Gegenseitige Simulation von Datenstrukturen

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart als Habilitationsschrift genehmigte Abhandlung

Vorgelegt von
Holger Petersen

Stuttgart, 11. Dezember 2002

Inhaltsverzeichnis

Danksagung

An erster Stelle müssen Amir M. Ben-Amram (Tel-Aviv) und Mike Robson (Bordeaux) dankend erwähnt werden, denn die in den Kapiteln 3 und 6 dokumentierten Ergebnisse beruhen auf der fruchtbaren Zusammenarbeit mit ihnen. Auch zu anderen Teilen der Arbeit haben sie wertvolle Hinweise geliefert.

Ganz besonders möchte ich Volker Diekert für zahlreiche Anregungen danken. Ihm, Martin Dietzfelbinger und Klaus-Jörn Lange danke ich für die Erstellung der Gutachten und hilfreiche Hinweise. Meinen Kolleginnen und Kollegen der Theorie-Abteilungen in Stuttgart danke ich für ihr Interesse an meiner Arbeit. Holger Austinat verdanke ich konstruktive Bemerkungen zu einer frühen Version von Kapitel 8. Franz-Josef Brandenburg, Jeff Shallit und Pierre McKenzie danke ich für Kommentare zum Inhalt von Kapitel 3. Paul Levi danke ich für Korrekturhinweise.

Während meines Aufenthaltes in Israel waren für mich zahlreiche Diskussionen mit Omer Berkman, Boris Epstein, Yana Kortsarts und Michal Parnas sehr hilfreich. Viele praktische Ratschläge erhielt ich von Shmuel Tyszberowicz.

Schließlich danke ich K. Inoue für Kopien verschiedener Arbeiten über Rebound-Automaten.

Teile der hier dargestellten Forschung wurden am Laboratoire Bordelais de Recherche en Informatique (LaBRI) der Universität Bordeaux 1 und am Academic College of Tel-Aviv-Yaffo durchgeführt. Für die mir entgegengebrachte Gastfreundschaft und die vielfältige Unterstützung bedanke ich mich sehr herzlich.

Das französisch-deutsche Projekt PROCOPE unterstützte zwei Reisen nach Bordeaux. Der Aufenthalt in Israel wurde durch die Deutsche Akademie der Naturforscher Leopoldina (Förderungsnummer BMBF-LPD 9901/8-1 des Bundesministeriums für Bildung und Forschung) ermöglicht, wofür ich mich sehr herzlich bedanke. Auch wurde die Teilnahme an der Konferenz MFCS 2000 in Bratislava, bei der einige der in dieser Arbeit enthaltenen Ergebnisse präsentiert wurden, in besonderer Weise unterstützt. Selbstverständlich liegt die Verantwortung für den Inhalt bei mir.

Kapitel 1

Einführung

Die vorliegende Arbeit stellt einige Ergebnisse zusammen, welche das Verhältnis verschiedener Berechenbarkeitsmodelle zueinander betreffen. Hierbei wird einerseits der Zusatzaufwand (im Bezug auf die Zeitkomplexität) bei der gegenseitigen Simulation solcher Modelle untersucht. Andererseits werden untere Schranken bewiesen oder auch die Unmöglichkeit einer Simulation. Diese Untersuchungen lassen sich einem Bereich zuordnen, der als konkrete Komplexitätstheorie bezeichnet wird.

Die Kapitel 3, 4 und 5 untersuchen zeiteffiziente Simulation unter Verwendung von Datenstrukturen wie Queues und Kellern. Eine Übersicht der bekannten Beziehungen wird in Kapitel 2 gegeben.

Die Kapitel 6, 7 und 8 vergleichen Modelle, die Sprachen mit logarithmischer Platzkomplexität akzeptieren.

Der erste dieser Vergleiche erbringt als Hauptergebnis eine optimale Simulation von Zeigern auf einer nicht veränderbaren Eingabe, die vorwärts oder rückwärts bewegt werden können, durch solche mit nur einer Bewegungsrichtung, wobei Zuweisungen von Zeigerwerten erlaubt sind.

Kapitel 7 zeigt, dass bei der Betrachtung beschränkter Sprachen für deterministische Berechnungen Zähler und Köpfe (Zeiger auf eine nicht veränderbare Eingabezeichenkette) äquivalent sind. Im Gegensatz dazu gilt diese Aussage nicht für unbeschränkte Eingabemengen.

Im Kapitel 8 wird ein Modell mit einer zweidimensionalen Eingabe untersucht. Hier ergibt sich, dass Nichtdeterminismus stärker als Determinismus ist, was ein offenes Problem löst. Weitere Ergebnisse betreffen den Vergleich mit Teilklassen der kontextfreien Sprachen.

Der Inhalt beruht in wesentlichen Teilen auf Veröffentlichungen, und zwar Kapitel 3 auf

Efficient simulations by queue machines. In K. G. Larsen, S. Skyum,

und G. Winskel, Hrsg., *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP), Aalborg, 1998*, Band 1443 in Lecture Notes in Computer Science, S. 884–895, 1998,

einer gemeinsamen Arbeit mit Mike Robson. Eine der Simulationen wurde in der Zwischenzeit verbessert.

Die folgende gemeinsame Arbeit mit Amir M. Ben-Amram enthält die in Kapitel 6 dargestellten Ergebnisse:

Backing up in singly linked lists. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, S. 780–788, 1999.

Die Ergebnisse von Kapitel 8 wurden in der folgenden Arbeit veröffentlicht:

Separation results for rebound automata. In M. Nielsen und B. Rovan, Hrsg., *Proceedings of the 25th Symposium on Mathematical Foundations of Computer Science (MFCS), Bratislava, 2000*, Band 1893 in Lecture Notes in Computer Science, S. 589–598, Berlin-Heidelberg-New York, 2000. Springer.

Die Hauptresultate aus Kapitel 5 wurden in der folgenden Arbeit veröffentlicht:

Stacks versus deques. In J. Wang (Hrsg.), *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON), Guilin, 2001*, Band 2108 in Lecture Notes in Computer Science, S. 218–227, Berlin-Heidelberg-New York, 2001. Springer.

Die Kapitel 4 und 7 enthalten bisher unveröffentlichtes Material.

Kapitel 2

Übersicht zeiteffizienter Simulationsresultate

Wir übernehmen im Bezug auf die untersuchten Datenstrukturen die Terminologie aus [39]. Hier verstehen wir unter Kellern, Queues und Deques (engl. „double-ended queues“) lineare Listen atomarer Symbole. Information kann gespeichert und (zusammen mit dem Löschen) an den Enden der Listen gelesen werden. Zusätzlich kann getestet werden, ob ein Speicher leer ist.

Genauer unterscheiden wir zwischen:

Keller: Alle Einfügungen (Push-Operationen) und Löschungen (Pop-Operationen) werden an einem Ende der Liste ausgeführt.

Queue: Alle Einfügungen werden an einem Ende der Liste ausgeführt, alle Löschungen am anderen Ende.

Ausgabebeschränkte Deque: Einfügungen können an beiden Enden der Liste ausgeführt werden, während das Löschen nur an einer Seite erlaubt ist.

Eingabebeschränkte Deque: Löschungen können an beiden Enden der Liste ausgeführt werden, während das Einfügen nur an einer Seite erlaubt ist.

Deque: Einfügungen und Löschungen können an beiden Enden der Liste ausgeführt werden.

Andere geläufige Bezeichnungen für die Datenstruktur Queue sind Warteschlange, Puffer oder FIFO-Speicher.

Im Vergleich zu [39] schließen wir jeden Zugriff auf das Innere der Liste aus.

Ein Band ist ein linearer Speicher, der Zugriff an einem Punkt erlaubt, welcher sich um jeweils eine Position verschieben lässt.

Der übliche Rahmen, in dem diese Datenstrukturen verglichen werden (siehe z.B. [45, 43]), ist sie unter der Kontrolle eines endlichen Programms (auch „endliche Kontrolle“ genannt) ohne weiteren Speicher arbeiten zu lassen. Dieses Programm greift auf ein separates Einweg-Eingabeband zu und akzeptiert eine Eingabe oder weist sie zurück. Abhängig von dem Symbol, das vom Eingabekopf gelesen wird, und von aus dem oder den Speichern gelesenen Daten (bzw. einem Signal, dass ein Speicher leer ist) legt die endliche Kontrolle eine oder mehrere der folgende Operationen fest:

- Vorrücken des Eingabekopfes auf die nächste Position,
- Je Speicher höchstens einen der oben beschriebenen Zugriffe.

Nach Ausführen dieser Operationen geht die Maschine in einen neuen Zustand über.

Wir unterscheiden zwischen deterministischen und nichtdeterministischen Simulationen, wobei sowohl Simulator wie simulierte Maschine im gleichen Modus arbeiten.

Eine obere Schranke der Form $O(f(n))$ bedeutet, dass n Schritte des simulierten Akzeptors in $O(f(n))$ Schritten simuliert werden können (im Unterschied zu späteren Kapiteln bezeichnet hier n also die Schrittzahl und nicht die Eingabelänge, was die Angabe der Schranken vereinfacht). Diese Simulation ist nicht notwendigerweise eine schrittweise Simulation. Eine untere Schranke der Form $\Omega(f(n))$ bedeutet, dass jede *allgemeine* Simulation von n Schritten $\Omega(f(n))$ Zeit benötigt. Um eine solche Schranke zu beweisen ist es daher ausreichend, ein spezielles Problem zu definieren, das einen solchen Aufwand erfordert. Wenn beispielsweise ein Problem existiert, das durch Maschinenmodell A in linearer Zeit gelöst werden kann, während Modell B hierfür quadratische Zeit benötigt, dann zeigt dies die Schranke $\Omega(n^2)$ für die Simulation von A durch B. Für einige Simulationen, die in den folgende Abschnitten erwähnt werden, sind keine Literaturquellen angegeben. Es handelt sich dann um sehr einfache Techniken, die allgemein bekannt sind.

Simulation wird hier generell in der allgemeinsten Form verstanden, d.h., Maschine A simuliert B, wenn beide die gleiche Menge von Eingaben akzeptieren. Häufig werden andere Begriffe der Simulation verwendet, wie Simulationen, in denen ein Speicher simuliert werden muss und es Aufgabe des Simulators ist, auf entsprechende Befehle hin Daten abzulegen oder zur Verfügung zu stellen. Ein noch weiter eingeschränkter Begriff der Simulation könnte vorschreiben, dass die Speicherung mehrerer Kopien eines Datums

nicht zulässig ist. Da dies die interne Arbeitsweise des Simulators betrifft, werden wir solche Beschränkungen nicht betrachten.

Eine gegenüber der allgemeinsten Arbeitsweise strengere Form der Simulation wird mit dem Begriff „Online“ bezeichnen. Auch einige unserer unteren Schranken gelten für Online-Simulationen, wobei wir dies genauer präzisieren werden. Es ist wichtig, in diesem Zusammenhang zu bemerken, dass die für Online-Simulationen erzielten unteren Schranken möglicherweise größer sind als obere Schranken für Simulationen ohne die Online-Bedingung.

Der Kommentar „online“ neben einer unteren Schranke bedeutet, dass diese Schranke für einen Arbeitmodus gilt, bei dem der Simulator eine *Funktion* berechnen muss (im Gegensatz zur Spracherkennung, für die eine Simulation möglicherweise leichter ist). Der Simulator hat die Ausgabe in einer solchen Weise zu erzeugen, dass das i-te Zeichen ausgegeben wird, bevor das $(i+1)$-te Zeichen gelesen wird. Ohne diesen Kommentar gelten die unteren Schranken für Spracherkennungsprobleme.

In einigen Beweisen für untere Schranken benötigen wir den Begriff der *Inkompressibilität*. Eine Zeichenkette x über dem Alphabet Σ heißt inkompressibel, wenn jede Eingabe über Σ, für die eine fest gewählte universelle Turingmaschine als Ausgabe x erzeugt, mindestens die Länge $|x|$ hat. Die Rolle der Turingmaschine kann von anderen universellen Berechenbarkeitsbegriffen übernommen werden. Für genauere Definitionen und eine große Zahl an Resultaten sei auf [44] verwiesen. Ein Abzählargument zeigt, dass es inkompressible Zeichenketten jeder Länge gibt.

Eine *selbst-begrenzende* Kodierung der Zeichenkette x gestattet es, die Kodierung von x von der Konkatenation mit jeder anderen Zeichenkette abzutrennen. Eine Möglichkeit, solch eine Kodierung zu erhalten, ist es, die Kette $1^{|x|}0x$ zu bilden. Diese Zeichenkette hat die Länge $2|x|+1$.

2.1 Simulationen durch Queues

In dieser und den folgenden Tabellen werden jeweils Simulationen durch eine Datenstruktur (bzw. eine Kombination verschiedener Datenstrukturen) zusammengefasst. In der ersten Spalte fndet man die simulierten Datenstrukturen.

Deterministische Simulationen durch eine Queue:

	untere Schranke	obere Schranke
1 umkehrbeschr. Keller	$\Omega(n^{4/3}/\log n)$ [43]	$O(n^{3/2})$ Satz 3
1 oder 2 Keller, 1 Band	$\Omega(n^{4/3}/\log n)$ [43]	$O(n^2)$
2 Queues	$\Omega(n^2)$ [43]	$O(n^2)$

Deterministische Simulationen durch k Queues:

	untere Schranke	obere Schranke
$k+1$ Queues	$\Omega(n^{1+1/k}/\log^{1/k} n)$ (online) [28]	$O(n^{1+1/k})$ [28]
3 Keller, 2 Bänder	$\Omega(n^{1+1/k}/\log^{1/k} n)$ (online) Korollar 1	$O(n^{1+1/k})$ [28]

Nichtdeterministische Simulationen durch eine Queue:

	untere Schranke	obere Schranke
1 Keller	$\Omega(n^{4/3}/\log n)$ [43]	$O(n^{3/2})$ Satz 1
2 Queues	$\Omega(n^2/\log^2 n \log\log n)$ [43]	$O(n^2)$

Nichtdeterministische Simulationen durch zwei Queues:

	obere Schranke
Queues	$O(n)$ [43]
1 Keller	$O(n \log n)$ [65]
mehrdimensionale Bänder	$O(n \log^2 n)$ [58] $O(n \log n)$ Satz 5

2.2 Simulationen durch Keller

Deterministische Simulationen durch zwei Keller:

	untere Schranke	obere Schranke
1 Queue		$O(n)$ [26]
Bänder, Queues, Keller	$\Omega(n^{4/3}/\log^{1/(3-\varepsilon)} n)$ $\forall \varepsilon \geq 0$ [42]	$O(n^2)$
d-dimensionale Bänder	$\Omega(n^{2-1/d})$ (online) [24]	$O(n^2)$

Die bekannten deterministischen Simulationen durch drei Keller führen zu denselben Schranken wie diejenigen durch ein lineares Band und einen Keller.

Nichtdeterministische Simulationen durch zwei Keller:

	untere Schranke	obere Schranke
1 Queue		$O(n)$ [26]
1 Deque		$O(n)$ Satz 7
Bänder, Queues, Keller	$\Omega(n^{4/3}/\log^{1/(3-\varepsilon)} n)$ $\forall \varepsilon \geq 0$ [42]	$O(n^2)$

Nichtdeterministische Simulationen durch drei Keller:

	obere Schranke
Bänder, Queues, Keller	$O(n)$ [5] (implizit)
mehrdimensionale Bänder	$O(n \log^2 n)$ [49] (implizit) $O(n \log n)$ Satz 5

2.3 Simulationen durch lineare Bänder

Deterministische Simulationen durch ein lineares Band:

	untere Schranke	obere Schranke
1 Queue	$\Omega(n^2)$ [45]	$O(n^2)$
2 Keller	$\Omega(n^2)$ [45]	$O(n^2)$

Nichtdeterministische Simulationen durch ein lineares Band:

	untere Schranke	obere Schranke
1 Queue	$\Omega(n^{4/3}/\log^{2/3} n)$ [45]	$O(n^{3/2}\log^{1/2} n)$ [42]
2 Keller	$\Omega(n^{3/2}/\log^{1/2} n)$ [45]	$O(n^{3/2}\log^{1/2} n)$ [42]
2 Bänder, 3 Keller	$\Omega(n^2/\log^{(k)} n)$ $\forall k \geq 1$ [16]	$O(n^2)$

Hier bedeutet $\log^{(k)}$ die k-fache Iteration von log.

