Anhang zum Lehrbuch über Datenstrukturen und Algorithmen

Hans Ulrich Simon

10. April 2018

Zusammenfassung

Der Stoff der Vorlesung *Datenstrukturen* im Sommersemester 2018 deckt sich zum großen Teil mit den Kapiteln 1 bis 6 (sowie evtl. 8) des Buches *Datenstrukturen und Algorithmen* von *Hartmut Güting* und *Stefan Dieker*, erschienen im *Teubner Verlag*. Wir werden allerdings, anders als in dem Lehrbuch von Güting und Dieker, auf die Angabe von Java-Code für die einzelnen Algorithmen verzichten. In diesem kleinen Manuskript stellen wir das Rechenmodell der Random Access Maschine (= RAM) vor, welches in dem Lehrbuch nicht explizit behandelt wird.

1 RAM: Random Access Machine

Die folgende Beschreibung der sogenannten *Random Access Maschine (RAM)* ist weitgehend aus dem Buch *The Design and Analysis of Computer Algorithms* von Ahu, Hopcraft und Ullman übernommen. Die RAM ist ein mathematisches Maschinenmodell für einen sequentiellen Rechner, das eine präzise Definition von *Rechenzeit* bzw. *Speicherplatz* erlaubt. Die RAM ist enger mit einem realen Rechner verwandt als zum Beispiel die Turing-Maschine. I. A. wird die Rechenzeit eines Algorithmus auf einer RAM als ein sehr brauchbares Maß für die Größenordnung der Rechenzeit auf einem realen Rechner betrachtet. Ziel diese Abschnittes ist, die RAM sauber zu definieren und so die Analyse der Rechenzeit von Algorithmen auf ein solides Fundament zu stellen.

