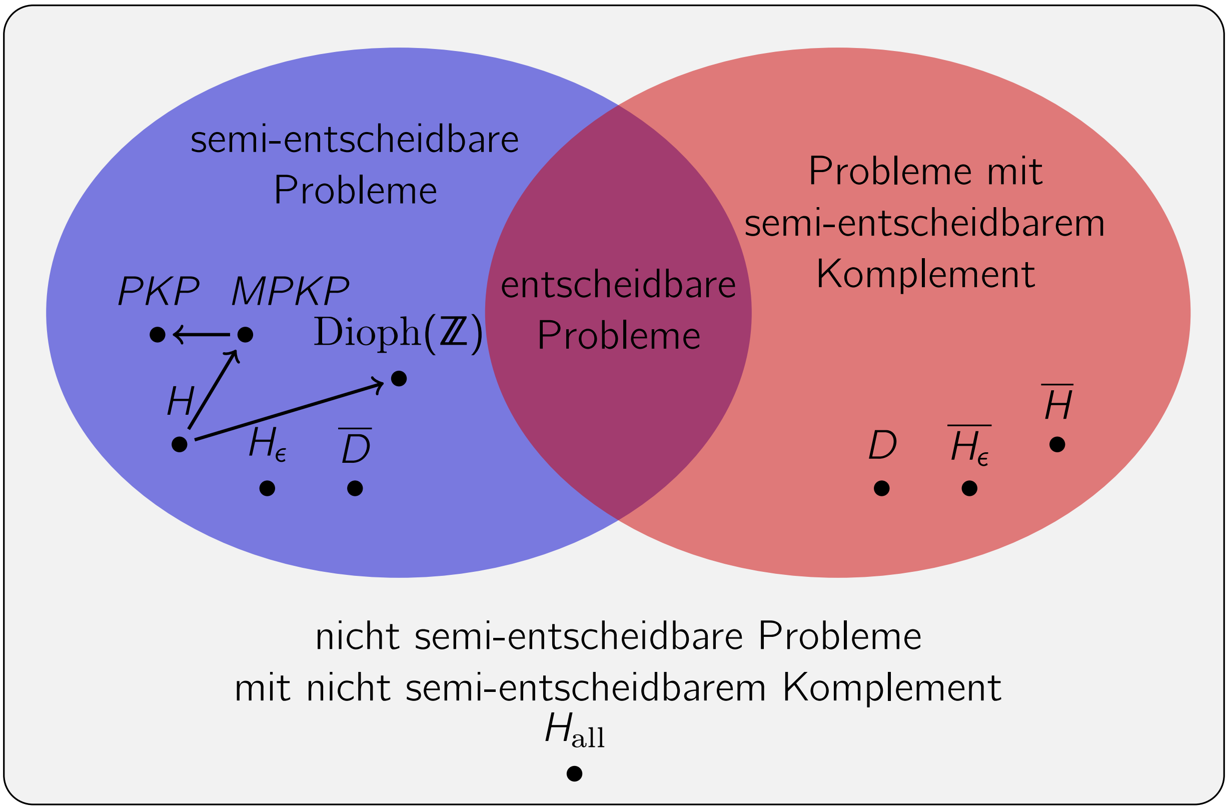
Eine Lösung ist

$$\begin{bmatrix} a \\ \hline ab \end{bmatrix} \begin{bmatrix} b \\ \hline ca \end{bmatrix} \begin{bmatrix} ca \\ \hline a \end{bmatrix} \begin{bmatrix} a \\ \hline ab \end{bmatrix} \begin{bmatrix} abc \\ \hline c \end{bmatrix} .$$

Satz

Das PKP ist nicht entscheidbar.

Wdh.: Berechenbarkeitslandschaft



Wdh.: Simulation einer TM durch Dominos

δ	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(\overline{q}, 1, N)$
q_1	$(q_2, 0, R)$	$(q_1, 1, R)$	$(\overline{q}, 1, N)$
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	(q_2, B, R)

Wird simuliert durch:

Startdomino: $\left[\frac{\#}{\#\#q_0 0011\#}\right]$. Kopierdominos: $\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{B}{B}\right], \left[\frac{\#}{\#}\right]$.