Deterministische Simulationen durch zwei lineare Bänder (von denen eines ein Keller sein kann):

	untere Schranke	obere Schranke
Bänder, Queues, Keller	$\Omega(n\log^{1/3} n)$ (online) [54]	$O(n\log n)$ [25]
d-dimensionale Bänder	$\Omega(n^{2-1/d})$ (online) [24]	$O(n^{2-1/d}\log n)$ [70]

Nichtdeterministische Simulationen durch zwei lineare Bänder (von denen eines ein Keller sein kann):

	obere Schranke
Bänder, Queues, Keller	$O(n)$ [5] (implizit)
mehrdimensionale Bänder	$O(n\log^2 n)$ [49] $O(n\log n)$ Satz 5

2.4 Simulationen mittels mehrdimensionaler Bänder

Deterministische Simulationen durch ein d-dimensionales Band ($d \geq 2$):

	untere Schranke	obere Schranke
Bänder, Queues, Keller	$\Omega(n^{1+1/d}/\log n)$ [10]	$O(n^{1+1/d}/\log n)$ [10]

Deterministische Simulation durch zwei d-dimensionale Bänder ($d \geq 2$):

	obere Schranke
Bänder, Queues, Keller	$O(n\log\log n)$ [29]

2.5 Simulationen durch hybride Maschinen

Deterministische Simulationen durch ein lineares Band und eine Queue:

	untere Schranke	obere Schranke
Bänder, Queues, Keller	$\Omega(n \log^{1/4} n)$ (online) [28]	$O(n \log n)$ [25] (implizit)
d-dimensionale Bänder	$\Omega(n^{2-1/d})$ (online) [24] (implizit)	$O(n^{2-1/d} \log n)$ [70] (implizit)

Deterministische Simulationen durch einen Keller und eine Queue:

	untere Schranke	obere Schranke
Bänder, Queues, Keller	$\Omega(n \log^{1/4} n)$ (online) [28]	$O(n \log n)$ Satz 6

Die nichtdeterministischen Simulationen durch ein lineares Band (oder einen Keller) und eine Queue führen zu den gleichen Schranken wie diejenigen mit zwei Bändern (Anpassung der Simulation aus [5] an eine Queue wie in Theorem 4.2 von [43]).

Kapitel 3

Simulationen durch Queuemaschinen

3.1 Einleitung

Ein klassisches Resultat besagt, dass ein Speicher, der in der Art einer Queue organisiert ist, als Basis universeller Berechnungen dienen kann. Implizit wurde dies von Post in [61] gezeigt, der die Universalität von normalen Produktionen bewies. Eine normale Produktion $u \to v$ transformiert Zeichenketten der Form ux in xv. Um eine tatsächliche Äquivalenz zu generellen formalen Systemen zu erhalten, ist lediglich ein Erzeugungsmodus erforderlich, der Nichtterminalsymbole benutzt.

Diese auf Post zurückgehende Idee griff Manna in [47] auf, wo die Post-Maschine in der Form eines Automaten mit endlicher Kontrolle und der Speicherstruktur Queue als Berechenbarkeitsmodell eingeführt wurde. Einige Arbeiten wurden ähnlichen Modellen im Rahmen der Theorie formaler Sprachen gewidmet. So untersuchte Vollmar in [74] Automaten mit einem Queue-Speicher. Er konzentrierte sich dabei auf solche Automaten, die in Realzeit arbeiten (d.h., die Anzahl der ausgeführten Schritten entspricht der Eingabelänge) und mit leerem Speicher akzeptieren. Er konnte die deterministische und nichtdeterministische Variante dieses Modells trennen, setzte die von ihnen definierten Sprachklassen in Beziehung zur Chomsky-Hierarchie, untersuchte Abschlusseigenschaften und Entscheidbarkeitsfragen. Als weiteres Beispiel einer Arbeit aus dem Bereich formaler Sprachen erwähnen wir [6].

Während also die Universalität von Queue-Speichern lange bekannt war, wurden erst vergleichsweise spät Untersuchungen über die Komplexität der Berechnungen von Modellen mit solchem Speicher durchgeführt. Die Arbeit

[43] von Li, Longré und Vitanyi enthält im Wesentlichen untere Schranken für die gegenseitige Simulation von Maschinen mit einer unterschiedlichen Anzahl von Queues, Kellern und Bändern. Mit Techniken der Beschreibungskomplexität wurde bewiesen, dass die Simulation eines Kellers durch eine Queue $\Omega(n^{4/3}/\log n)$ Schritte erfordert, was sowohl für deterministische wie nichtdeterministische Modelle gilt (aus Gründen der Übersichtlichkeit bezeichnen wir hier wie in Kapitel 2 mit n die Anzahl der zu simulierenden Schritte). Eine optimale quadratische untere Schranke gilt für die deterministische Simulation einer Queue durch ein Band [45], während eine nichtdeterministische Simulation in Zeit $O(n^{3/2}\log^{1/2} n)$ möglich ist [42]. Die untere Schranke für diese Simulation ist $\Omega(n^{4/3}/\log^{2/3} n)$ [45].

Für lineare Bänder ist bekannt, dass sich viele Speicher effizient auf zwei Bändern simulieren lassen. Im deterministischen Fall wurde die Schranke $O(n\log n)$ von Hennie und Stearns gezeigt [25], während für nichtdeterministische Maschinen sogar eine Lösung in Realzeit bekannt ist [5]. Das letztgenannte Resultat lässt sich auf Queuemaschinen übertragen, [6, Theorem 4.5] und [43, Theorem 4.2]. Dagegen ließen Li, Longré und Vitanyi die Frage offen, ob eine bessere Simulation als die offensichtliche quadratische für deterministische Queuemaschinen möglich ist. Diese Frage wurde positiv von Hühne in [28] beantwortet, der zeigte, dass Maschinen mit mehreren linearen Speichern durch Maschinen mit k Queues in Zeit $O(n^{1+1/k})$ simuliert werden können. Unter der Annahme einer Schritt-für-Schritt Simulation ist diese Lösung fast optimal, denn er konnte eine untere Schranke der Form $\Omega(n^{1+1/k}/\text{polylog } n)$ beweisen. Die Arbeit von Rosenberg [65] gibt ein Verfahren der Zeitkomplexität $O(n\log n)$ für die nichtdeterministische Erkennung von kontextfreien Sprachen durch Maschinen mit zwei Queues an. Da hierbei wesentlicher Gebrauch von der Charakterisierung kontextfreier Sprachen durch Grammatiken gemacht wird, lässt sich Rosenbergs Lösung nicht auf die Simulation anderer Speicherstrukturen übertragen.

Von Hartmanis und Stearns [22] stammt eine quadratische Simulation von k-dimensionalen Bändern durch lineare Bänder. Pippenger und Fischer [60] verbesserten die Zeitschranke zu $O(n^{2-1/k})$ und das Resultat von Hennie [24] (mit der Korrektur durch [20]) zeigt, dass bei deterministischer Schritt-für-Schritt Simulation keine weitere Beschleunigung möglich ist. Grigor'ev [21] und Loui [46] gaben effizientere Simulation unter Verwendung von m-dimensionalen Bändern für $m > 1$ an. Das auf Grigor'ev zurückgehende Verfahren benutzt Nichtdeterminismus, um eine kompakte Darstellung des Speichers zu erraten, während Loui deterministische Maschinen verwendet. Monien [49] verbesserte die nichtdeterministische Simulation auf $O(n\log^2 n)$. Die Simulation erhält die Platzschranke der simulierten Maschine und ist stark in einem weiter unten definierten Sinne.

Die diskutierten Ergebnisse wurden von Mike Robson und dem Autor dieser Arbeit in den folgenden Aspekten ergänzt und verbessert:

- Jede kontextfreie Sprache kann mit Hilfe einer Queue in Zeit $O(n^{3/2})$ nichtdeterministisch erkannt werden.

- Jede deterministische Maschine mit einem Keller, der während der Berechnung höchstens einmal vom schreibenden in lesenden Modus wechselt (one-turn pushdown) kann deterministisch in Zeit $O(n^{3/2})$ simuliert werden. Diese umkehrbeschränkten Maschinen akzeptieren genau die deterministisch linear kontextfreien Sprachen.

- Jede Einband-Turingmaschine (ohne separates Eingabeband, die Eingabe steht zu Beginn der Rechnung auf dem Arbeitband) mit Zeitschranke $t(n)$ kann nichtdeterministisch in $O(t(n))$ Schritten simuliert werden.

- Jede Turingmaschine mit mehreren mehrdimensionalen Bändern, die in Zeit $t(n)$ mit Platzschranke $s(n)$ akzeptiert, kann nichtdeterministisch durch zwei Queues (oder lineare Bänder) in Zeit $O(t(n) \log s(n))$ und Platz $s(n)$ simuliert werden. Da zur Erkennung kontextfreier Sprachen linearer Platz ausreicht, verallgemeinert dieses Resultat das oben erwähnte Ergebnis von Rosenberg.

- Jede deterministische Turingmaschine mit mehreren linearen Bändern, die in $t(n)$ Schritten akzeptiert, kann durch eine Maschine mit einer Queue und einem Keller in $O(t(n) \log t(n))$ Schritten simuliert werden.

Außer dem Resultat für Einband-Turingmaschinen können diese Simulationen stark ausgeführt werden, in dem Sinn dass alle *möglichen* Berechnungen des Simulators die Schranke einhalten müssen, wenn dies die simulierte Maschine tat.

Unter diesen Resultaten sind die ersten sub-quadratischen Simulationen anderer Speicher, wie Keller und Bänder, durch eine Queue. Die Simulationen von Maschinen mit Kellerspeichern erreichen fast die entsprechenden unteren Schranken.

Die platzerhaltende Simulation und eine vollständige Darstellung der Simulation einer Maschine mit mehreren Speichern durch eine Queue und einen Keller sind Verbesserungen gegenüber der Arbeit [58], die ebenfalls in Zusammenarbeit mit Mike Robson erzielt wurden.

3.2 Vorbemerkungen

Falls nicht ausdrücklich anderes gesagt wird, sind alle simulierenden Maschinen nichtdeterministisch und haben Zugriff auf eine oder mehrere Queues.

Eine Maschine akzeptiert in $t(n)$ Schritten, wenn sie für jede akzeptierte Eingabe der Länge n eine Berechnung besitzt, die nach höchstens $t(n)$ Schritten mit Akzeptierung endet. In Gegensatz zu diesem (schwachen) Begriff der Akzeptierung ist eine Zeit- oder Platzschranke stark, falls alle *möglichen* Berechnungen auf einer Eingabe der Länge n die Schranke einhalten.

3.3 Simulation von Maschinen mit einem einzigen Speicher

Wir geben zunächst einen arithmetischen Hilfssatz an, der in einigen weiteren Beweisen dieses Abschnitts angewendet wird.

Lemma 1 *Unter Verwendung eines speziellen Markierungssymbols kann eine Queuemaschine Q ihren Queue-Inhalt v in Teilketten v_i mit $|v_i| = \Theta(\sqrt{|v|})$ zerlegen, mit Ausnahme der letzten Kette, die kürzer sein darf. Die Anzahl der Markierungssymbole, die auf diese Weise erzeugt werden, bezeichnen wir mit root$(|v|) = \Theta(\sqrt{|v|})$. Die Zeitkomplexität dieses Vorgangs ist $O(|v| \log |v|)$.*

Beweis. Queuemaschine Q markiert das Ende der Queue mit einem Symbol \$ und schreibt in einem Durchlauf durch den Inhalt der Queue (wobei die ursprünglichen Zeichen nach dem Lesen wieder gespeichert werden) die Zeichenkette $*\#$ hinter jedes Symbol des ursprünglichen Queue-Alphabets ($\$, *, \#$ sind neue Symbole).

Nun beginnt Q in einer Folge von Durchläufen die Markierungssymbole in der Queue zu verändern. Solange die Anzahl der $*$ in der Queue größer als eins ist, entfernt Q jedes zweite $*$ mit dem ersten beginnend und entfernt in jedem zweiten Durchgang jedes zweite $\#$, außer dasjenige direkt vor \$. Die Maschine Q macht in dieser Weise $\lfloor \log |v| \rfloor$ Durchläufe in $O(|v| \log |v|)$ Schritten und teilt die Anzahl der $\#$-Symbole $\lfloor \lfloor \log |v| \rfloor / 2 \rfloor$ Male annähernd durch 2. Nach diesen Operationen ist v in k Blöcke v_i aufgeteilt, die durch $\#$ begrenzt werden. Wir haben

$$|v_i| \leq 2^{\lfloor \frac{\lfloor \log |v| \rfloor}{2} \rfloor} \leq \sqrt{|v|}$$

und

$$k \leq \frac{|v|}{2^{\lfloor \frac{\lfloor \log |v| \rfloor}{2} \rfloor}} \leq 2\sqrt{|v|}.$$

□

3.3.1 Erkennung kontextfreier Sprachen

Wir zeigen nun, dass kontextfreie Sprachen in $O(n\sqrt{n})$ Schritten von nichtdeterministischen Queuemaschinen erkannt werden können.

Der erste Satz, der bewiesen wird, ist eine schwächere Version des zweiten Resultats. Um den Ansatz besser erklären zu können scheint es jedoch vorteilhaft, zunächst die einfachere Form zu beschreiben und dann die Verschärfung anzugeben.

Satz 1 *Jeder nichtdeterministische Kellerautomat kann von einer Maschine mit einer Queue in $O(n\sqrt{n})$ Schritten simuliert werden. Daher kann jede kontextfreie Sprache von einer solchen Maschine in Zeit $O(n\sqrt{n})$ akzeptiert werden.*

Beweis. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass der zu simulierende Kellerautomat P in linearer Zeit arbeitet und mit leerem Keller akzeptiert. Wir nehmen an, dass Kellerpositionen durch natürliche Zahlen in aufsteigender Reihenfolge entsprechend Push-Operationen adressiert werden. Wir nennen eine Folge von Operationen des Kellerautomaten P eine *Sektion*, falls seine erste Operation eine Push-Operation ist, die die Kellerposition a beschreibt, die letzte Operation eine Pop-Operation ist, welche Position a liest und keine Operation enthält, die auf eine Position mit einer kleineren Adresse zugreift.

Die Simulation beruht auf der Existenz einer Folge $s_1, \ldots, s_m$ mit folgenden Eigenschaften:

- Jede Operation der Berechnung ist in s_1.
- Jedes Paar von Sektionen ist entweder disjunkt oder eine ist vollständig in der anderen enthalten.
- $m = O(\sqrt{n})$.
- Die Anzahl der Kelleradressen, die von Operationen in s_j, aber von keinem in s_j eingebetteten s_k benutzt werden, ist $\leq \sqrt{n}$.

Dass eine solche Folge von Sektionen existiert, ist leicht einzusehen, indem die $\lfloor\sqrt{n}\rfloor$ möglichen Aufteilungen des Kellers in Blöcke betrachtet werden, in denen der erste Block eine positive Länge $i \leq \lfloor\sqrt{n}\rfloor$ besitzt, während alle übrigen die exakte Länge $\lfloor\sqrt{n}\rfloor$ haben. Jede dieser Aufteilungen induziert

eine Folge von Sektionen, die mit einer Operation beginnen, in der P ein Symbol auf die erste Adresse des Blocks schreibt, und mit der nächsten Operation endet, in der das gleiche Feld freigegeben wird. Eine Ausnahme ist der erste Block, für den die gesamte Berechnung die zugehörige Sektion bildet. Da jede der $O(n)$ Push-Operationen den Beginn einer Sektion für höchstens eine der $\lfloor\sqrt{n}\rfloor$ Aufteilungen definiert, existiert eine Aufteilung, die $O(\sqrt{n})$ Sektionen liefert.

Die Queuemaschine Q simuliert P in zwei Phasen. In der ersten Phase rät Q eine Berechnung von P und imitiert sowohl das Lesen der Eingabe wie auch die Zustandsübergänge von P. Den Zugriff von P auf seinen Keller simuliert Q dadurch, dass Symbole auf die Queue geschrieben werden. Hier nutzt Q wiederum die nichtdeterministische Arbeitsweise, um den Eintritt und das Verlassen einer Sektion zu erraten. Alle Daten einer Sektion, die zu keiner eingebetteten Sektion gehören, werden in einem zusammenhängenden Teil der Queue gespeichert. Falls es Q gelingt, eine akzeptierende Rechnung von P zu erraten, prüft Q die Konsistenz der in der Queue gespeicherten Information. Dies geschieht in $O(\sqrt{n})$ zyklischen Durchläufen durch die Queue. Schließlich akzeptiert Q, wenn diese Prüfung des Inhalts der Queue erfolgreich abgeschlossen werden kann.

Wenn das Alphabet des Kellerautomaten mit X bezeichnet wird, dann sei das Queue-Alphabet

$$\{(push, x)|x \in X\} \cup \{(read, x)|x \in X\} \cup \{*, \$, pop, finished\}.$$

Phase 1 Die Blöcke der Queue, welche den Sektionen entsprechen, werden durch Symbole \$ getrennt und bestehen aus Zeichenketten über $(push, x)$, $(read, y)$ und pop, sowie möglicherweise dem Symbol *finished* am Anfang des Abschnitts. Abgesehen von zyklischen Verschiebungen, ist die Anordnung der Abschnitte die gleiche wie die der zugehörigen Sektionen. In der normalen Simulation ist der Start des aktiven Abschnitts der Queue mit einem Symbol $*$ markiert. Wenn die erratene Berechnung von P auf den Keller zugreift, wird das passende Symbol ($(push, x)$, $(read, y)$ oder pop) an den aktuellen Inhalt der Queue angefügt.

Wenn Q rät, dass die Kelleroperation die erste einer Sektion ist, schreibt Q ein \$ gefolgt von einem $*$ um einen neuen aktiven Abschnitt einzurichten. Dann durchläuft Q die Queue einmal, kopiert dabei alle gelesenen Symbole zum Ende der Queue, außer das erste $*$ (dieses wird somit entfernt). Wenn das zweite $*$ gefunden wird, nimmt Q die normale Simulation wieder auf.