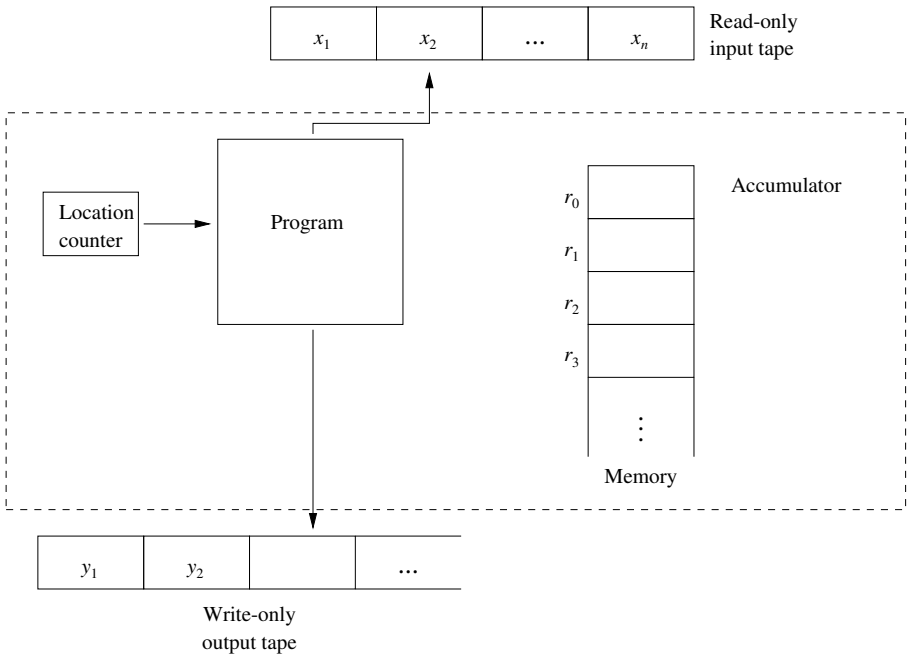

Abbildung 1: „Physikalische Gestalt“ der RAM

Wir beginnen mit einer anschaulichen Beschreibung der RAM, die in Abbildung 1 nachvollzogen werden kann. Das Programm besteht aus einer endlichen Folge von elementaren Instruktionen. Der Location Counter (Befehlszähler) zeigt stets auf die aktuelle Instruktion. Das Memory (Rechenspeicher) ist eine unendliche Folge $(r_i)_{i\geq 0}$ von Registern. Jedes Register kann eine beliebige ganze Zahl speichern. r_0 ist ein Spezialregister, genannt der Akkumulator. Die Eingabe $(x_1, \ldots, x_n) \in \mathbb{Z}^n$ steht auf dem „Read-only“ Eingabeband. Die Ausgabe $(y_1, y_2, \ldots) \in \mathbb{Z}^*$ muß auf das „Write-only“ Ausgabeband geschrieben werden.

Die folgende Tabelle zeigt, welche Instruktionstypen in dem Programm einer RAM zugelassen sind:

Operation code		Address	Operation code		Address
1.	LOAD	operand	7.	READ	operand
2.	STORE	operand	8.	WRITE	operand
3.	ADD	operand	9.	JUMP	label
4.	SUB	operand	10.	JGTZ	label
5.	MULT	operand	11.	JZERO	label
6.	DIV	operand	12.	HALT	

Eine konkrete elementare Instruktion entsteht aus diesen Instruktionstypen, wenn wir für „operand“ einen konkreten Operanden einsetzen und für „label“ ein konkretes Label. Die folgende Tabelle listet die zulässigen Operanden auf und gibt jeweils den davon repräsentierten Wert an:

Operand	sein Wert
$= i$	Konstante $i \in \mathbb{Z}$
i	Inhalt von r_i (direkte Adressierung)
$*i$	Inhalt von r_j, wenn j Inhalt von r_i ist (indirekte Addressierung)

Wir legen für $a \in \{= i, i, *i\}$ die folgende Notation fest:

$$c(i) = \text{Inhalt von } r_i \ , \ v(= i) = i \ , \ v(i) = c(i) \text{ und } v(*i) = c(c(i)) \ .$$

Der Buchstabe „c“ steht dabei für „content (Inhalt)“ und der Buchstabe „v“ für „value (Wert)“. Mit dieser Notation ist $v(a)$ der vom Operanden a repräsentierte Wert.

Ein *Label* ist eine symbolische Marke, die im Programm auf eindeutige Weise eine Instruktion markiert.

Die folgende Tabelle klärt die Bedeutung der Instruktionen (in einem Pseudocode, der aus sich heraus verständlich sein sollte):

1.	LOAD a	$c(0) \leftarrow v(a)$
2.	STORE i	$c(i) \leftarrow c(0)$
	STORE $*i$	$c(c(i)) \leftarrow c(0)$
3.	ADD a	$c(0) \leftarrow c(0) + v(a)$
4.	SUB a	$c(0) \leftarrow c(0) - v(a)$
5.	MULT a	$c(0) \leftarrow c(0) \cdot v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
7.	READ i	$c(i) \leftarrow$ aktueller Eingabeparameter
	READ $*i$	$c(c(i)) \leftarrow$ aktueller Eingabeparameter
8.	WRITE a	Mache $v(a)$ zum nächsten Ausgabeparameter
9.	JUMP b	Sprung zu der Marke „b“
10.	JGTZ b	Sprung zu der Marke „b“ sofern $c(0) > 0$
11.	JZERO b	Sprung zu der Marke „b“, sofern $c(0) = 0$
12.	HALT	Anhalten des Programmlaufes

Der erste Befehl besagt in Worten: lade die Zahl $v(a)$ in den Akkumulator. Analog besagt der zweite Befehl: speichere den Inhalt des Akkumulators im Register i bzw. (bei indirekter Adressierung mit „$*i$“) im Register $c(i)$. Die Befehle ADD, SUB, MULT und DIV erlauben die arithmetische Verknüpfung zweier Operanden, wobei einer der beiden Operanden stets der Akkumulator ist. Das Ergebnis wird im Akkumulator abgelegt. Beachte, dass es sich bei DIV um die ganzzahlige Division handelt. Mit READ kann eine Liste von (ganzzahligen) Eingabeparametern der Reihe nach in Register eingelesen werden. WRITE ist der entsprechende Schreibbefehl. JUMP und JGTZ[1] bzw. JZERO[2] realisiert den den unbedingten und zwei Varianten eines bedingten Sprungbefehls.