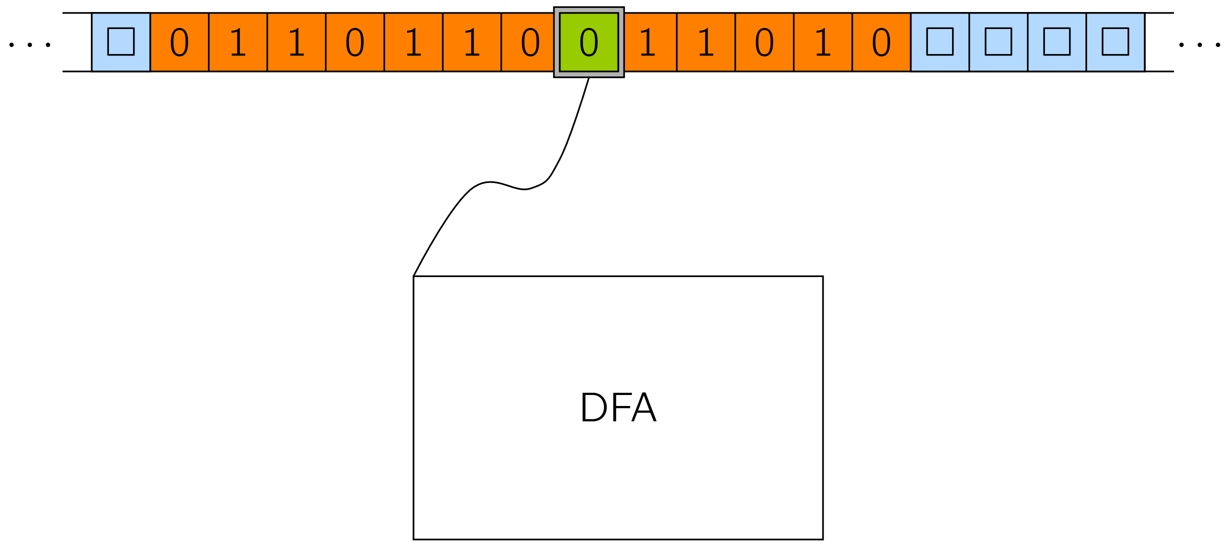
Überführungsdominos: $\left[\frac{q_0 0}{0 q_0}\right], \left[\frac{q_0 1}{1 q_1}\right], \left[\frac{q_0 B}{\overline{q} 1}\right], \left[\frac{q_1 0}{0 q_2}\right], \left[\frac{q_1 1}{1 q_1}\right], \left[\frac{q_1 B}{\overline{q} 1}\right], \left[\frac{q_2 0}{0 q_2}\right], \left[\frac{q_2 1}{1 q_2}\right], \left[\frac{q_2 B}{B q_2}\right]$

Spezielle Überführungsdominos: $\left[\frac{q_0 \#}{\overline{q} 1 \#}\right], \left[\frac{q_1 \#}{\overline{q} 1 \#}\right]$

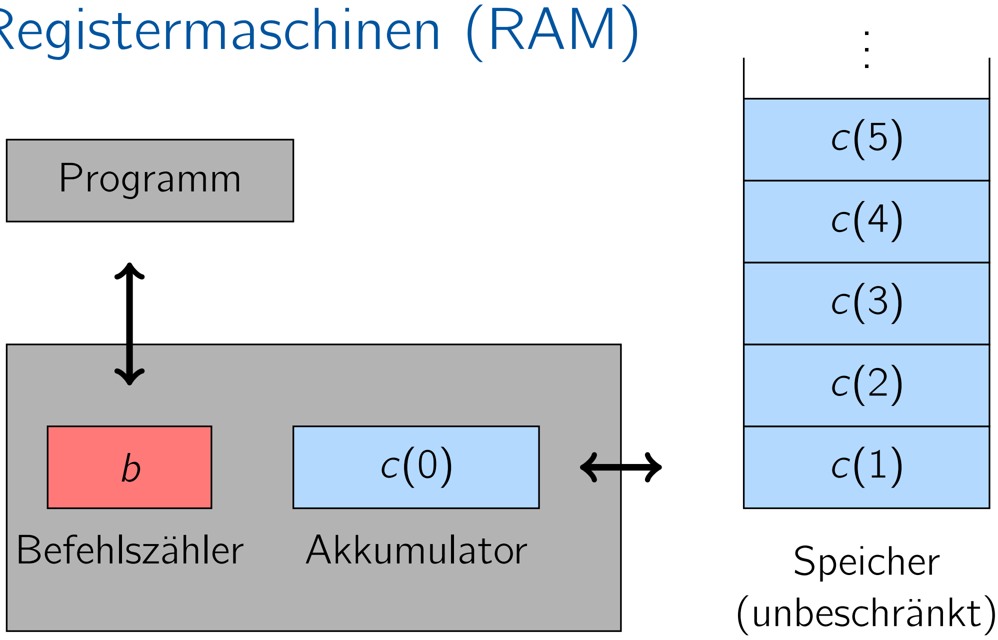
Löschdominos: $\left[\frac{\overline{q} 0}{\overline{q}}\right], \left[\frac{\overline{q} 1}{\overline{q}}\right], \left[\frac{\overline{q} B}{\overline{q}}\right], \left[\frac{0 \overline{q}}{\overline{q}}\right], \left[\frac{1 \overline{q}}{\overline{q}}\right], \left[\frac{B \overline{q}}{\overline{q}}\right]$