Wenn Q rät, dass die gerade gespeicherte Kelleroperation die letzte einer Sektion war, wobei diese Sektion nicht s_1 ist, dann muss Q den letzten Abschnitt finden, der zu einer noch nicht abgeschlossenen Sektion gehört. Die

Queuemaschine durchläuft die Queue, wobei der Inhalt zyklisch kopiert wird, rät den Anfang eines vorherigen Abschnittes, der nicht mit *finished* beginnt, und markiert ihn mit $*$. Dann prüft Q, ob der nächste Abschnitt mit einem $*$ wirklich der nächste ist, der nicht mit *finished* beginnt, ersetzt dies alte $*$ durch *finished*, und setzt dann das Kopieren fort, bis sie das verbleibende, neue $*$ wiederfindet und kopiert weiter bis zum Ende des direkt folgenden Abschnitts. Falls die Sektion (gemäß dem Raten) s_1 ist, verifiziert Q, dass das Ende der Eingabe erreicht ist, schreibt *finished* in die Queue und startet Phase 2. In diesem Abschnitt folgt also das Symbol *finished* den anderen, aber dies wird keine Probleme in Phase 2 verursachen.

Hierdurch wird sichergestellt, dass die Symbole, welche die Kelleroperationen einer Sektion kodieren, in der Reihenfolge der Ausführung der Operation in die Queue geschrieben werden.

Durch Raten einer Einteilung der Berechnung in $O(\sqrt{n})$ Sektionen kann Q die Phase 1 in $O(n\sqrt{n})$ Schritten ausführen.

Phase 2 In Phase 2 muss Q verifizieren, dass die erratene Berechnung von P wirklich akzeptierend ist und die gespeicherten Kelleroperationen einer möglichen Folge von Sektionen entsprechen. Um dies zu tun, wird überprüft, dass jeder Block der Queue (eine Folge von Symbolen zwischen zwei \$) folgende Eigenschaften hat:

- Er enthält *finished*.
- Er hat die gleiche Zahl *push* und *pop* Symbole.
- Er besitzt keinen Präfix mit mehr *pop* als *push* Symbolen.
- Er enthält vor jedem $(read, x)$ ein $(push, x)$, wobei die dazwischen liegenden Symbole die gleiche Anzahl von *push* und *pop* Symbolen beinhalten und kein Präfix eine Mehrheit von *pop* Symbolen im Verhältnis zu *push* Symbolen.

Dies wird in $O(\sqrt{n})$ Durchläufen durch die Queue erreicht, wobei jeder Durchlauf mindestens jede Folge von Symbolen entfernt, die mit $(push, x)$ beginnt, mit *pop* endet und zwischen diesen Symbolen nur $(read, x)$ enthält. Da die Anzahl von Kelleradressen, die in jeder der Sektionen verwendet wird, die Ordnung $O(\sqrt{n})$ hat, reduzieren $O(\sqrt{n})$ Durchläufe jeden Block zu einem einzigen Symbol *finished*, es sei denn, eine Inkonsistenz wird entdeckt.

Im Einzelnen führt Q die folgenden Operationen aus. Symbole werden vom Kopf der Queue zum Ende kopiert, mit Ausnahme der folgende Transformationen:

- Nach $(push, x)$ wird jedes folgende $(read, x)$ ignoriert.
- Wenn ein *pop* einem *push* folgt, werden beide entfernt.

Wenn ein Durchlauf durch die Queue (von einem Lesen eines einzelnen $*$ zum nächsten) nur Symbole \$ und *finished* findet, dann akzeptiert Q. Es ist klar, dass die Zeit, die für Phase 2 benötigt wird, $O(n\sqrt{n})$ ist und dass Q nur dann akzeptiert, wenn die nichtdeterministischen Schritte in Phase 1 korrekt erraten wurden und zu einer gültigen und akzeptierenden Rechnung von P gehören. □

Satz 2 *Jeder nichtdeterministische Kellerautomat kann durch eine nichtdeterministische Queuemaschine in $O(n\sqrt{n})$ Schritten im starken Sinne simuliert werden. Daher kann jede kontextfreie Sprache durch eine nichtdeterministische Queuemaschine in $O(n\sqrt{n})$ im starken Sinne akzeptiert werden.*

Beweis. Wir skizzieren die Unterschiede zum Beweis von Theorem 1.

Wir zerlegen die Simulation in *Segmente*, die zu Stadien der simulierten Berechnung gehören. Am Anfang des Segmentes i wird der aktuelle Inhalt des Kellers durch eine Folge von *push* Symbolen repräsentiert, die in der korrekten Reihenfolge in der Queue gespeichert sind. Wir schreiben $s(i)$ für die Größe des Kellers zu diesem Zeitpunkt.

Segment i verläuft wie folgt:

- Alle *push* Symbole werden markiert und ein Leersymbol wird an eine nichtdeterministisch gewählte Position geschrieben.
- In wiederholten Durchläufen durch die Queue wird jeweils bei jedem zweiten markierten *push* Symbol die Markierung entfernt, wobei mit dem ersten begonnen wird. Außerdem wird nach jedem Leersymbol in einer nichtdeterministischen Position vor dem folgenden Leersymbol ein weiteres eingefügt. Dies wird so lange wiederholt, bis alle Markierungen entfernt worden sind.
  Insgesamt hat dies den Effekt, eine Anzahl von Leersymbolen einzufügen, die der ersten Potenz von 2 entspricht, die größer als $s(i)$ ist. Wir nennen diese Zahl $s^*(i)$.
- Mit Hilfe der Methode aus Lemma 1 werden root$(s(i))$ Markierungen eingefügt.
- Nichtdeterministisch werden Markierungen in die gespeicherten Kelleroperationen eingefügt, so dass mindestens eine Kelleroperation zwischen zwei Markierungen gespeichert ist.

- Beginnend mit dem Lesekopf am Ende des kodierten Kellerinhalts, wird der Prozess des nichtdeterministischen Ratens der Berechnung und Speicherns der Kelleroperationen wie in Phase 1 der Simulation von Theorem 1 ausgeführt, mit den folgenden Veränderungen:

  1. Neue Symbole werden nur an Positionen geschrieben, die vorher Leersymbole enthielten.
  2. Jedes Mal, wenn die Queue vollständig durchlaufen worden ist, wird eine Markierung entfernt, oder – falls keine Markierung mehr vorhanden ist – der nichtdeterministische Prozess beendet.

  Wir schreiben $s'(i)$ für die Anzahl der Kelleroperationen, die während Segment i nichtdeterministisch erzeugt wurden.

- Wenn die erratene Berechnung noch nicht terminiert ist, wird verifiziert, dass alle Leersymbole überschrieben worden sind. Falls nicht, wird die Simulation abgebrochen.

- Mit Hilfe der Methode von Lemma 1 werden $\text{root}(s(i) + s'(i))$ Markierungen erzeugt.

- Der Prozess des Überprüfens und Löschens von Symbolen, die Kelleroperationen kodieren, wird wie in Phase 2 der Simulation von Theorem 1 ausgeführt, aber bei jedem vollständigen Durchlauf durch die Queue wird eine Markierung entfernt. Die Simulation wird abgebrochen, falls keine Markierung mehr vorhanden ist und noch mindestens ein *pop* Symbol übrigbleibt.

- Alle Segmentmarkierungen und *finished* Symbole werden entfernt.

- Wenn die nichtdeterministisch erzeugte Berechnung terminiert, stoppt auch Q und akzeptiert genau dann, wenn es keine verbleibenden Operationssymbole in der Queue mehr gibt.

Das Argument der möglichen Wahl günstiger Grenzen im Beweis von Theorem 1 zeigt, dass – durch passende nichtdeterminstische Schritte – mindestens $s^*(i)$ Kelleroperationen ausgeführt werden können, ohne dass dafür mehr als $\sqrt{s(i)}$ Durchläufe durch die Queue erforderlich sind, um sie an den korrekten Positionen zu speichern. Daher sind höchstens $\sqrt{s(i) + s'(i)}$ Durchläufe nötig, um alle *read* und *pop* Operationen zu löschen. Durch das Einfügen von Leersymbolen während der ersten beiden Schritte, kann die Simulation die Operationen durch Überschreiben der Leersymbole korrekt

ausgeführt werden. Daher existiert eine Berechnung, die erfolgreich die Simulation für Segment i ausführt und einen Queue-Inhalt zurücklässt, der die Fortsetzung mit Segment $i+1$ ermöglicht.

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass P nur eine konstante Anzahl Schritte ausführen kann, ohne die Eingabe zu lesen oder eine Kelleroperation auszuführen. Daher kann die simulierende Maschine Q ebenfalls nur eine konstante Anzahl Schritte ausführen, ohne die Eingabe zu lesen oder eine Queueoperation auszuführen. Somit ist die gesamte Anzahl von Schritten, die Q auf einer Eingabe der Länge n ausführt $O(n+q)$, wobei q die Anzahl der geschriebenen Queuesymbole ist.

Wir schreiben $q(i)$ für die Anzahl der Queuesymbole, die im Segment i geschrieben wurden, und $t(i)$ für die Zeit, die P in dem entsprechenden Abschnitt i der Berechnung benötigt. Die Größe der Queue ist immer $O(s(i)+s'(i))$ und die Anzahl der Durchläufe durch die Queue ist $O(\sqrt{s(i)+s'(i)})$ um mindestens $s'(i)$ Schritte zu simulieren. Außer möglicherweise für das letzte Segment gilt $t(i) \geq s'(i) \geq s(i)$, was ein Verhältnis von $q(i)$ zu $t(i)$ von $O(\sqrt{s(i)+s'(i)})$ ergibt. Dies ist $O(\sqrt{n})$, was für q über alle Segmente außer dem letzten $O(n\sqrt{n})$ ergibt. Für den Fall, dass Segment i das letzte Segment einer erfolgreichen oder scheiternden Simulation ist, nutzten wir aus, dass $s'(i) = O(s(i))$, so dass $q(i) = O(s(i)\sqrt{s(i)})$. Daher hat dieses Segment ebenfalls $q(i) = O(n\sqrt{n})$ und somit gilt $q = O(n\sqrt{n})$.

Dies schließt den Beweis ab, dass jede Berechnung, egal ob akzeptierend oder ablehnend, in $O(n\sqrt{n})$ Schritten simuliert werden kann. □

Die nächste Maschine, die wir simulieren wollen, ist mit einem umkehrbeschränkten Keller ausgestattet. In jeder Berechnung einer solchen Maschine darf es nur einen Wechsel von *push* Operationen zu *pop* Operationen geben (wobei natürlich auch *read* Operationen möglich sind). Deterministische Maschinen dieser Klasse akzeptieren genau die deterministisch linear kontextfreien Sprachen. Hier sei nur daran erinnert, dass linear kontextfreie Sprachen durch lineare Grammatiken charakterisiert werden und eine echte Teilmenge der kontextfreie Sprachen bilden.

Satz 3 *Jede mit einem umkehrbeschränkten Keller ausgestattete deterministische Maschine kann durch eine deterministische Maschine mit einer Queue in $O(n\sqrt{n})$ Schritten simuliert werden.*

Beweis. Sei P eine Maschine wie im Satz beschrieben. Zunächst bemerken wir, dass P, falls sie akzeptiert, dies in linearer Zeit tut.

Sei Q die Queuemaschine, die P simuliert. Diese Maschine habe das Alphabet $X \cup \{\#, *, \$\}$, wobei X das Kelleralphabet von P ist und die Vereinigung disjunkt ist. Die Länge der Eingabe sei n.

Die Idee der Simulation ist es, die Berechnung von Q in drei Phasen zu zerlegen. In der ersten Phase simuliert Q Push-Operationen von P, indem sie die von P im Keller gespeicherten Symbole in die Queue schreibt. Wenn P beginnt, den Kellerinhalt zu lesen, unterbricht Q die direkte Simulation der Schritte von P und bereitet ihren Speicher auf die schnelle Simulation der folgenden Schritte vor. In der letzten Phase nimmt Q wieder die direkte Simulation von Schritten auf, in denen P den Inhalt seines Kellers liest. Wir geben im Folgenden eine genauere Darstellung der zweiten und dritten Phase.

Wir bezeichnen den Inhalt der Queue in dem Augenblick, wenn P den Zugriff auf den Keller umkehrt, mit v, $|v| = O(n)$. Mit der in Lemma 1 angegebenen Methode teilt Q die Zeichenkette v in Stücke v_i der Größe $O(\sqrt{|v|})$ auf. Jedes Teilwort wird durch ein neues Symbol # abgeschlossen. Danach fügt Q in einem Durchlauf durch die Queue vor jedem # ein Symbol $*$ ein. Dann beginnt Q, die Blöcke v_i zu spiegeln. Hierzu entfernt Q jeweils ein Symbol $x \in X$ am Anfang eines noch nicht vollständig gespiegelten Blockes, speichert es in der endlichen Kontrolle und fügt x nach dem $*$ des gleichen Blockes ein. Dieser Vorgang wird wiederholt, bis alle Blöcke mit einem $*$ beginnen. Danach werden alle Symbole $*$ in einem Durchlauf gelöscht. Nun ist jeder Block $v_i\#$ in $v_i^R\#$ umgeformt worden. Die Anzahl der hierfür erforderlichen Schritte ist $O(|v|\sqrt{|v|})$.

Die dritte Phase der Simulation, welche Q ausführt, erfordert eine weitere Vorbereitung, die den Zugriff auf den letzten Block der Queue beschleunigt. In k Zyklen fügt Q ein $*$ am Anfang jedes Blockes ein, der bisher bereits $*$ Symbole erhalten hat, und in den ersten Block, der bis zu diesem Zeitpunkt noch kein Symbol $*$ enthält. Dies wird so lange wiederholt, bis jeder Block mindestens ein $*$ beinhaltet. Als Folge dieser Operationen enthalten die Blöcke k, $k-1$, ..., 1 Symbole $*$. Die Vorbereitung ist $O(n\sqrt{n})$ zeitbeschränkt. Nun beginnt Q die dritte Phase der Simulation und ahmt die Operationen von P nach, die den Kellerinhalt lesen. Hierzu rotiert Q Blöcke an das Ende der Queue, bis sie den eindeutig bestimmten Block mit einem einzigen $*$ findet. Sie löscht das $*$ und liest diesen Block während sie das nächste Segment der Eingabe verarbeitet. Wenn Q das abschließende # liest, löscht sie dieses Symbol und schreibt \$ an das Ende der Queue. Dann löscht sie in einem Durchlauf durch die Queue ein $*$ in jedem Block. Sie wiederholt diesen Ablauf bis die Eingabe vollständig verarbeitet worden ist. Ein leerer Keller kann leicht erkannt werden, weil dann \$ das erste Symbol der Queue ist. Jede Rotation der Queue kann in $O(|v|)$ Schritten durchgeführt werden und es existieren $k = O(\sqrt{|v|})$ Blöcke. Daher werden für diese Phase nur $O(|v|\sqrt{|v|})$ Schritte benötigt. □

Die Simulation aus dem Beweis des letzten Satzes kann auf die Sprache $L = \{w\#w^R \mid w \in \{0,1\}^*\}$ angewendet werden, die in [43, Section 3.2] untersucht wurde. Unsere obere Schranke $O(n\sqrt{n})$ erreicht annähernd die untere Schranke $\Omega(n^{4/3}/\log n)$ aus [43].

3.3.2 Ein-Band Turingmaschinen

Satz 4 *Jede nichtdeterministische Turingmaschine mit einem zweiseitig unendlichen Band (das gleichzeitig Eingabe- und Arbeitsband ist), die in $t(n)$ Schritten akzeptiert, kann von einer nichtdeterministischen Maschine mit einer Queue in $O(t(n))$ Schritten simuliert werden.*

Beweis. Wir nennen die zu simulierende Turingmaschine S und die simulierende Queuemaschine Q. Wir benennen die Bandfelder von S mit aufeinander folgenden ganzen Zahlen, das Feld, das das erste Symbol der Eingabe enthält, trägt die Zahl 0. Wir nehmen weiter ohne Einschränkung der Allgemeinheit an, dass S in jedem Schritt ihren Kopf bewegt und dass es einen einzigen akzeptierenden Zustand gibt, der nur durch eine Rechtsbewegung des Kopfes erreichbar ist.

Eine *Kreuzungsfolge* der Maschine S ist die chronologische Folge der Zustände der endlichen Kontrolle, in die S übergeht, während ihr Kopf die Grenze zwischen zwei Nachbarfeldern überschreitet. Wir bezeichnen mit c_i die Kreuzungsfolge einer Berechnung, die zwischen Zelle $i-1$ und i auftritt. Zusätzlich halten wir die Richtung, in die sich der Kopf bewegt, fest und bezeichnen den Übergang in den Zustand q durch $\overrightarrow{q}$ für eine Rechtsbewegung und $\overleftarrow{q}$ für eine Linksbewegung. Als Konvention vereinbaren wir, dass c_0 mit dem Startzustand und einer (virtuellen) Rechtsbewegung beginnt.

Die Berechnung von S im Bezug auf Kreuzungsfolgen kann in drei Phasen zerlegt werden:

- Zellen links vom Eingabebereich ($i \leq 0$).
- Zellen innerhalb der Eingabe w ($0 < i \leq |w|$).
- Zellen rechts vom Eingabebereich ($i > |w|$).

Die Queuemaschine Q simuliert das Verhalten von S auf jedem Bandfeld, das von S benutzt wird, von links nach rechts, d.h., im Allgemeinen nicht in chronologischer Reihenfolge. Während einer Runde der Simulation, die zu einem Bandfeld gehört, speichert Q das aktuell auf dem Feld befindliche Zeichen x in ihrer endlichen Kontrolle. Im ersten und im letzten Fall der obigen Einteilung ist das ursprüngliche Symbol ein Leersymbol, während im

mittleren Fall jeweils ein Eingabesymbol gelesen und gespeichert wird. Dieses Eingabesymbol steht Q auf ihrem eigenen Eingabeband zur Verfügung.