Bemerkung 1.1 *Falls kein Sprungbefehl vorliegt oder die Bedingung zum bedingten Sprungbefehl nicht erfüllt ist, wird der Befehlszähler um 1 inkrementiert.*

Wir kommen jetzt zu dem zentralen Begriff der Konfiguration (=Momentaufnahme) einer RAM. Intuitiv müssen in eine Konfiguration alle Informationen aufgenommen werden, die erlauben eine Rechnung temporär zu stoppen und später wieder weiterzuführen. Formal ergibt sich die folgende Definition. Die *Konfiguration* einer RAM besteht aus folgenden Komponenten:

[1] Jump if Greater Than Zero
[2] Jump if equal to Zero

- Programm $\Pi = (\pi_1, \ldots, \pi_p)$
- Wert des Befehlszählers
- Inhalt $c(i)$ von r_i für alle $i \geq 0$
- Eingabeparameter $x = (x_1, \ldots, x_n) \in \mathbb{Z}^n$
- Index i des aktuell gelesenen Eingabeparameters x_i
- Bisher produzierte Ausgabe $(y_1, \ldots, y_j)$ {leere Sequenz ε, falls $j = 0$}

Beim Start der Rechnung befindet sich die RAM in der *Anfangskonfiguration*, die die folgende Form hat:

- Programm $\Pi = (\pi_1, \ldots, \pi_p)$,
- Befehlszähler mit Wert 1 (Zeiger auf die erste Instruktion π_1)
- alle Register auf Null gesetzt
- Eingabe $x = (x_1, \ldots, x_n) \in \mathbb{Z}^n$
- $i = 1$ (erster Eingabeparameter einlesbar)
- leere Sequenz ε als bisherige Ausgabe

Danach wird das Programm Π (anfangs also die Instruktion π_1) gemäß der Semantik der Instruktionen abgearbeitet. Die Ausführung einer Instruktion führt die aktuelle Konfiguration in eine eindeutige *direkte Folgekonfiguration* über. Hierdurch entsteht eine (evtl. unendliche) Folge von Konfigurationen. Die Rechnung (=Konfigurationsfolge) bricht ab, falls Instruktion „HALT“ zur Ausführung kommt. Eine Konfiguration, bei der der Befehlszähler auf die Instruktion „HALT“ zeigt, heißt *Endkonfiguration*.

Wir illustrieren die vorangehenden Definitionen an einem Beispiel, nämlich an der Berechnung von

$$f(n) = \begin{cases} 0 & \text{falls } n \leq 0 \\ n^n & \text{falls } n \geq 1 \end{cases} .$$

Bei den Kommentaren zu dem folgenden RAM-Programm (die Spalte „Effekt“) identifizieren wir ein Register mit seinem Inhalt. Wir schreiben zum Beispiel $r_i := r_j$ statt $c(i) \leftarrow c(j)$. Dies ist die in Programmiersprachen übliche Schreibweise für die Wertzuweisung an eine Variable.

RAM-Programm			Effekt
	READ	1	Einlesen von n in r_1
	LOAD	1	**if** $r_1 \leq 0$ **then write** 0
	JGTZ	pos	
	WRITE	=0	
	JUMP	endif	
pos:	LOAD	1	$r_2 := r_1$
	STORE	2	
	LOAD	1	$r_3 := r_1 - 1$
	SUB	=1	
	STORE	3	
while:	LOAD	3	**while** $r_3 > 0$ **do**
	JGTZ	continue	
	JUMP	endwhile	
continue:	LOAD	2	$r_2 := r_2 * r_1$
	MULT	1	
	STORE	2	
	LOAD	3	$r_3 := r_3 - 1$
	SUB	=1	
	STORE	3	
	JUMP	while	
endwhile:	WRITE	2	**write** r_2
endif:	HALT		

Man verifiziert leicht induktiv, dass (im Falle $n \geq 1$) nach $k - 1 \leq n - 1$ Iterationen der while-Schleife gilt:

$$c(1) = n \ , \ c(2) = n^k \text{ und } c(3) = n - k \ .$$

Offensichtlich wird die Schleife nach $n-1$ Durchläufen verlassen (weil dann r_3 den Inhalt 0 hat). Das Register r_2 hat zu diesem Zeitpunkt den gewünschten Inhalt n^n.