Abschlussdomino: $\left[\frac{\# \overline{q} \#\#}{\#}\right]$

Wdh.: Deterministische Turingmaschine (TM bzw. DTM)



Wdh.: Registermaschinen (RAM)



Befehlssatz:

LOAD, STORE, ADD, SUB, MULT, DIV
INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT, INDDIV
CLOAD, CADD, CSUB, CMULT, CDIV
GOTO, IF c(0)?x THEN GOTO j (wobei ? aus {=,<,<=,>,>=} ist),
END

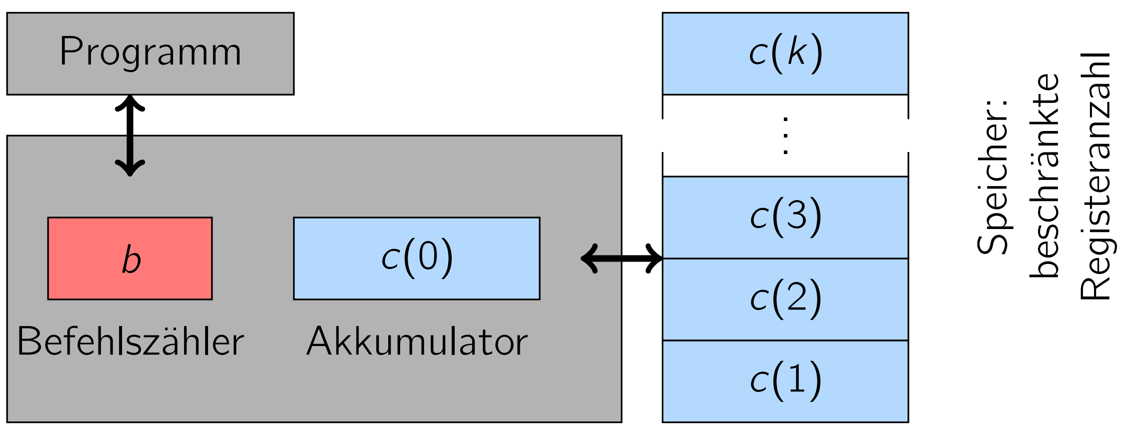
Turing-mächtige Rechnermodelle

Definition

Ein Rechnermodell wird als Turing-mächtig bezeichnet, wenn jede Funktion, die durch eine TM berechnet werden kann, auch durch dieses Rechnermodell berechnet werden kann.

- Da die Registermaschine die Turingmaschine simulieren kann, ist sie Turing-mächtig.
- Ebenso kann man zeigen, dass das Spiel des Lebens Turing-mächtig ist.

Eingeschränkte Registermaschine (eingeschränkte RAM)



Befehlssatz:

LOAD, STORE

CLOAD, CADD, CSUB

GOTO,
IF c(0) $\neq$ 0 THEN GOTO j,
END

Man kann zeigen, dass die eingeschränkte RAM Turing-mächtig ist.