Die grundsätzliche Idee der Simulation ist es, eine Kreuzungsfolge c_i auf der Queue zu speichern und die Kreuzungsfolge c_{i+1} (die von c_i durch ein \$ getrennt wird) in zu S's endlicher Kontrolle konsistenter Art zu raten. Genauer haben wir die die folgenden sich nicht gegenseitig ausschließenden Fälle, wobei der in der Queue verbleibende Suffix von c_i mit c bezeichnet wird (dagegen ist c' der Teil der Kreuzungsfolge, der von dem aktuell simulierten Schritt nicht betroffen ist):

- $c = \overrightarrow{q_1}\overleftarrow{q_2}c'$ und es existiert eine Transition von q_1 nach q_2 die x liest, ein Symbol y schreibt, und den Kopf nach links bewegt. Dann entfernt Q die Teilfolge $\overrightarrow{q_1}\overleftarrow{q_2}$ von der Queue und ersetzt x durch y.

- $c = \overrightarrow{q_1}c'$ und es gibt eine Transition von q_1 von q_2 die x liest, ein Symbol y schreibt, und den Kopf nach rechts bewegt. Dann entfernt Q das Symbol $\overrightarrow{q_1}$ von der Queue, schreibt $\overrightarrow{q_2}$ und ersetzt x durch y.

- $c = \overleftarrow{q_2}c'$ und es existiert eine Transition von q_1 nach q_2 die x liest, ein Symbol y schreibt und den Kopf nach links bewegt. Dann entfernt Q das Symbol $\overleftarrow{q_2}$, schreibt $\overleftarrow{q_1}$ in die Queue und ersetzt x durch y.

- Es existiert eine Transition von q_1 nach q_2 die x liest, ein Symbol y schreibt und den Kopf nach rechts bewegt. Dann schreibt Q die Folge von Symbolen $\overleftarrow{q_1}\overrightarrow{q_2}$ in die Queue und ersetzt x durch y.

Wenn das letzte Symbol der aktuellen Kreuzungsfolge verarbeitet worden ist und keine weiteren Operationen gemäß dem letzten Fall der obigen Aufzählung erforderlich sind, wird das Markierungssymbol \$ aus der Queue entfernt, an ihrem anderen Ende wieder eingefügt, und der nächste Simulationszyklus beginnt.

Wird der Endzustand erreicht, dann wird kein Folgezustand in der Queue gespeichert, sondern die Tatsache, dass der Endzustand in der Rechnung vorkam, in der endlichen Kontrolle von Q gespeichert.

Die Simulation wird initialisiert, indem eine (möglicherweise leere) Folge von Zustandspaaren gemäß dem letzten Fall in die Queue eingefügt werden, wobei der ursprüngliche Bandinhalt das Leersymbol ist. Die Simulation schreitet in Phase 1 fort, bis Q rät, dass c_{-1} in der Queue gespeichert ist und Phase 2 der Simulation beginnen soll. In diesem Augenblick wird der Startzustand von S in die Queue als erstes Element von c_0 eingefügt. Nachdem c_0 zusammengestellt worden ist, wird in jedem Zyklus der Simulation ein Eingabesymbol gelesen, bis das letzte Eingabesymbol verarbeitet worden

ist und $c_{|w|}$ gebildet worden ist. Die Simulation wird in Phase 3 fortgesetzt, bis die Queue keine Symbole außer \$ enthält. Die Maschine Q akzeptiert, falls der Endzustand von S während der Simulation erreicht wurde.

Wenn S eine akzeptierende Berechnung besitzt, dann können die Kreuzungsfolgen, die in der Berechnung auftreten, der Reihe nach von Q in ihrer Queue gespeichert werden und führen zu einer akzeptierenden Rechnung der Queuemaschine. Umgekehrt kann die Folge der Queueinhalte einer akzeptierenden Berechnung, die sich jeweils nach einem Zyklus der Rechnung von Q ergeben, zu einer akzeptierenden Rechnung von S zusammengefügt werden. Die Anzahl der Schritte, die von S ausgeführt werden, ist gleich der Summe der Längen aller Kreuzungsfolgen. Für jedes Element einer Kreuzungsfolge führt Q eine konstante Anzahl von Schritten aus. Hieraus folgt die behauptete Zeitschranke. □

Eine Umkehrung der obigen Simulation in dem Sinne, dass jede Queuemaschine im Wesentlichen ohne Zeitverlust durch ein Turingmaschine mit einem Band simuliert werden kann, ist nicht möglich. Mit anderen Worten sind also Queuemaschinen im Allgemeinen effizienter als solche Turingmaschinen. Diese Aussage ergibt sich aus dem folgenden Beispiel.

Beobachtung 1 *Die Sprache $D^{\#} = \{w\#w \mid w \in \{0,1\}^*\}$ kann in Realzeit von einer deterministischen Maschine mit einer Queue akzeptiert werden, aber nicht in $o(n^2)$ Schritten von einer Turingmaschine mit einem Band.*

Beweis. Die Technik von Barzdin' [2] und Hennie [23] zum Beweis unterer Schranken für Turingmaschinen mit einem Band zeigt, dass $D^{\#}$ von einer solchen Maschine nicht in $o(n^2)$ Schritten akzeptiert werden kann, siehe auch [75, Theorem 8.13]. Auf der anderen Seite kann eine Maschine mit einer Queue alle Symbole bis zum ersten # auf der Queue speichern und sie dann mit der Zeichenkette nach dem # vergleichen. Die Eingabe wird abgelehnt, wenn hierbei ungleiche Zeichen gefunden werden, die Längen nicht übereinstimmen, oder kein # vorhanden ist. Ansonsten wird die Eingabe akzeptiert. □

3.4 Simulation von mehreren Speichern durch Queuemaschinen

Satz 5 *Jede nichtdeterministische Maschine mit einer endlichen Zahl von d-dimensionalen Bändern, die $t(n)$ Schritte ausführt und dabei $s(n)$ Bandfelder benutzt, kann von einer nichtdeterministischen Maschine mit zwei Queues*

in $O(t(n)\log s(n))$ Schritten mit Platzbedarf $O(s(n))$ auf den Queues simuliert werden. Wenn die simulierte Maschine ihre Schranken im starken Sinne einhält (jede Rechnung kann innerhalb der Schranke ausgeführt werden), dann gilt dies auch für den Simulator.

Der Beweis des Satzes nimmt den Rest dieses Kapitels in Anspruch. Die gleiche Simulationstechnik kann wegen [5] auch von einer Maschine mit 3 Kellern, einem Band und einem Keller oder zwei Bändern durchgeführt werden.

3.4.1 Übersicht

Um die Darstellung der Simulation zu erleichtern, nehmen wir an, dass dem Simulator mehr als zwei Queues zur Verfügung stehen. Eine leicht veränderte Variante der Simulation von k Queues durch zwei [43] wird dann den Beweis abschließen.

Die Simulation verläuft in Phasen.

Wenn der Platz, der auf allen Bändern bis zum Beginn von Phase i benutzt worden ist, mit $\sigma(i)$ bezeichnet wird, dann rät Phase i die Berechnung für die nächsten $\sigma(i)$ Operationen (oder bis zum Ende der gesamten Berechnung) und die Queuemaschine schreibt die erratenen Operationen auf eine eigene Queue für jedes Band. Dann werden die Bandoperationen auf Konsistenz überprüft, wobei $O(\sigma(i))$ Platz und $O(\sigma(i)\log\sigma(i))$ Zeit aufgewendet wird. Am Ende der Phase wird der aktuelle Inhalt aller Bänder (in $\sigma(i+1)$ Platz) auf den Queues in $O(\sigma(i+1))$ Platz festgehalten. Die Simulation wird nur dann mit Phase $i+1$ fortgesetzt, wenn die nichtdeterministischen Schritte in Phase i eine gültige Rechnung ergeben. Die erste Phase führt die gleichen Schritte wie die simulierte Maschine bis zu dem Punkt aus, wenn die letztere das erste Symbol auf ein Band schreibt. Dann schreibt der Simulator das gleiche Symbol auf die Queue für den Bandinhalt zwischen den Phasen, und Phase 2 kann beginnen.

Die einzige Schwierigkeit der Simulation ist die Überprüfung der Konsistenz der erratenen Bandoperationen. Da dies für jedes Band getrennt durchgeführt werden kann, beschreiben wir hier nur die Behandlung eines einzigen Bandes.

Alle binären Zahlen, die in einer der Queues gespeichert werden, beginnen mit der niedrigstwertigen Ziffer. Wir schreiben length(i) für die Länge der binären Darstellung der Zahl i.

3.4.2 Datenformate

Wir erläutern nun einige Formate zur Speicherung von Daten, die mit einem *Pfad* auf einem der Bänder zusammenhängen. Ein Pfad ist die Folge von Bandpositionen, die von der jeweils vorhergehenden in jeder Dimension um höchstens 1 abweichen. Die Daten können entweder Bandoperationen sein (Paare der Form (x, y), wobei die Bedeutung ist, dass x gelesen und y geschrieben wird) oder einfach der aktuelle Bandinhalt. Wir bezeichnen die Länge des Pfades mit ℓ.

Rohformat

Im *Rohformat* werden die Datenelemente durch d-Tupel von Symbolen getrennt, welche die Änderungen der Adressen in jeder der d Dimensionen repräsentieren. Jedes dieser Symbole ist $+$, $-$ oder $\emptyset$.

Offensichtlich ist der Platzbedarf im Rohformat $O(\ell)$.

Markiertes Format

Das *markierte Format* weicht vom Rohformat dadurch ab, dass jedem Symbol, das eine Adressänderung repräsentiert, eine Markierung folgt, welche die Binärdarstellung der *Potenz* der Adressänderung ist. Die Potenz einer Adressänderung ist der Index der größten Zweierpotenz, die in der Repräsentation der Adresse durch die Operation verändert wird. So hat zum Beispiel die Erhöhung von 39 auf 40 die Potenz 3, weil das Bit mit der Wertigkeit 2^3 verändert wird, aber alle höherwertigen Binärstellen unverändert bleiben.

Das markierte Format hängt vom Startpunkt eines Pfades ab. Eine Startadresse kann für jede Dimension gewählt werden, so dass mindestens die Hälfte der Änderungen die Potenz 0 haben, mindestens die Hälfte der verbleibenden die Potenz 1 usw. Mit dieser Wahl einer Startadresse ist der für das markierte Format erforderliche Platz $O(\ell)$, sogar wenn die Markierungen unär dargestellt würden.

Quadtree-Format

Ein *Block der Seitenlänge* 2^k ist ein d-dimensionaler Hyperwürfel von Bandpositionen, so dass in jeder Dimension der Adressbereich $[j2^k \cdots (j+1)2^k - 1]$ für ein j ist.

Das *Quadtree-Format* [12] einer Folge von Daten ist ihre Kodierung innerhalb des kleinsten Blockes, der alle Bandfelder des Pfades enthält.

Die Repräsentation einer Folge von Daten *innerhalb* eines Blockes der Seitenlänge 2^k ist

- leer, falls keine der Bandpositionen innerhalb des Blockes liegen,
- eine Liste (in der gleichen Anordnung wie in der Operationsfolge) von allen Daten-Elementen einer Position in dem Block, falls $k = 0$,
- anderenfalls eine Liste der 2^d Repräsentation der Folgen innerhalb der eingeschlossenen Teilblöcke der Seitenlänge 2^{k-1}, in einer festen Ordnung und jeweils durch Klammersymbole zusammengefasst und gegeneinander abgegrenzt.

Während jeweils 2^k aufeinanderfolgender Operationen der Berechnung kann die Adresse der Kopfposition nur ein Vielfaches von 2^k in jeder Dimension überstreichen, so dass sie innerhalb einer Menge von 2^d Blöcken der Seitenlänge 2^k bleibt. Daher ist, unabhängig von der Startadresse eines Pfades, die Anzahl der Blöcke der Seitenlänge 2^k, die eine Bandposition auf dem Pfad enthalten, von der Ordnung $O(\ell/2^k)$, womit der Platzbedarf der Quadtree-Darstellung $O(\ell)$ ist.

Überdeckende Pfade

Die von A. Hemmerling angegebene platzeffiziente Simulation von mehrdimensionalen Speichern beruht auf der Existenz eines *überdeckenden Pfades* der Länge $2ds(n)$, der durch alle $s(n)$ in einer zusammenhängenden Folge von Bandoperationen betroffenen Bandpostionen verläuft, siehe [75, Theorem 5.2]. Die Repräsentation eines überdeckenden Pfades eines Bandinhalts ist die Darstellung im Rohformat eines solchen Pfades, wobei die Daten entweder den aktuelle Inhalt einer Bandzelle kodieren (dies trifft genau einmal für jede Position zu, die benutzt wird) oder $\emptyset$ (für alle anderen Zeitpunkte, zu denen eine Position besucht wird). Der Platzbedarf einer solchen Darstellung ist offensichtlich $O(s(n))$.

3.4.3 Transformationen

Zu Beginn der Phase i enthält eine Queue die aktuelle Beschriftung des Bandes in Form eines überdeckenden Pfades. Eine weitere Queue enthält die Binärkodierung der Adresse der aktuellen Position des Kopfes (in jeweils allen Dimensionen) relativ zu dem Anfang des überdeckenden Pfades. Während Schritte erraten werden, schreibt der Simulator Operationskodes im Rohformat auf die entsprechende Queue. Die nicht-$\emptyset$ Einträge im überdeckenden Pfad werden gelesen, um sicherzustellen, dass die Anzahl der geratenen Bandoperationen $\sigma(i)$ ist (oder weniger, falls die geratene Berechnung hält). Dann

wird ein d-Tupel von Anfangsadressen (jede mit der Länge length($\sigma(i)$) erraten, auf eine andere Queue geschrieben und zu den aktuellen Adresskomponenten addiert, um die *Phasen-Startadresse* zu erhalten. Nun werden in length($\sigma(i)$) Durchläufen die erratenen Operationen in markiertes Format konvertiert. Nach jedem Durchlauf, unter der Voraussetzung, dass „gute“ nichtdeterministische Schritte ausgewählt wurden, übersteigt die Größe der resultierenden Darstellung nicht $\kappa\sigma(i)$ mit der Konstanten $\kappa = O(6d + 1)$ (für die Wahl von siehe κ siehe letzten Paragraphen dieses Unterkapitels).

In ähnlicher Weise wird dann die Darstellung des überdeckenden Pfades konvertiert.

Danach werden die Kodierungen in markiertem Format in Quadtree-Format umgewandelt. Dann können die beiden Quadtrees auf Konsistenz überprüft werden und zur Bildung einer neuen Kodierung des Bandinhalts (im Quadtree-Format) zusammengeführt werden. Schließlich wird der Quadtree in die Rohformat-Darstellung eines überdeckenden Pfades zurückkonvertiert und die Daten sind für den Start der nächsten Phase vorbereitet.

Die in Phase i geratene Startadresse wird in drei Transformationen benutzt, in denen eine „gute“ Auswahl zur Sicherstellung der Platzeffizienz notwendig ist. Diese Transformationen sind die Konvertierung von überdeckendem Pfad in Quadtree-Format am Anfang *und* Ende von Phase i und die Konvertierung von Rohformat zu Quadtree-Format für die Bandoperationen in Phase i. Da die Gesamtlänge der drei Pfade höchstens $(6d + 1)\sigma(i)$ ist, können wir die Adressen so wählen, dass der gesamte Platzbedarf über alle drei Vorgänge $O((6d+1)\sigma(i)) = O(\sigma(i))$ ist und somit der Platzbedarf jedes einzelnen Prozesses ebenfalls von dieser Größenordnung ist.

Rohformat nach markiertem Format

Für jede Dimension werden wiederholte Durchläufe durch die die Datenqueue gemacht. Während des i-ten Durchlaufs werden Markierungen $i - 1$ zu allen Adressänderungen der Potenz $i - 1$ hinzugefügt. Um dies zu erreichen, wird das nächste Bit der relevanten Startadresse gelesen (die Phasen-Startadresse für die Bandoperationen oder einfach die erratene Startadresse für den Bandinhalt) und auch die Daten werden gelesen, wobei der Wert modulo 2 der Startadresse plus Anzahl der unmarkierten Veränderungen, die stattfinden, gespeichert wird. Jede Erhöhung, die den Wert von 0 auf 1 verändert, oder Verminderung von 1 nach 0 wird mit $i - 1$ markiert.

Markiertes Format nach Quadtree-Format

Wir gehen in length($\sigma(i)$) Durchläufen vor. Nach j Durchläufen sind die Klammersymbole für die obersten j Ebenen des Quadtrees konstruiert worden, aber auf der nächsten Ebene werden die Daten innerhalb des Blocks (mit Seitenlänge $2^{\log_2 \sigma(i)-j} = 2^{j'}$) einfach durch 0 oder mehr Teilpfade dargestellt, welche die verschiedenen Zeiträume repräsentieren, während derer der Kopf den Block durchlaufen hat. Vor jedem Teilpfad steht seine Startadresse relativ zur Position des Blockes. Im nächsten Durchlauf werden die Daten in die entsprechende von 2^d Queues geschrieben, die den 2^d Unterblöcken zugeordnet sind. Übergänge zwischen den Unterblöcken werden an den Markierungen der Adressänderungen erkannt. Nachdem ein Block gelesen wurde, werden die Unterblöcke an das Ende der Haupt-Datenqueue angefügt.