Uniformes Kostenmaß. Die *Rechenzeit eines Programms* Π *auf Eingabe* $x \in \mathbb{Z}^n$ *(bezüglich des uniformen Kostenmaßes)* ist definiert als die Anzahl der Rechenschritte bis zum Abbruch der Rechnung (bzw. unendlich, falls die Rechnung nicht abbricht). In anderen Worten: wenn aus Programm Π und Eingabe x die Konfigurationsfolge $K_0, K_1, \ldots, K_T$ mit Endkonfiguration K_T

resultiert, dann gibt T die Rechenzeit von Π auf Eingabe x an. Wir sagen eine RAM mit Programm Π ist *$T(n)$-zeitbeschränkt (bezüglich des uniformen Kostenmaßes*, wenn für jedes $n \geq 1$ die Rechenzeit auf allen Eingaben $x \in \mathbb{Z}^n$ nach oben durch $T(n)$ beschränkt ist. In der Regel interessiert uns nur die asymptotische Größenordnung der Zeitschranke: Π ist *$O(T(n))$-zeitbeschränkt (bezüglich des uniformen Kostenmaßes* bedeutet dann, dass eine Zeitschrake der Größenordnung $O(T(n))$ existiert.

Logarithmisches Kostenmaß. Beim logarithischen Kostenmaß gelten analoge Definitionen, aber jede Instruktion wird gewichtet mit der maximalen Bitlänge des Wertes einer der beteiligten Register.

Wir werden uns in der Regel auf das uniforme Kostenmaß beziehen. Es sei allerdings kurz angemerkt, dass dieses Maß „unfair“ ist (also die reale Rechenzeit unterschätzt), wenn die im Laufe der Rechnung auftretenden Zahlen sehr lang werden. Das uniforme Kostenmaß soll im Prinzip nur verwendet werden, wenn die binäre Kodierungslänge der aus der Rechnung resultierenden Zahlen polynomiell in der gesamten Kodierungslänge der Eingabeparameter $x_1, \ldots, x_n$ beschränkt ist. Ist dies nicht der Fall sollte das logarithmische Kostenmaß verwendet werden.

Diskussion des Maschinenmodelles RAM. Man könnte einwenden, dass die RAM im Vergleich zu modernen realen Rechnern ein zu primitives Modell ist. Die Implementierung eines Algorithmus auf einer RAM könnte evtl. eine deutlich andere Rechenzeit beanspruchen als die Implementierung des selben Algorithmus auf einem realen Rechner. Dem kann man aber entgegenhalten, dass der Code einer RAM überraschend ähnlich ist zu dem 3-Adress-Code, den Compiler ereugen, wenn sie Programme höherer Programmiersprachen in einen maschinennahen Code übersetzen. Die im RAM-Modell erbrachte Laufzeitanalyse wird daher nicht nur von Theoretikern sondern auch von Praktikern als (in der Regel) aussagekräftig eingestuft.

Wir beschließen unsere Ausführungen mit dem folgenden Resultat (ohne Beweis):

Bemerkung 1.2 *RAM's und deterministiche Turing-Maschinen sind polynomiell verknüpft*[3]*, wenn wir bei RAMs das logarithmische Kostenmaß verwenden (oder das uniforme Kostenmaß bei fairem Gebrauch).*

[3]d.h., es existieren wechselseitige Simulationen, die die Rechenzeit jeweils nur polynomiell anwachsen lassen