Wdh.: Simulation TM durch RAM

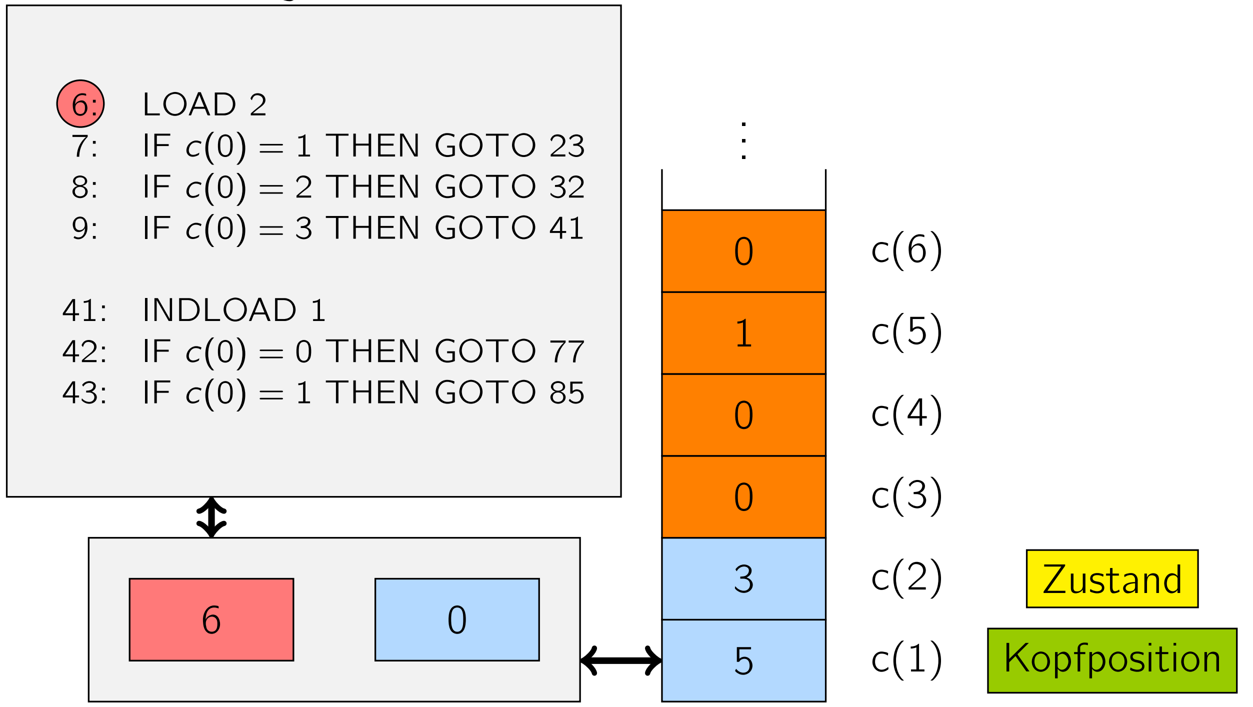
simulierte Turingmaschine M

B	0	0	**1**	0	B	B	B	…

δ	0	1	B
q_1			
q_2			
q_3		$(q_2, 0, R)$	

q_3

simulierende Registermaschine



Simulation TM durch eingeschränkte RAM

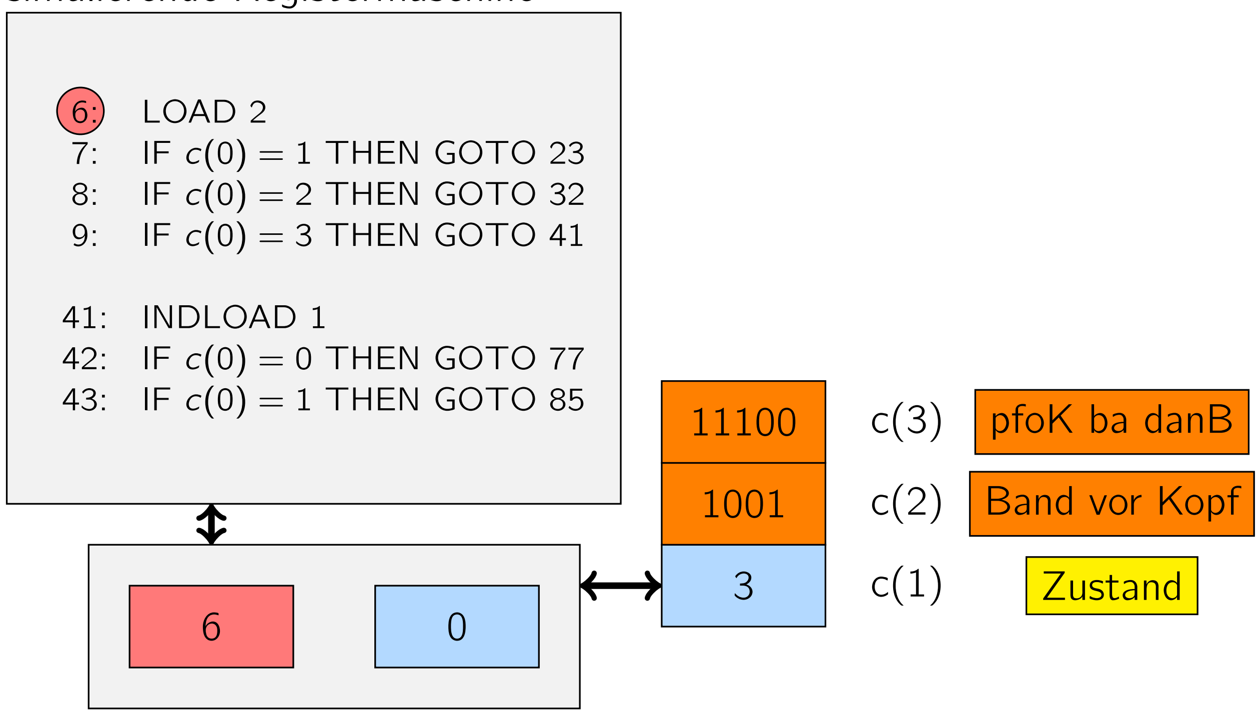
simulierte Turingmaschine M

…	B	0	0	1	**0**	0	1	1	B	B	B	…

δ	0	1	B
q_1			
q_2			
q_3		$(q_2, 0, R)$	

q_3

simulierende Registermaschine



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.

Welche Elemente benötigt eine Programmiersprache, um Turing-mächtig zu sein?

Die Programmiersprache WHILE – Syntax

Elemente eines WHILE-Programms

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

Die Programmiersprache WHILE – Syntax

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 Funktion der Form $f : \mathbb{N}^k \to \mathbb{N}$.

- Die Eingabe ist in den Variablen $x_1, \dots, x_k$ 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 (wobei $0 + (-1) = 0$).
- 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:

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

Da die eingeschränkte RAM Turing-mächtig ist, ist somit auch die Programmiersprache WHILE Turing-mächtig.

Sei Π ein beliebiges Programm der eingeschränkten RAM. Sei ℓ die Anzahl der Zeilen in Π und k die Anzahl der verwendeten Register.

Beweis Turing-Mächtigkeit von WHILE-Programmen

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 eingeschränkten RAM ab.

Die Variable x_{k+2} verwenden wir als Hilfsvariable, die immer mit dem Wert 0 initialisiert ist.