Dadurch, dass die Länge der relativen Adresse für jeden Teilpfad die Potenz der Adressänderung ist, welche die Überschreitung der Blockgrenze verursacht hat, ist der Platzbedarf stets $O(\sigma(i))$.

Addressspeicherung Während eines Durchlaufs durch einen Block wird die aktuelle Adresse in jeder Dimension mod $2^{j'}$ durch zwei Queues in der folgenden Weise repräsentiert. Eine *binäre* Queue enthält die Adresse eines früheren Datenelements in binärer Kodierung. Eine andere, *unäre* Queue enthält die Änderung der Adresse seit diesem früheren Datenelement in unärer Kodierung ($+$ oder $-$ Symbole entsprechen Erhöhung oder Verminderung der Adresse).

Auswahl eines Teilblocks Es ist leicht, die Adressinformation zwischen zwei Einträgen auf eine Queue, die temporäre Informationen aufnimmt, zu kopieren und zu entscheiden, welche der 2^d Zielqueues das nächste Datenelement aufnehmen soll. Wenn dieses der Anfang eines Blockes ist, dann hängt das Ziel von den höchstwertigen Bits der Startadressen ab. Anderenfalls ist es das gleiche wie für das vorige Datenelement, falls nicht eine Markierung gleich j' ist.

Der Test, ob eine Markierung gleich j' ist, ist nicht trivial, wenn wir sicherstellen wollen, dass dies in linearer Zeit bezüglich der Länge der Markierung durchgeführt wird. Eine zusätzliche Queue J enthält j' in Binärdarstellung mit einem Separator # nach den Ziffern. Während die Markierung gelesen und auf die temporäre Queue kopiert wird, wird sie ebenfalls auf eine weitere Test-Queue kopiert, gefolgt von einem #. Dann werden die Ziffern auf dieser Test-Queue rotiert, während ebenfalls J auf der Suche nach dem Separator rotiert wird. Falls der Separator nicht gefunden wird, kann sofort geschlossen werden, dass die Markierung ungleich j' ist, da die Test-Queue kürzer

war. Anderenfalls wird die Rotation fortgesetzt, bis das # in der Test-Queue gefunden wird, und dann wird ihr Inhalt Ziffer für Ziffer mit J verglichen. Diese komplizierte Vorgehensweise ist notwendig, weil der Separator in J möglicherweise seit dem letzten (erfolglosen) Vergleich noch nicht zum Beginn der Queue zurückgekehrt ist, und es nicht möglich ist, darauf zu warten, wenn die Markierung viel kürzer als j' ist.

Bevor das nächste Datum (x, y) auf die entsprechende Queue geschrieben wird, muss die korrekte Adressänderung festgehalten werden.

Aktualisierung von Adressen Wenn das nächste Datenelement aus dem gleichen Block stammt, dann wird die unäre Queue durch Lesen oder Schreiben eines Symbols verlängert oder verkürzt, um die entsprechende Änderung der Länge zu bewirken.

Wenn das nächste Datenelement zu einem anderen Teilblock gehört, dann muss die binäre Queue mit den Änderungen, die auf den unären Queues gespeichert sind, aktualisiert werden. Um dies zu erreichen werden die unären Queues wiederholt rotiert, dabei die Parität ihrer Länge bestimmt und die Anzahl der Symbole halbiert (wobei nach unten gerundet wird). Jede so bestimmte Binärziffer wird zu der aktuellen Ziffer der binären Queue addiert bzw. von ihr subtrahiert, wobei alte Überträge berücksichtigt und neue in der endlichen Kontrolle des Simulators gespeichert werden. Schließlich wird der möglicherweise verbleibende Teil der binären Queue zur Verarbeitung von Überträgen vollständig rotiert. Die insgesamt erforderliche Zeit ist linear in der Summe der Längen der beiden Queues. Folglich ist der Zeitaufwand auch linear im Verhältnis zu den Operationen, die Symbole in die Queues eingefügt haben.

Die Adressen, die in dieser Weise am Ende des einzigen Blocks in Phase 1 erzeugt worden sind, werden als neue aktuelle Adressen (relativ zu den erratenen Startadressen) gespeichert.

Schreiben der Adressänderungsinformationen Die Änderungsinformation, die auf die temporäre Queue geschrieben worden ist, kann aktualisiert und auf die korrekte Zielqueue transferiert werden. Wenn das nächste Datenelement in den gleichen Block fällt, dann wird die Information unverändert kopiert. Wenn es in einen anderen Teilblock fällt, dann wird die Information von der binären Queue kopiert.

Im Falle des ersten Datenelements in einem Block sind die Adressen modulo $2^{j'}$ vorangestellt. Das höchstwertige Bit wird von jeder Adresse entfernt und die übrige Information auf die Zielqueue und die binäre Adressqueue kopiert. Außerdem werden die unären Queues gelöscht.

Quadtree-Format nach Rohformat des überdeckenden Pfades

Nachdem die beiden Quadtrees auf interne und wechselseitige Konsistenz geprüft worden sind, werden sie zu einem Quadtree, der den aktuellen Bandinhalt kodiert, zusammengeführt. Danach wird ein neuer überdeckender Pfad für den Bandinhalt in Rohformat geraten (wobei sichergestellt wird, dass die Gesamtlänge der geratenen Kodierung tatsächlich höchstens die Länge $2d$ mal die Summe der Einträge des Quadtrees hat). Von dieser Kodierung wird eine Kopie erstellt, wie oben in Quadtree Format konvertiert, und überprüft, dass dieser Quadtree identisch mit dem bereits existierenden ist. Um die Konvertierung durchzuführen, ist es auch nötig, die Startadresse des überdeckenden Pfades (relativ zur aktuellen Startadresse) zu erraten. Dies zusammen mit der vorher berechneten aktuellen Adresse ergibt die Adresse für die nächste Phase.

Dies schließt die Vorbereitung der nächsten Phase ab.

3.4.4 Zeit- und Platzbedarf der Simulation

Wir haben gesehen, dass für eine passende Zahl κ, unter der Voraussetzung „guter“ nichtdeterministischer Schritte, die Längen der Queues durch $\kappa \times \sigma(i-1)$ und daher $O(s(n))$ begrenzt sind. Es bleibt zu zeigen, dass die Auswahl „schlechter“ Alternativen dazu führt, dass die Simulation terminiert ohne die Schranken zu verletzen. Die ist leicht zu erreichen, indem eine zusätzliche „Uhr“ in Form einer Queue C eingeführt wird. Ursprünglich enthält C lediglich eine Markierung. Nach Phase i werden in einem Durchlauf durch die Queue, welche den Bandinhalt aufnimmt, für jede der $\sigma(i)$ Bandpositionen κ Symbole $\emptyset$ auf C geschrieben. Während jedes Durchlaufs, der eine Queue beschreibt, wird C verwendet, um sicherzustellen, dass höchstens $\kappa\sigma(i)$ Symbole auf diese Queue gelangen, indem zuerst die Markierung auf C gesucht wird, und dann für jedes zusätzliche Symbol ein $\emptyset$ von C gelesen und sofort zurückgeschrieben wird. Falls die Markierung von C dabei wieder angetroffen wird, wird die Rechnung abgebrochen.

In Phase i werden $O(\log \sigma(i))$ Durchläufe durch die Daten gemacht, welche einen Umfang von $O(\sigma(i))$ haben. Daher kann Phase i in $O(\sigma(i) \log \sigma(i))$ abgeschlossen werden, wobei $\Omega(\sigma(i))$ Operationen simuliert werden. Daher ist der gesamte Zeitaufwand $O(t(n) \log s(n))$ und, falls die Schranken t und s auch für erfolglose nichtdeterministische Rechnungen gelten, gilt dies auch für die Schranken der Simulation.

3.4.5 Simulation von mehreren Queues durch zwei

Li et al. [43] beschreiben eine einfache nichtdeterministische Simulation von k Queues durch zwei Queues in linearer Zeit. Wir erläutern eine kleine Veränderung dieser Simulation, durch die *alle* nichtdeterministischen Berechnungen des Simulators linear in der Länge der simulierten Rechnung beschränkt sind.

Die Maschine mit zwei Queues (hier mit a und b bezeichnet) arbeitet in Phasen. Am Anfang einer Phase enthält Queue a den aktuellen Inhalt aller simulierten Queues, jeweils durch Markierungen getrennt.

Nichtdeterministisch wird eine Anzahl von Schritten der simulierten Maschine geraten, die gleich dem Platzbedarf auf der Queue a ist. Während dies geschieht, wird die Kodierung der Operationen auf Queue b geschrieben. Danach wird der Inhalt der Queue b an Queue a angefügt. Dann wird in k Durchläufen durch die Queue a die Konsistenz der Operationen für jeweils eine der simulierten Queues überprüft und der neue Inhalt auf Queue a zurückgeschrieben. Um die Konsistenz für die i-te simulierte Queue zu testen, wird zunächst ihr bisheriger Inhalt auf Queue b gebracht und dann die erratenen Operationen für Queue i auf Queue b des Simulators ausgeführt. Kann diese Simulation erfolgreich abgeschlossen werden, dann wird der Inhalt von Queue b zurück an die korrekte Position auf Queue a gebracht.

3.5 Simulation von Maschinen mit mehreren Speichern durch hybride Maschinen

Während wir bisher Simulationen durch Maschinen angegeben haben, die mit einer oder mehr Queues ausgestattet sind, wenden wir uns nun der Kombination einer Queue mit einem anderen Speicher zu, also einer hybriden Speicherstruktur.

Hühne bemerkt in [28], dass seine deterministische Simulation von Maschinen mit mehreren Speichern durch eine Maschine mit zwei Queues in $O(t(n)\sqrt{t(n)})$ Schritten durch eine Maschine mit einer Queue und einem Keller durchgeführt werden kann. Er führt eine untere Schranke $\Omega(t(n)\sqrt[4]{\log t(n)})$ für die letztgenannte Simulation an. Im Falle nichtdeterministischer Maschinen können eine Queue und ein Keller jede endliche Anzahl von Kellern (und damit auch Bänder) in linearer Zeit simulieren. Dazu wird die Technik von Book und Greibach [5] folgendermaßen angepasst: Eine Folge von partiellen Konfigurationen, die den Zustand der endlichen Kontrolle der simulierten Maschine, das oberste Symbol auf jedem der simulierten Keller, das aktuell gelesene Eingabesymbol und die Operationen von Eingabekopf und Speicher

wird geraten. Diese Folge wird auf die Queue geschrieben. Dann überprüft der Simulator, dass die Folge einer zulässigen Berechnung im Bezug auf jeden der beteiligten Kellerspeicher und die Eingabe entspricht.

Wir geben hier eine deterministische Simulation einer beliebigen Anzahl von Bändern durch eine Queue und einen Keller an, die fast die erwähnte untere Schranke erreicht.

Satz 6 *Jede deterministische durch $t(n)$ zeitbeschränkte Turingmaschine mit mehreren linearen Bändern kann durch eine deterministische Maschine mit einer Queue und einem Keller in $O(t(n) \log t(n))$ Schritten simuliert werden.*

Beweis. Wir bemerken zunächst, dass ein Band mit linearem Aufwand durch zwei Keller in naheliegender Weise simuliert werden kann. Wir können daher die Beschreibung auf die effiziente Simulation einer endlichen Anzahl von Kellern durch eine Queue und einen Keller beschränken.

Die Idee der Simulation ist es, den Inhalt mehrerer Keller in getrennten Spuren des einen Kellers des Simulators zu speichern. Jedes Feld auf diesen Spuren kann leer sein oder Information des simulierten Kellers enthalten. Die Konkatenation aller nicht-leeren Zellen auf Spur k (von obersten Symbol bis zum Kellerboden) bildet die Zeichenkette, die auf dem k-ten simulierten Keller gespeichert ist. Die Verteilung der Leerfelder auf diesen Spuren kann variieren und hängt von der Zugriffsreihenfolge auf die Keller ab. Kellerbodensymbole der simulierten Speicher werden so wie andere Symbole des Kelleralphabets behandelt.

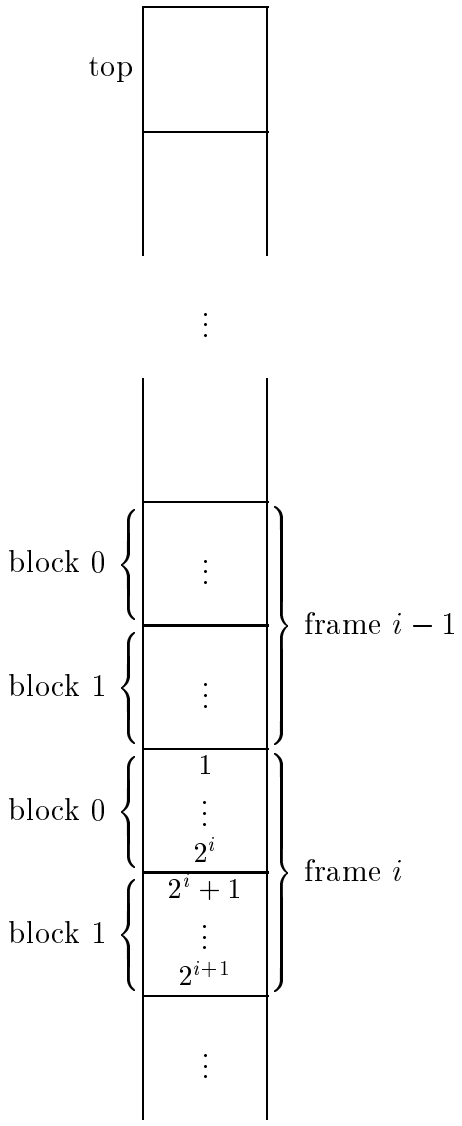
Um die Beschreibung der Simulation zu vereinfachen, konzentrieren wir uns auf einen Keller und erläutern die Veränderungen der Beschriftung der entsprechenden Spur während der Simulation von push- und pop-Operationen. Der Inhalt der übrigen Spuren wird unverändert gelassen, außer der möglichen Einfügung von neuen Leersymbolen über dem Kellerboden des Simulators.

Jede Spur ist in eine spezielle, oberste Zelle und eine potenziell unendliche Folge von *Frames* wachsender Größe eingeteilt, wobei Frame i eine Anzahl von 2^{i+1} Zellen enthält ($i \geq 1$). Jeder Frame ist wiederum in zwei Blöcke gleicher Größe unterteilt. Wir nehmen an, dass diese Strukturierung leicht erkennbar ist, z.B. durch Markierungen an den Grenzen von Frames und Blöcken.

Eine Invariante, die von den Phasen der Simulation aufrecht erhalten wird, ist es, dass ein Frame entweder leer ist, einen vollen Block 0 und einen leeren Block 1 hat, oder zwei gefüllte Blöcke hat. Die Anzahl der Frames ist für alle Spuren gleich und das Kellerbodensymbol des Simulators markiert den letzten Frame.

Die Simulation beginnt mit einem leeren Keller (außer dem obersten Symbol und dem Kellerbodensymbol). Auch die Queue ist leer.

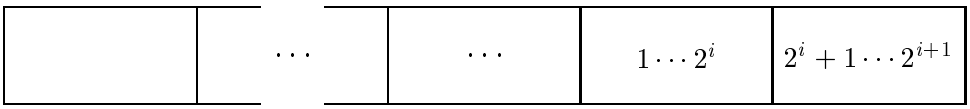
Wir beschreiben zunächst die Simulation der pop-Operation. Sei der i-te Frame der erste nicht-leere. Auf Grund der Invariante ist Block 1 voll und Block 0 voll oder leer. Der Zustand des Kellers wird durch das folgende Bild veranschaulicht, wobei ein Teil der Positionen nummeriert ist, um den Zusammenhang zum späteren Inhalt der Queue zu verdeutlichen.



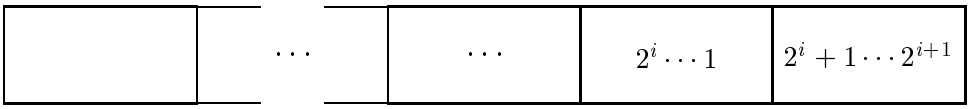
Phase 1 (Lesen des Speichers): Der Simulator verschiebt den Inhalt des Kellers vollständig auf die Queue bis er den ersten nicht-leeren Frame gelesen hat. Wir nehmen an, dass die Struktur des Speichers bei dieser Verschiebung erkennbar bleibt.

Phase 2 (Umverteilung): In dieser Phase nutzt der Simulator den Keller als Hilfsspeicher, wobei der auf dem Keller verbliebene Inhalt von den Veränderungen innerhalb der Phase geschützt wird. Dies ist möglich, indem ein Markierungssymbol auf dem Keller abgelegt und am Ende der Phase wieder entfernt wird. Auf diese Weise ist der Inhalt des Kellers am Ende der Phase identisch zu seinem Inhalt am Beginn der Phase.

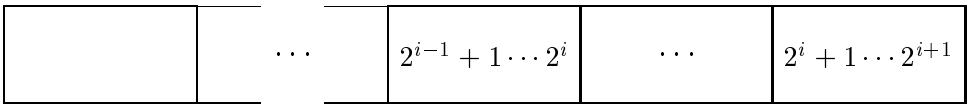
Der aktuelle Zustand der Queue direkt vor der Umverteilung sieht folgendermaßen aus:



Zunächst rotiert der Simulator die Queue bis er die Zeichenkette erreicht, die vorher im nicht-leeren Frame gespeichert war. Dann kopiert er den Inhalt von Block 0, falls er nicht leer ist – oder anderenfalls Block 1 – auf den Keller. Dann unternimmt der Simulator einen weiteren Durchlauf durch die Queue und überschreibt diesen Teil der Queue mit dem Spiegelbild seines ursprünglichen Inhalts, wobei der Keller geleert wird:

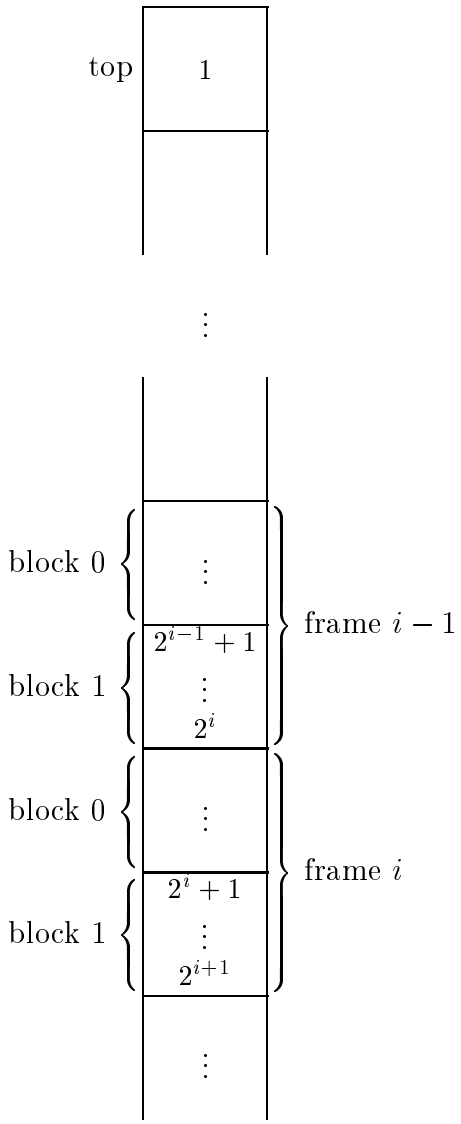


Der Simulator kopiert nun wiederum die gespiegelte Zeichenkette auf den Keller, wobei die gelesenen Zellen gelöscht werden. Dann schreibt er ein Symbol an die Position der ersten Zelle in der Queue und, während die Queue rotiert wird, 2^i Symbole von der Kellerspitze in die ersten Blöcke der Frames, die sich auf der Queue befinden. Da $2^i = 1 + \sum_{k=0}^{i-1} 2^k$ ist genau die erforderliche Anzahl von Zellen vorhanden, um alle Symbole unterzubringen und den Keller (bis zur Markierung) vollständig zu leeren, indem 2^k Symbole in Block 1 von Frame k geschrieben werden. Jeder Block, der Symbole aus dem Keller erhält, ist damit voll. Somit gilt die Invariante nach Ausführung dieser Operationen.



Schließlich transformiert der Simulator die gesamte Spur in ihr Spiegelbild.

Phase 3 (Schreiben des Speichers): Die Markierung, welche die nicht an der Simulation beteiligte Sektion des Kellers abgetrennt hatte, ist am Ende der vorherigen Phase entfernt worden. Daher kann die Queue entleert werden, wobei ihr Inhalt auf dem Keller gespeichert wird. Zu beachten ist, dass die ursprüngliche Reihenfolge der Frames durch Bildung des Spiegelbildes des Queue-Inhalts am Ende der vorigen Phase erhalten bleibt. Im Beispiel erhalten wir:



Dies schließt die Simulation der pop-Operation ab.

Für die push-Operation ist die Situation etwas komplizierter, da die aktuell vorhandene Anzahl von Frames auf dem Keller möglicherweise nicht ausreicht, um ein weiteres Symbol unterzubringen. Im Folgenden beschreiben wir die Simulation einer push-Operation wiederum in drei Phasen, wobei wir auf die entsprechenden Phasen der pop-Operation verweisen, falls es gemeinsame Abschnitte gibt.

Phase 1 (Lesen des Speichers): Der Simulator prüft, ob die oberste Zelle leer ist. Wenn ja, dann füllt er diese Position mit dem neuen Symbol, überspringt die folgenden Phasen und beendet die Simulation der push-Operation. Anderenfalls liest er den Inhalt des Kellers und schreibt ihn auf die Queue, bis er einen Frame findet, der nicht vollständig gefüllt ist oder der Keller leer ist. In letzterem Fall legt der Simulator einen neuen Frame auf der Queue an. Dies erreicht er durch Zählen der aktuellen Länge ℓ der Queue auf dem Keller und Einfügen von $\ell + 1$ neuen Zellen (für jede Spur).

Phase 2 (Umverteilung): Der verbleibende Inhalt des Kellers wird vor den kommenden Operationen in der gleichen Weise wie in der Simulation der pop-Operation geschützt. Sei Frame i derjenige, der noch nicht vollständig gefüllt ist. Der Simulator liest $2^{i+1} - 1$ Symbole aus der obersten Zelle des Kellers und den Frames, die vollständig gefüllt sind. Dann spiegelt er die so gelesene Zeichenkette wie oben beschrieben. Danach trägt er 2^k Symbole in Block 1 von Frame k für jedes $0 \leq k < i$ ein und die verbleibenden 2^i Symbole in Block 1 von Frame i, falls er leer ist. Anderenfalls nutzt er Block 0. Schließlich wird das zusätzliche Symbol in die oberste Zelle geschrieben.

Nach jeder Operation, die den Frame i betrifft, gibt es $2^i - 1$ leere Zellen oberhalb dieses Frames im Keller. Innerhalb von 2^i Schritten ist daher höchstens ein Zugriff auf Frame i möglich. Die Anzahl der Operationen in der oben beschriebenen Simulation ist proportional zur Länge der betroffenen Frames. Daher können wir die Zeitkomplexität der Simulation von $t(n)$ Schritten der Mehrband-Maschine durch

$$\sum_{k=0}^{\lfloor \log t(n) \rfloor} c2^k \cdot \lfloor t(n)/2^k \rfloor = O(t(n) \log t(n))$$

mit einer Konstanten c beschränken. Da die Anzahl der Spuren für eine gegebene Maschine fest ist, erhalten wir für die Zeitkomplexität der Simulation ebenfalls $O(t(n) \log t(n))$. □

Die im vorigen Beweis angewandte Simulation basiert auf der Idee, Speicherblöcke mit exponentiell wachsender Größe zu verwenden, welche auf die durch Hennie und Stearns [25] angegebene effiziente Simulation von mehreren Bändern durch zwei zurückgeht. Es ist bekannt, dass diese Simulation mit einem Band und einem Keller durchgeführt werden kann. Das gleiche gilt

für unsere Simulation, indem die Queue durch ein Band ersetzt wird und die entsprechenden Schritte angepasst werden. Eine direkte Anpassung der Simulation aus [25] für eine Maschine mit einer Queue und einem Keller scheint dagegen nicht möglich zu sein, da es die Struktur einer Queue nicht erlaubt, die Mehrheit der Speicherzugriffe in der Nähe eines zentralen Feldes durchzuführen. Sattdessen scheint es für eine Anpassung erforderlich, den gesamten Speicherinhalt zu rotieren, was zu einem quadratische Zeitverlust führen würde.

Kapitel 4

Untere Schranke der Simulation von linearem Speicher durch Queues

Wir geben in diesem Kapitel eine verbesserte untere Schranke für die deterministische Online-Simulation von Bändern oder Kellern durch Queues an. Hühne hat gezeigt, dass die Online-Simulation von $2(k+1)$ Kellern oder Bändern durch k Queues $\Omega(n^{1+1/k}/\log^{1/k} n)$ Schritte benötigt [28, Corollary 4.4] und dass die analoge Simulation von zwei Bändern oder drei Kellern $\Omega(n^{1+1/k}/\log^{1+2/k} n)$ Zeit erfordert [28, Corollary 4.5]. Der Beweis beruht auf quadratischen unteren Schranken für die Berechnung gewisser Funktionen durch Maschinen mit k Queues, und unser Argument nutzt die gleichen Funktionen wie der Beweis von Hühne. Der Unterschied ist, dass wir spezifische Algorithmen zur Berechnung dieser Funktionen auf den Modellen angeben, die für k Queuemaschinen schwer zu simulieren sind. Im Gegensatz dazu verwendet der Beweis von Hühne allgemeine Simulationsergebnisse, die einen größeren Aufwand bedingen.

Die Funktion F_k wird durch eine k Queuemaschine definiert, die sie in linearer Zeit berechnet. Der relevante Teil möglicher Eingaben – auf Eingaben anderer Form kann die Ausgabe beliebig sein – wird definiert durch

$$\begin{aligned} L_k \;=\; \{ & x_{1,1}\cdots x_{1,f_1}\#\cdots\#x_{i,1}\cdots x_{i,f_i}\#\cdots\#x_{k,1}\cdots x_{k,f_k}\$ \\ & x_{1,f_1+1}\cdots x_{i,f_i+1}\cdots x_{k,f_k+1}\$\cdots\$ \\ & x_{1,f_1+j}\cdots x_{i,f_i+j}\cdots x_{k,f_k+j}\$\cdots\$ \\ & x_{1,f_1+m}\cdots x_{i,f_i+m}\cdots x_{k,f_k+m}\$ \mid \\ & \quad \forall 1\le i\le k, 1\le j\le f_i+m : x_{i,j}\in\{0,1\}\}. \end{aligned}$$

Auf einer Eingabe aus der Menge L_k füllt eine k Queuemaschine M_k ihre i-

te Queue mit $x_{i,1} \cdots x_{i,f_i}$ aus dem Abschnitt der Eingabe vor dem ersten \$ (die Ausgabe in dieser Phase ist irrelevant). Nachdem das erste Zeichen \$ gelesen worden ist, arbeitet M_k in m Runden. In Runde j liest M_k die Symbole der j-ten Zeichenkette zwischen \$-Zeichen und fügt der Reihe nach die x_{i,f_i+j} der Queue i hinzu, während die ersten Symbole gelöscht und ausgegeben werden. Jedes \$ wird direkt ausgegeben. Dies hat zur Folge, dass die Längen der Queues konstant gehalten werden, nachdem das erste \$ gelesen wurde, und die Ausgabe $x_{1,1} \cdots x_{k,1}\$ \cdots \$x_{1,j} \cdots x_{k,j}\$ \cdots \$x_{1,m} \cdots x_{k,m}$ ist. Die Maschine M_k arbeitet in linearer Zeit und berechnet F_k.

Lemma 2 *Für jedes $k \geq 1$ kann die oben definierte Funktion F_k von einer deterministischen Turingmaschine mit einem Band und einem Keller in linearer Zeit berechnet werden.*

Beweis. Die Turingmaschine T_k, die F_k berechnet, benutzt k Spuren ihres Arbeitsbandes um die k Zeichenketten der Queues von M_k zu repräsentieren. Nachdem $x_{i,1} \cdots x_{i,f_i}$ gelesen und auf die i-te Spur kopiert worden ist, kehrt T_k zu dem ersten Feld des Bandabschnittes zurück, auf dem diese Zeichenkette gespeichert worden ist. Nach dem Lesen des ersten \$ beginnt T_k mit der Ausgabe der auf den Spuren gespeicherten Symbole und überschreibt sie mit neuen. Diese Symbole werden zusätzlich als neu markiert. Wenn die Zeichenkette auf Spur i vollständig ausgegeben worden ist, kehrt T_k zu dem ersten neuen Symbol auf dieser Spur zurück während die gelesenen Zeichen auf den Keller geschrieben werden, bewegt dann den Kopf zur Position wo das Lesen unterbrochen wurde, und schreibt den Inhalt des Kellers auf die folgenden Felder von Spur i. Der gelesene Abschnitt wird als alt markiert und das Lesen an der ursprünglichen Position wieder aufgenommen.

Die Anfangsphase, in der die Zeichenketten $x_{i,1} \cdots x_{i,f_i}$ gespeichert werden, kann in einer Anzahl von Schritten, die proportional zur Länge der Zeichenkette vor dem ersten \$ ist, abgeschlossen werden. Das Kopieren des Segmentes auf Spur i ist in $O(f_i)$ Schritten möglich, nachdem f_i Symbole ausgegeben worden sind. □

Wir können nun die verschärfte Version der unteren Schranke aus [28] formulieren.

Korollar 1 *Jede deterministische Online-Simulation eines Bandes und eines Kellers, von zwei Bändern oder drei Kellern durch k Queues erfordert einen Aufwand von $\Omega(n^{1+1/k}/\log^{1/k} n)$ Schritten.*

Beweis. Die übrigen beiden Varianten können ein Band und einen Keller in linearer Zeit simulieren. Daher ist es ausreichend, die untere Schranke für das letztgenannte Modell zu zeigen.

Nach Lemma 2 kann die Funktion F_{k+1} in linearer Zeit mit der Hilfe eines Bandes und eines Kellers berechnet werden. Der Beweis von Theorem 4.2 in [28] zeigt, dass jede k Queuemaschine, die F_{k+1} online berechnet, $\Omega(n^{1+1/k}/\log^{1/k} n)$ Schritte bei einer Eingabe der Länge n benötigt. □

Kapitel 5

Keller versus Deques

In diesem Kapitel vergleichen wir die Effizienz einer endlichen Anzahl von Kellern mit verschiedenen Varianten von Deques, d.h. Listen mit zweiseitiger Zugriffsmöglichkeit. Im nichtdeterministischen Fall können zwei Keller eine Deque in linearer Zeit simulieren. Dies hat eine negative Antwort auf die von Brandenburg gestellte Frage zur Folge, ob eine Deque endlich viele Bänder in linearer Zeit simulieren kann. Wir zeigen ebenfalls, dass in deterministischen Berechnungen eine ausgabebeschränkte Deque nicht in der Lage ist, zwei Keller in Realzeit zu simulieren. Es ist bekannt, dass eine allgemeine Deque durch drei Keller in linearer Zeit simuliert werden kann. Wir beschreiben einen Ansatz, der einfacher zu analysieren ist und einen kleineren konstanten Faktor (im Bezug auf erforderlichen Kelleroperationen) als eine ältere Lösung besitzt.

5.1 Einleitung

Funktionale Programmierung mit LISP-artigen Daten führt auf die Frage, ob bekannte Datenstrukturen wie Queues und Deques effiziente Implementierung besitzen, die auf den verfügbaren „einseitigen“ Listen oder Kellern basieren. „Effizient“ bedeutet für uns die Simulation in linearer Zeit oder sogar in Realzeit. Unter Realzeit wird in diesem Zusammenhang verstanden, dass das zeitliche Verhalten des Simulators gegenüber dem simulierten Modell durch eine datenunabhängige Konstante beschränkt ist. Die ist eine stärkere Einschränkung der möglichen Simulationsstrategien als die bloße Forderung einer Simulation in linearer Zeit. Im letzteren Fall können beispielsweise Aufräumarbeiten („garbage-collection“) mit im Verhältnis zur gespeicherten Datenmengen linearem Aufwand durchgeführt werden. Diese Phasen arbeiten aber nicht in konstanter, von der Vorgeschichte der Berechnung unabhängiger

Zeit und sind daher in Realzeit-Simulationen ausgeschlossen

Auf einer abstrakteren Ebene ist es seit langem bekannt, dass sequentieller Speicher mit mehreren Zugriffspunkten – und somit Deques und Queues – mit der Hilfe von sequentiellem Speicher und jeweils einem einzigen Zugriffspunkt effizient simuliert werden kann. Eine Lösung in linearer Zeit wurde in [69] angegeben, was in [14, 41] zu Realzeit verschärft wurde.

Da jedes lineare Band in zwei Keller zerlegt werden kann, zeigt die etwas überraschende Tatsache, dass (asymptotisch) kein Geschwindigkeitsverlust eintritt, wenn Queues und Deques in funktionalen Programmiersprachen realisiert werden. Konkrete Implementationen von Queues in einer funktionalen Sprache wurden durch Hood und Melville [26] angegeben, während Chuang und Goldberg dies auf Deques erweiterten [9]. Die bekannten Realzeit-Simulationen erfordern allerdings eine große Anzahl von Kellern per Queue oder Deque, die simuliert wird. Dagegen existiert eine seit langem bekannte Linearzeit-Simulation einer einzelnen Queue durch zwei Keller. Es ist ebenfalls bekannt, dass eine Deque in linearer Zeit durch drei Keller simuliert werden kann [64]. Fragen, die sich stellen, sind: Was ist die minimale Anzahl an Kellern, die zur Simulation von Queue bzw. Deque notwendig ist? Wird die Simulation einer Deque einfacher, wenn der Zugriff auf die Daten beschränkt ist?