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

Beweis Turing-Mächtigkeit von WHILE-Programmen

RAM vs. **WHILE**

RAM:
- LOAD, STORE
- CLOAD, CADD, CSUB, GOTO
- IF c(0) $\neq$ 0 GOTO
- END

WHILE:
- $x_i \; := \; x_j + c$ für $c \in \{-1, 0, 1\}$
- $P_1; P_2$
- WHILE $x_i \neq 0$ DO P END

Der RAM-Befehl LOAD i wird beispielsweise ersetzt durch

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

Beweis Turing-Mächtigkeit von WHILE-Programmen

RAM vs. **WHILE**

RAM:
- LOAD, STORE ✓
- CLOAD, CADD, CSUB, GOTO
- IF c(0) $\neq$ 0 GOTO
- END

WHILE:
- $x_i \; := \; x_j + c$ für $c \in \{-1, 0, 1\}$
- $P_1; P_2$
- WHILE $x_i \neq 0$ DO P END

Der RAM-Befehl CLOAD i wird ersetzt durch

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

Beweis Turing-Mächtigkeit von WHILE-Programmen

RAM vs. **WHILE**

- LOAD, STORE ✓
- CLOAD, CADD, CSUB, GOTO ✓
- IF c(0) $\neq$ 0 GOTO
- END

- $x_i := x_j + c$ für $c \in \{-1, 0, 1\}$
- $P_1; P_2$
- WHILE $x_i \neq 0$ DO P END

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

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

Beweis Turing-Mächtigkeit von WHILE-Programmen

RAM vs. **WHILE**

- LOAD, STORE ✓
- CLOAD, CADD, CSUB, GOTO ✓
- IF c(0) $\neq$ 0 GOTO ✓
- END

- $x_i := x_j + c$ für $c \in \{-1, 0, 1\}$
- $P_1; P_2$
- WHILE $x_i \neq 0$ DO P END

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.

Aus P_i konstruieren wir nun ein WHILE-Programm P'_i mit der folgenden Semantik:

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

Übungsaufgabe: Implementiere das WHILE-Programm P'_i mit Unterprogramm P_i.

Beweis Turing-Mächtigkeit von WHILE-Programmen

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

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

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

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

END

P berechnet die gleiche Funktion wie Π. $\square$

Ausblick: Die Programmiersprache LOOP

Syntax

Änderung im Vergleich zu WHILE-Programmen:

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

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

wobei die Variable x_i nicht in P vorkommen darf.

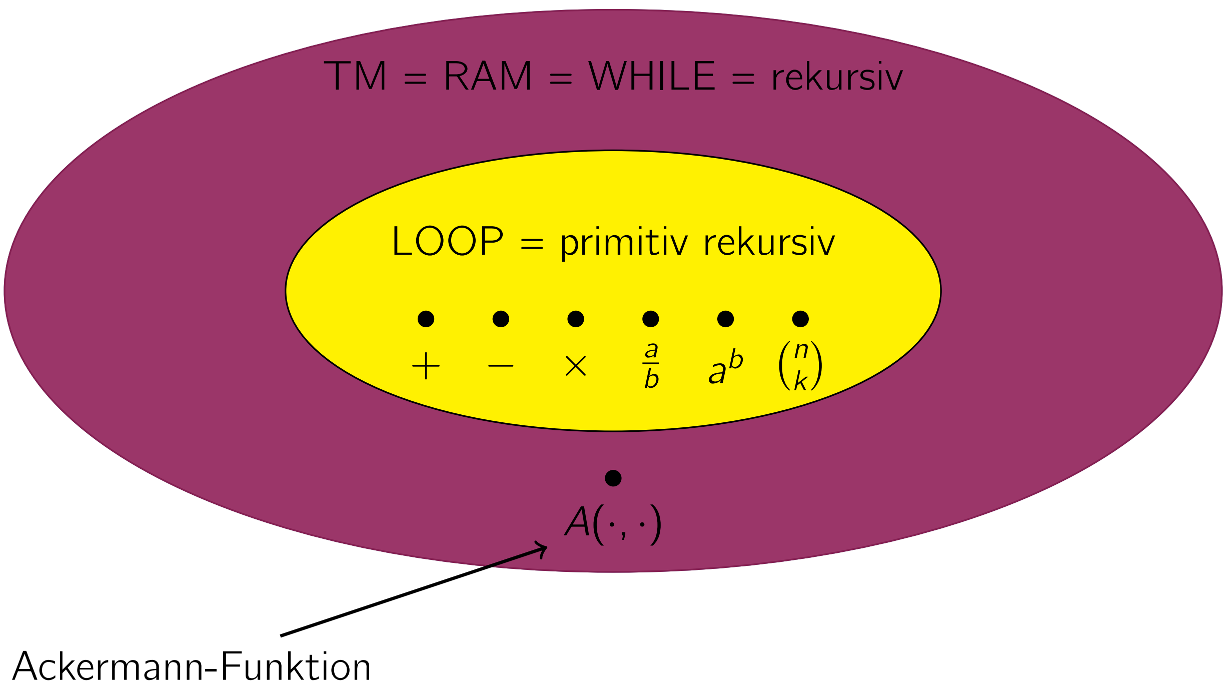
Semantik

Das Programm P wird x_i mal hintereinander ausgeführt.

Frage

Sind LOOP-Programme Turing-mächtig?

Ausblick: Mächtigkeit von LOOP-Programmen



Ausblick: Ackermann-Funktion

Definition

Die Ackermann-Funktion $A : \mathbb{N}^2 \to \mathbb{N}$ ist folgendermaßen definiert:

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

Ackermann-Funktion – Beispiele

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

Ein paar Beispiele:

$A(1,0) = A(0,1) = 2$

$A(1,1) = A(0,A(1,0)) = A(1,0) + 1 = 3$

$A(1,2) = A(0,A(1,1)) = A(1,1) + 1 = 4$

Allgemein: $A(1,n) = 2 + n$

$A(2,0) = A(1,1) = 3$

$A(2,1) = A(1,A(2,0)) = A(2,0) + 2 = 5$

Allgemein: $A(2,n) = 2n + 3$

$A(3,0) = A(2,1) = 5$

$A(3,1) = A(2,A(3,0)) = 2A(3,0) + 3$

Allgemein: $A(3,n) = 8 \cdot 2^n - 3$

Ausblick: Ackermann-Funktion

Wenn man den ersten Parameter fixiert ...

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

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