Im Bereich formaler Sprachen wurden Automaten mit einer einzelnen Deque durch Ayers [1] untersucht. Die von ihr betrachteten Automaten arbeiten in Realzeit (in jener Arbeit als „Quasi-Realzeit“ bezeichnet). In seinen [1] betreffenden Kommentaren formulierte Brandenburg mehrere offene Probleme aus dem Themenkreis dieser Untersuchungen. Er fragte, ob die Hierarchie der Sprachfamilien, die durch nichtdeterministische Maschinen mit einer Queue, einer ausgabebeschränkten Deque, einer Deque und einer endlichen Anzahl von linearen Bändern in Realzeit (oder linearer Zeit) akzeptiert werden, echt ist. Bis auf die letzte Beziehung ist klar, dass diese Klassen aufgrund der Definition der Speicher in Inklusionsbeziehungen stehen. Auf die Deque kann die oben erwähnte Simulation durch mehrere Bänder angewendet werden. Die Anzahl der Bänder läßt sich dann ohne Zeitverlust auf zwei reduzieren [5]. Die von Brandenburg gestellten Fragen sind zum Teil beantwortet worden. Li e.a. [43] zeigten, dass im Allgemeinen selbst eine deterministische Maschine mit einem Keller (der offensichtlich durch eine ausgabebeschränkte Deque simuliert wird) nicht in linearer Zeit durch eine nichtdeterministische Maschine mit einer Queue simuliert werden kann. Eine ausgabebeschränkte Deque kann durch zwei Keller in linearer Zeit simuliert werden, was über die nichtdeterministische Simulation vieler Bänder durch drei Keller in Realzeit [5] und die untere Schranke aus [42, Corollary 3] zeigt, dass eine ausgabebeschränkte Deque schwächer ist als eine endliche Anzahl von Bändern.

Die Resultate, die wir in diesem Kapitel darstellen, konzentrieren sich auf den Vergleich einer kleinen Anzahl von Kellern mit Varianten der Deque.

5.2 Nichtdeterministische Simulation einer Deque durch zwei Keller

Wir betrachten zunächst nichtdeterministische Maschinen, und zeigen dass – bis auf einen linearen zeitlichen Mehraufwand – Maschinen mit einer einzigen Deque und solche mit zwei Kellern die gleiche Mächtigkeit besitzen.

Satz 7 *Jede $t(n)$ zeitbeschränkte nichtdeterministische Maschine mit einer Deque kann von einer $O(t(n))$ zeitbeschränkten nichtdeterministischen Maschine mit zwei Kellern simuliert werden.*

Beweis. Wir zerlegen die Simulation in zwei Phasen. Die erste Phase wird zu einer Simulation einer Deque durch eine Queue und zwei Keller führen, wobei der Zugriff auf die Queue eingeschränkt ist. Diese Simulation wird in Realzeit durchgeführt, wobei wir hierunter eine Simulation verstehen, die pro nachgebildetem Schritt nur eine konstante Anzahl von Schritten des Simulators verlangt, unabhängig von der Eingabelänge.

In der zweiten Phase werden die Queue und die Keller durch zwei Keller innerhalb der im Satz angegebenen linearen Schranke simuliert. Die zweite Phase ist *keine* Realzeit-Simulation, sondern benötigt in einer Stufe eine eingabeabhängige Berechnung, während keine Speicherzugriffe simuliert werden können.

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass die Deque leer ist, wenn die Berechnung endet. Wir behaupten, dass eine einzelne Deque durch eine Queue und zwei Keller S_L und S_R Schritt für Schritt wie folgt simuliert werden kann. Der Simulator beginnt mit einer leeren Queue und leeren Kellern. Er liest Eingabesymbole und führt Operationen auf dem Speicher in Übereinstimmung mit der endlichen Kontrolle der Deque-Machine aus. Zusätzlich befindet sich der Simulator in einem von zwei Arbeitsmodi. Wir nennen diese Modi links-rechts und rechts-links Modus (die Bedeutung dieser Namen wird weiter unten erklärt). Der anfängliche Modus kann beliebig festgelegt werden. Während der Simulation kann der Simulator von einem zum anderen Modus wechseln, wenn die Queue zu diesem Zeitpunkt leer ist. Die möglichen Deque-Operationen werden in der folgenden Weise simuliert:

Einfügen am linken Ende: Nichtdeterministisch wird entweder das neue Symbol auf S_L geschrieben oder, unter der Voraussetzung dass S_L leer

ist und die Simulation sich im Modus links-rechts befindet, in die Queue eingefügt.

Einfügen am rechten Ende: Nichtdeterministisch wird entweder auf S_R das neue Symbol geschrieben oder, unter der Voraussetzung dass S_R leer ist und die Simulation sich im Modus rechts-links befindet, in die Queue eingefügt.

Löschen am linken Ende: Falls S_L nicht leer ist, wird ein Symbol von diesem Keller gelesen. Wenn S_L leer ist, wird unter der Voraussetzung, dass der aktuelle Modus rechts-links ist, ein Symbol aus der Queue gelesen. Anderenfalls wird die Simulation abgebrochen und die Eingabe abgelehnt.

Löschen am rechten Ende: Falls S_R nicht leer ist, wird ein Symbol von diesem Keller gelesen. Wenn S_R leer ist, wird unter der Voraussetzung, dass der aktuelle Modus links-rechts ist, ein Symbol aus der Queue gelesen. Anderenfalls wird die Simulation abgebrochen und die Eingabe abgelehnt.

Test auf Leerheit: Überprüfung, dass beide Keller und die Queue leer sind.

Die Hauptinvariante der Simulation, die nach Durchführung aller zu einer Deque-Operation gehörenden Änderungen gilt, ist es, dass der aktuelle Inhalt der simulierten Deque sich aus der Konkatenation des Inhalts von S_L mit der Kellerspitze links, der Queue, entweder mit ihrem Ende links (im Modus links-rechts) oder mit dem Ende rechts (im Modus rechts-links) und dem Inhalt von S_R mit der Spitze rechts ergibt. Diese Invariante wird durch die zu den Deque-Operationen oben angegebenen Simulationen aufrecht erhalten. Zugriffe erfolgen stets auf die Enden der simulierten Deque. Diese Überlegung zeigt, dass jeder akzeptierenden Rechnung des Simulators eine solche in der simulierten Deque-Maschine entspricht. Es ist ebenfalls klar, dass die Anzahl der durch den Simulator für einen Zugriff auf die Deque ausgeführten Operationen konstant ist.

Umgekehrt müssen wir nun zeigen, dass *jede* akzeptierende Rechnung der simulierten Deque-Maschine eine Entsprechung auf der Seite des Simulators findet. Wir halten eine solche Rechnung der simulierten Maschine fest und nehmen für den Rest des Beweises auch an, dass die gespeicherten Symbole unterscheidbar sind. Mit jedem Symbol, das auf der der Deque gespeichert wird, können wir zwei Boole'sche Variablen verbinden. Dabei ist es nicht notwendig, dass diese Information tatsächlich gespeichert wird. Es handelt sich lediglich um eine Hilfsinformation bei der Begründung der Korrektheit der

Simulation. Die erste Variable enthält die Information, ob das Symbol am linken oder rechten Ende der Deque eingefügt worden ist. Die zweite Variable speichert, ob das Symbol die Deque am linken oder rechten Ende verlassen wird. Da wie annehmen, dass die Deque schließlich leer sein wird, ist diese Information für jedes Symbol in einer festen Rechnung definiert. Die Symbole werden stets in einer nach beiden Komponenten sortierten Reihenfolge in der Deque vorliegen, was zu der wichtigen Beobachtung führt, dass in jeder Konfiguration der Deque während einer Simulation Symbole höchstens eines der „gemischten“ Modi (links-rechts oder rechts-links) vorliegt. Hiermit ist gemeint, dass ein Symbol mit unterschiedlichen Variablenwerten für Eintritt in die Deque und Austritt aus ihr Symbole mit den komplementären Werten ausschließt. Wir ordnen nun jeder Konfiguration der Deque den Modus ihrer gemischten Symbole zu (oder einen beliebigen Modus, falls es keine solchen Symbole gibt). Der Simulator folgt in seiner Arbeit nun der Information, die den Symbolen zugeordnet ist, und arbeitet in dem Modus der durch den Modus der Konfiguration der Deque vorgegeben wird. Dies kann der Simulator erreichen, indem er links-links Symbole auf S_L unterbringt, rechts-rechts Symbole auf S_R schreibt und alle Symbole mit gemischtem Modus (links-rechts oder rechts-links) auf die Queue. Es ist klar, dass sich der Modus der Queue nur ändern kann, wenn es kein Symbol mit gemischtem Modus gibt. Dann ist die Queue des Simulators leer und er ist gemäß der oben gegebenen Beschreibung in der Lage, ebenfalls seinen Modus anzupassen. Wenn die Deque Symbole enthält, die am linken Ende entnommen werden sollen, dann sind sie in der Simulation entweder auf S_L abgelegt, wo sie ständig zur Verfügung stehen, oder es handelt sich um Symbole mit gemischtem Modus, und dann befinden sie sich an der Spitze der Queue, wo sie für den Simulator ebenfalls zugreifbar sind. Eine analoge Argumentation zeigt, dass Löschungen am rechten Ende korrekt nachgebildet werden können.

Nach der obigen Darstellung kann jede Berechnung der Deque-Maschine durch den mit einer Queue und zwei Kellern ausgestatteten Simulator nachvollzogen werden (es besteht lediglich die Möglichkeit, durch falsche Zuordnung der Modi in ablehnende Simulationen zu gelangen) und jeder Lauf des Simulators entspricht einer Rechnung der Deque-Maschine.

Im Folgenden zeigen wir, wie die zwei Keller und eine Queue der obigen Simulation durch nur zwei Keller nachgebildet werden können.

Die simulierten Keller werden einfach den beiden Kellern des Simulators zugeordnet, wobei der Kellerboden zusätzlich mit einer Markierung gekennzeichnet wird. Unter diesen Bodenmarkierungen werden zwei Zeichenketten gespeichert, wobei die kürzere ein Suffix der längeren sein muss (bzw. bei Längengleichheit beide Ketten identisch sein müssen), falls die Simulation zu einem akzeptierenden Abschluss gebracht werden soll. Wenn die simu-

lierte Maschine im links-rechts Modus ist, dann ist die auf S_L gespeicherte Zeichenkette mindestens so lang wie die auf S_R und entsprechend umgekehrt im rechts-links Modus. Wenn ein Symbol in der simulierten Queue gespeichert werden soll, dann wird dies Symbol durch den Simulator der „längeren“ Zeichenkette hinzugefügt, wenn ein Symbol zu löschen ist, dann wird es an die kürzere angefügt. Dies ist im Verlauf einer Simulation immer möglich, da zu den Zeitpunkten des Zugriffs auf die Queue der entsprechende Keller leer ist, was durch die eingeschränkte Verwendung der Datenstrukturen in der ersten Phase der Simulation gewährleistet wird.

Wenn schließlich ein akzeptierender Zustand der simulierten Maschine erreicht worden ist, dann werden die Inhalte der beiden Keller Symbol für Symbol in $O(t(n))$ Schritten verglichen. Die Eingabe wird akzeptiert, wenn die Zeichenketten identisch sind.

Die bis jetzt beschriebene Strategie der Simulation erlaubt es, jede gültige Berechnung nachzuvollziehen. Es sind aber auch ungültige Rechnungen möglich, die sich aus zwei Gründen ergeben können. Erstens darf der Modus nur gewechselt werden, wenn die Queue leer ist, und der Simulator muss diese Bedingung überprüfen. Zweitens können, selbst wenn der Übergang von einem Modus in den anderen korrekt nachvollzogen wird, Symbole von der simulierten Queue gelesen werden, bevor sie geschrieben werden. Das erste dieser beiden Probleme wird gelöst, indem eine spezielle Null-Markierung nach dem letzten eingefügten bzw. gelesenen Symbol vor dem Wechsel des Modus in die erratenen Zeichenketten eingefügt wird. Betrachten wir die Situation, dass der Simulator im links-rechts Modus ist. Wenn das letzte Symbol vor dem Wechsel auf S_R geschrieben worden ist, dann fügt der Simulator die Null-Markierung hinzu und speichert die Information, dass sie gesetzt wurde. Oberhalb dieser Markierung kann der Keller noch simuliert werden, aber weitere Einfügungen in die Zeichenkette unterhalb der Null-Markierung sind nicht mehr erlaubt bis eine solche Markierung auch in S_L eingefügt wird und der Modus zu rechts-links wechselt. In analoger Weise arbeitet der Simulator im rechts-links Modus. Null-Markierungen werden dadurch notwendigerweise in Paaren erzeugt, jeweils eine auf jedem der beiden Keller des Simulators. Daher ist die Länge der Zeichenketten zu den Zeitpunkten das Wechsels zwischen den Modi jeweils gleich (unter der Annahme, dass der Vergleich am Ende der Simulation erfolgreich verläuft) und die simulierte Queue ist an diesen Punkten leer.

Um das zweite Problem zu lösen, ordnen wir jedem gespeicherten Symbol eine Variable mit einem Wert aus der Menge $\{0, 1\}$ zu und lassen den Simulator den aktuellen Wert f speichern. Anders als in dem obigen Korrektheitsargument sind diese Variablen nun nicht nur Hilfkonstruktionen des Beweises, sondern werden tatsächlich gemeinsam mit dem Symbol gespeichert. Mit je-

dem in die simulierte Queue gespeicherten Symbol verbinden wir den Wert $f + 1 \pmod 2$, mit jedem gelöschten Symbol f. Wenn der aktuelle Wert von f für mindesten eine Einfügung verwendet worden ist, dann kann der Simulator nichtdeterministisch den Wert um eins (modulo 2) erhöhen. Nehmen wir, um einen Widerspruch abzuleiten, an, dass die resultierenden Zeichenketten gleich sein könnten, obwohl im Rahmen der Simulation ein Symbol gelöscht worden ist, bevor es geschrieben wurde. Betrachten wir den Zustand direkt vor der ersten Situation, in der dies eintritt. Dann sind die Längen der gespeicherten Zeichenketten gleich, die Werte, die mit den beiden obersten Symbolen verbunden sind, stimmen überein und haben beide den Wert f (da wir annehmen, dass die Ketten schließlich identisch sind). Der Wert von f ist seit der letzten Einfügung erhöht worden, ohne dass ein weiteres Symbol eingefügt wurde. Daher erhält das (fehlerhaft) gelöschte Symbol ebenfalls den aktuellen Wert von f. Dagegen wird das nächste eingefügte Symbol den Wert $f + 1 \pmod 2$ erhalten. Dies führt, entgegen unserer Annahme, zu einer Abweichung der Zeichenketten.

Da jeder Schritt der beiden Phasen der Simulation in konstanter Zeit Zeit ausgeführt werden kann, sehen wir, dass der gesamte Zeitaufwand $O(t(n))$ ist. □

In Kombination mit der offensichtlichen Simulation von zwei Kellern durch eine Deque (die Keller werden an den beiden Enden der Deque simuliert, wobei die Böden markiert sind) erhalten wir:

Korollar 2 *Für nichtdeterministische Maschinen sind zwei Keller und eine Deque bis auf einen linearen Zusatzaufwand äquivalent.*

Wir können nun die Frage aus [7] beantworten, ob nichtdeterministische Maschinen mit mehreren Bändern (oder auch zwei Bändern, die nach [5] ohne Zeitverlust eine beliebige Anzahl simulieren können) und die in linearer Zeit arbeiten stärker sind als solche mit einer Deque.

Korollar 3 *Es existiert keine allgemeine nichtdeterministische Simulation von mehreren Bändern durch eine Deque, die in linearer Zeit arbeitet.*

Beweis. Angenommen, es existiert eine lineare Simulation von zwei Bändern durch eine Deque. Für Maschinen, die in Realzeit arbeiten, erhalten wir eine Schranke $O(n)$ auf zwei Kellern. Werden diese beiden Keller mit der Methode aus [42] auf einem einzelnen Band simuliert, dann erhalten wir eine Schranke $O(n^{1.5}\sqrt{\log n})$. Aber dies widerspricht der fast quadratischen unteren Schranke für die nichtdeterministische Simulation von zwei Bändern durch eines, beispielsweise der Schranke $\Omega(n^2/\log^{(k)} n)$ für jedes positive k aus [15] ($\log^{(k)}$ ist die k-fache Iteration von log). □

Durch Nachrechnen des erforderlichen Mehraufwandes erhalten wir eine Schranke $\Omega(n^{4/3}/\sqrt{\log n})$ für die nichtdeterministische Simulation von zwei Bändern durch eine Deque.

5.3 Deterministische Simulationen

Für nichtdeterministische Berechnungen sind drei Keller so effizient wie jede endliche Anzahl von Bändern oder Kellern [5]. Daher betrachten wir für Simulationen durch drei oder mehr Keller nur deterministische Maschinen.

Zunächst wenden wir uns aber auch hier Simulationen durch zwei Keller zu. Leider haben wir ein weniger vollständiges Bild als im nichtdeterministischen Fall. Die bekannte Simulation einer Queue durch zwei Keller in linearer Zeit (beispielsweise in [26, 43] oder [39, Ex. 2.2.1.14]) lässt sich auf ausgabebeschränkte Deques verallgemeinern, aber ein analoges Resultat für eingabebeschränkte oder allgemeine Deques scheint nicht möglich zu sein.

Im Interesse der Vollständigkeit skizzieren wir kurz die Idee der Simulation einer (am linken Ende) ausgabebeschränkten Deque. Seien die Keller wiederum mit S_L und S_R bezeichnet. Alle Löschungen werden mit Hilfe von S_R durchgeführt, während Einfügungen durch Push-Operationen auf S_L (linkes Ende der simulierten Deque) und S_R (rechtes Ende der simulierten Deque) nachgebildet werden. Wenn Das Löschen eines Symbols von einer nicht-leeren Deque simuliert werden soll, sind zwei Fälle zu betrachten:

Keller S_R ist nicht leer: Das oberste Symbol aus S_R wird gelöscht und zurückgegeben.

Keller S_R ist leer: Der Inhalt von S_L wird komplett gelesen und (in gespiegelter Form, wenn beide Kellerböden in der gleichen Weise ausgerichtet werden) auf S_R geschrieben. Da wir annehmen, dass die simulierte Deque nicht leer ist, kann danach ein Symbol gemäß dem ersten Fall entnommen werden.

Jedes in die simulierte Deque eingebrachte Symbol wird höchstens einmal von S_L nach S_R transferiert, womit die Anzahl der Keller-Operationen pro simulierter Deque-Operation auf 4 begrenzt ist.

Diese lineare Lösung für die Simulation einer ausgabebeschränkten Deque durch zwei Keller kann keinesfalls zu einer Realzeit-Äquivalenz verstärkt werden, wie das folgende Ergebnis zeigt.

Definiere $L \subseteq \{0,1,2,3,\$\}^*$ durch

$$L \;=\; \{w_1\$w_2\$w_3xy\$z \mid$$

$$w_1, w_2, w_3 \in \{0,1\}^*, x \in \{2,3\},$$
$$y = w_2^R \text{ wenn } x = 2, \quad y = w_3^R \text{ wenn } x = 3,$$
$$\exists t \in \{0,1\}^* : w_1 = tz^R\},$$

wobei u^R die Spiegelung von u bedeutet.

Lemma 3 *Es gibt eine deterministische Maschine mit zwei Kellern, die L in Realzeit akzeptiert.*

Beweis. Es ist klar, dass die Maschine Mitgliedschaft in der regulären Sprache $\{0,1\}^*\$\{0,1\}^*\$\{0,1\}^*\{2,3\}\{0,1\}^*\$\{0,1\}^*$ testen kann während die Eingabe verarbeitet wird. Wir können daher annehmen, dass die Zeichenkette gemäß der Definition von L in Blöcke zerlegt werden kann und wir benutzen die Bezeichnungen aus dieser Definition. Die Maschine, die L akzeptieren soll, schreibt Kopien von w_1 auf *beide* Keller und markiert diese Segmente seiner Speicher. Dann kopiert sie w_2 auf Keller 1 und w_3 auf Keller 2. Wenn x erreicht wird, vergleicht sie entsprechend die Zeichenkette auf Keller 1 oder Keller 2 mit dem Segment y der Eingabe, das auf x folgt. Wenn y mit der gespeicherten Zeichenkette übereinstimmt, dann werden die verbleibenden Symbole der Eingabe nach dem letzten \$ mit dem obersten Abschnitt des Kellers, der gerade geleert worden ist, verglichen (vorher wird natürlich die Markierung entfernt). □

Lemma 4 *Es gibt keine deterministische Maschine, die mit nur einer ausgabebeschränkten Deque ausgestattet ist, und die in der Lage ist, L in Realzeit zu akzeptieren.*

Beweis. Für diesen Beweis stellen wir uns die ausgabebeschränkte Deque in individuell identifizierbare Zellen zerlegt vor, auf denen zwei Zeiger für das linke bzw. rechte Ende operieren.

Um einen Widerspruch herzuleiten, nehmen wir an, dass L von einer Maschine M von der Art akzeptiert werden kann, wie sie im Lemma beschrieben wird. Auf Grund der Möglichkeit zur linearen Beschleunigung (auf Kosten der Größe des Speicheralphabets und der Zustandszahl der Maschine) können wir annehmen, dass in jedem Schritt ein Symbol der Eingabe gelesen wird. Sei m die Anzahl der verschiedenen Symbole, die M zur Speicherung von Information in der Deque zur Verfügung steht. Wir legen eine Konstante $c = \max(2, 3\lceil \log_2 m \rceil)$ fest. Dann wird eine inkompressible Zeichenkette w (nichtkonstruktiv) ausgewählt (siehe Kapitel 2), wobei w die Länge $(c^2 + 2c + 1)n$ für ein hinreichend großes n hat, um die Argumentation durchführen zu können. Nun zerlegen wir w in drei Abschnitte $w = w_1 w_2 w_3$

mit $|w_1| = (c^2 + c)n$, $|w_2| = n$ und $|w_3| = cn$. Wir betrachten akzeptierende Berechnungen von M auf Eingaben mit einem Präfix $w_1\$$. Unsere erste Behauptung ist, dass nach dem Lesen dieses Präfix die Länge ℓ von M's Deque den Wert $2(c+1)n$ übersteigt. Wir können w_1 aus dem Inhalt der Deque durch Eingabe der Zeichenkette \$2\$ (der Kodierung von drei leeren Zeichenketten w_2, w_3 und y) in M zurückgewinnen, indem wir dann Symbol für Symbol nach dem längsten Suffix suchen, der M akzeptieren lässt. Da jede Kette, die zur Akzeptierung führt, ein Präfix von w_1^R ist, kann mit diesem Vorgehen w_1^R zurückgewonnen werden. Nehmen wir nun an, dass $\ell \leq 2(c+1)n$. Die gesamte Zeichenkette w kann folgendermaßen beschrieben werden:

- Eine selbstbegrenzende Kodierung des oben angegebenen Algorithmus zur Rekonstruktion von w_1 mit der Länge $O(1)$.
- Eine selbstbegrenzende Beschreibung von M mit der Länge $O(1)$.
- Der Zustand von M nach dem Lesen von $w_1\$$ mit der Länge $O(1)$.
- Eine selbstbegrenzende Kodierung von ℓ in $O(\log n)$ Bits.
- Eine selbstbegrenzende Kodierung von n in $O(\log n)$ Bits.
- Der Inhalt von M's Deque binär kodiert in höchstens $\ell\lceil \log_2 m \rceil \leq \frac{2}{3}(c^2 + c)n$ Bits.
- Die Zeichenketten w_2 und w_3 (hier ist keine Kodierung erforderlich, da n zur Verfügung steht).

Für genügend große n wird dies zu einer Kompression von w führen, was seiner Inkompressibilität widerspricht.

Zu beachten ist, dass die Deque nach dem Lesen von $w_1\$$ länger ist als der verbleibende Suffix jeder anderen unserer Definition entsprechenden, zu akzeptierenden Eingabe mit leerem z. Dies hat zur Folge, dass das erste Symbol auf M's Deque zu diesem Zeitpunkt niemals entfernt werden kann und kein an diesem Ende hinzugefügtes Symbol auf die weitere Rechnung Einfluss nimmt.

Betrachten wir als nächstes den Zustand von M wenn $w_1\$w_2\$w_3$ gelesen worden ist. Wir zählen die Positionen am Ende der Deque, die während des Lesens von w_3 verändert werden. Nehmen wir zunächst an, dass diese Anzahl größer als $n+1$ ist. Wenn die letzten $n+1$ Symbole in der Deque bekannt sind, dann kann w_2 rekonstruiert werden, indem M im Zustand q_1, der nach Lesen von $w_1\$w_2\$w_3$ erreicht wurde, gestartet wird. In diesem Zustand beginnend werden Zeichenketten der Form $2u\$$ mit $u \in \{0,1\}^n$ eingegeben bis eine

eindeutige Kette $2w_2^R\$$, die zur Akzeptierung führt, gefunden worden ist. Das Endsegment der Deque mit der Länge $n+1$ kann erzeugt werden, indem M vom Zustand q_2 aus gestartet wird, den M eingenommen hatte, bevor sie auf das erste Symbol des Segmentes zum letzten Mal zugegriffen hatte. In diesem Zustand beginnend wird der Abschnitt von w_3 eingegeben, der noch nicht verarbeitet worden war.

Wir können nun w folgendermaßen beschreiben:

- Eine selbstbegrenzende Kodierung des oben angegebenen Algorithmus zur Rekonstruktion von w_2 mit der Länge $O(1)$.
- Eine selbstbegrenzende Beschreibung von M mit der Länge $O(1)$.
- Die Zustände q_1 und q_2 von M mit der Länge $O(1)$.
- Eine selbstbegrenzende Kodierung von n in $O(\log n)$ Bits.
- Eine selbstbegrenzende Kodierung der Position innerhalb von w_3, von der aus M gestartet werden muss, um das Endsegment der Deque zu erzeugen, in $O(\log n)$ Bits.
- Die Zeichenketten w_1 und w_3 (wiederum unkodiert).

Für hinreichend großes n erhalten wir eine Kompression im Widerspruch zur Wahl von w.

Betrachten wir nun den anderen Fall, in dem ein Endsegment u von höchstens $n+1$ Zellen der Deque alle Positionen enthält, die während des Einlesens von w_3 verändert worden sind. Das Endsegment der Deque, das möglicherweise gelesen werden kann während Zeichenketten, die w_3 sein könnten, gelesen werden, kann dadurch erzeugt werden, dass M auf $w_1\$w_2\$$ läuft und u in die danach vorliegende Zeichenkette eingefügt wird. Nun kann w_3 aus den letzten $cn+2$ Symbolen der Deque rekonstruiert werden, indem 3 in M eingegeben wird und wiederum nach einer passenden Zeichenkette gesucht wird. Dies erlaubt uns, w durch folgende Information zu beschreiben:

- Eine selbstbegrenzende Kodierung des oben angegebenen Algorithmus zur Rekonstruktion von w_2 mit der Länge $O(1)$.
- Eine selbstbegrenzende Beschreibung von M mit der Länge $O(1)$.
- Der Zustand von M nach Lesen des Symbols 3 mit der Länge $O(1)$.
- Eine selbstbegrenzende Kodierung von n in $O(\log n)$ Bits.
- Eine selbstbegrenzende Kodierung $|u|$ in $O(\log n)$ Bits.

- Eine binäre Kodierung von u in höchstens $|u|\lceil \log_2 m \rceil \leq \frac{1}{3}c(n+1)$ Bits.
- Eine selbstbegrenzende Kodierung der Position von u in der Deque (relativ zum Ende der Deque nachdem $w_1\$w_2\$$ verarbeitet worden ist) in $O(\log n)$ Bits.
- Die Zeichenketten w_1 und w_2 (auch hier ist keine Kodierung erforderlich).

Wieder führt dies zu einer Kompression im Widerspruch zur Auswahl von w. □

Als Konsequenz der beiden eben bewiesenen Lemmata erhalten wir:

Satz 8 *Deterministische Maschinen mit einer ausgabebeschränkten Deque können Maschinen mit zwei Kellern (und somit auch unbeschränkte Deque-Maschinen) nicht in Realzeit simulieren.*

Zwei Keller sind nicht in der Lage, die Sprache

$$J = \{x2x' \mid x \in \{0,1\}^* \text{ und } x' \text{ ist ein Präfix von } x\}$$

in Realzeit zu erkennen, was aus dem wesentlich stärkeren Resultat von Jiang e.a. [36] folgt (dort wurden Bänder betrachtet). Andererseits reicht sogar eine einfache Queue zur Erkennung dieser Sprache in Realzeit aus. Daher sind zwei Keller und eine ausgabebeschränkte Deque für deterministische Berechnungen in Realzeit unvergleichbar.

Diese Aussage gilt allerdings mit dem hier vorliegenden Beweis wirklich nur für Realzeit, denn unsere trennende Sprache L kann von beiden Modellen in linearer Zeit akzeptiert werden und ist somit nicht in der Lage, sie zu trennen.

Jiang e.a. erwähnen, dass J durch drei lineare Bänder, zwei zweidimensionale Bänder oder ein Band mit zwei Köpfen in Realzeit erkannt werden kann. Wir fügen diesen Modellen als weiteres Beispiel das einer deterministischen Maschine mit vier Kellern hinzu. Eine interessante von Amir M. Ben-Amram vorgeschlagene Variante linearer Bänder (vielleicht treffend als Edit-Bänder bezeichnet) erlaubt das Einfügen und Löschen von Bandzellen. Es ist evident, dass zwei Keller einem Edit-Band äquivalent sind, und daher entsprechen vier Keller zwei Edit-Bändern.

Zum Abschluss – wie angekündigt – eine Skizze des Algorithmus, um J durch vier Keller in Realzeit zu erkennen. Dazu betrachten wir eine Situation, in der ein Präfix v der Eingabe w mit $|v| = 2^k$ für $k \geq 2$ gelesen worden ist. Wir zerlegen v in

$$v = a_1 \cdots a_{2^{k-2}} a_{2^{k-2}+1} \cdots a_{2^{k-1}} a_{2^{k-1}+1} \cdots a_{2^k},$$

wobei die a_i einzelne Symbole sind. Ein Keller (hier als „klein“ bezeichnet) speichert

$$a_1 a_{2^{k-1}} a_2 a_{2^{k-1}-1} \cdots a_{2^{k-2}} a_{2^{k-2}+1}$$

(der Kellerboden liegt auf der rechten Seite). Ein „großer Keller“ speichert

$$a_1 a_{2^k} a_2 a_{2^k-1} \cdots a_{2^{k-1}} a_{2^{k-1}+1}.$$

Beginnend von dieser Situation konstruiert der Kellerautomat einen größeren Keller. Nehmen wir zuerst an, dass weitere 2^k Symbole aus $\{0,1\}$ in der Eingabe zur Verfügung stehen. Dann kombiniert der Kellerautomat gelesene Symbole $a_{2^k+1} \cdots a_{2^{k+1}}$ mit den Symbolen des großen Kellers, wobei in zwei Durchgängen (unter Verwendung des dritten Kellers) die Folge

$$a_1 a_{2^{k+1}} a_2 a_{2^{k+1}-1} \cdots a_{2^k} a_{2^k+1}$$

auf den vierten Keller geschrieben wird, ohne den großen Keller zu zerstören. Nun gilt die Invariante, wenn der neu konstruierte Keller den vormals großen ersetzt, der bisherige große den kleinen ersetzt und der kleine Keller aufgegeben wird. Das Aufgeben des kleinen Kellers wird durch Schreiben eines Trennsymbols erreicht, das wie eine Bodensymbol wirkt (es wäre in Realzeit nicht möglich, den Keller zu leeren). Die Rollen der Keller können einfach in der endlichen Kontrolle vertauscht werden.

Nun betrachten wir den Fall, dass innerhalb der nächsten 2^k Symbole eine 2 gelesen wird. Dann können weitere 2^{k-2} Eingabesymbole mit Hilfe des kleinen Kellers verglichen werden (jedes zweite Symbol wird dabei ignoriert). In damit verschränkter Weise werden jeweils *mehrere* Schritte der Übertragung von Information zwischen dem großen Keller und dem im Aufbau befindlichen Keller ausgeführt. Dabei werden die nun fehlenden neuen Eingabesymbole durch Platzhalter ersetzt. Am Ende der Übertragung ist der vormals große Keller erhalten geblieben und kann (nun nur noch als unärer Zähler dienend) zur Bestimmung des Symbols an Position 2^{k-2} auf dem neu konstruierten Keller verwendet werden. Ist der kleine Keller erschöpft, dann stoppt der Übertragungsprozess und der neu konstruierte Keller dient als Quelle, beginnend nach Position 2^{k-2}. Die Geschwindigkeit des Übertragungsprozesses kann so groß gewählt werden, dass sowohl diese Umspeicherung als auch das Auffinden von Position 2^{k-2} innerhalb der Zeit ausgeführt werden kann, in der Symbole aus dem kleinen Keller gelesen werden.

Rosenberg [64] beschreibt die Simulation einer allgemeinen Deque durch drei Keller in linearer Zeit. Die Idee dieser Simulation ist es wie in der oben beschriebenen Simulation einer Queue, einen linken und einen rechten Abschnitt der Deque auf zwei der Keller zu halten. Wenn einer der beiden Abschnitte leer ist und eine Löschung in diesem Teil durchgeführt werden soll,

wird der dritte Keller benutzt um den verbleibenden, nicht-leeren Abschnitt in zwei Teile von annähernd gleicher Größe zu zerlegen. Die Analyse dieser Simulation ist nicht einfach, da Einträge der Deque (anders als im Falle der Simulation einer Queue) unbeschränkt oft von einem Keller auf den anderen und zurück kopiert werden. Um dies an einem einfachen Beispiel zu sehen, nehmen wir an, dass der rechte Abschnitt die Folge $XIXXXXX$ speichert, während der linke Abschnitt leer ist. Wir stellen diese Konfiguration der beiden Keller durch

$$| XIXXXX$$

dar, wobei | die Position der beiden Kellerböden markiert. Nach einer Löschung an der linken Seite wird der Zustand

$$IX | XXX$$

sein und nach vier Einfügungen von Elementen X am linken Ende und drei Löschungen am rechten Ende ist der Zustand dann

$$XXXXIX |,$$

was gerade die Spiegelung des ursprünglichen Zustands ist. Durch eine analoge Folge von Operationen kann dieser ursprüngliche Zustand wieder erreicht werden. Durch Wiederholung der gesamten Folge von Operationen kann das Element I unbeschränkt oft von einem Keller auf den anderen wechseln. Die Gesamtkosten dieses Verfahrens sind trotzdem linear, weil nach dem Aufteilen eines Kellers der Länge s mindestens $\lceil s/2 \rceil$ der „alten“ Elemente gelöscht werden (die aktuelle Operation eingeschlossen) bevor die nächste Aufteilung notwendig wird. Informell zeigt dies, dass ein Element im schlechtesten Fall an im Mittel 2 solchen Restrukturierungen beteiligt ist.

Wir stellen hier eine andere Simulation einer Deque durch drei Keller in linearer Zeit dar, die einfacher zu analysieren ist und einen kleineren linearen Faktor im Bezug auf die Anzahl der Push- und Pop-Operationen erfordert. Während Rosenbergs Lösung höchstens $18n$ Keller-Operationen erfordert, um n Deque-Operationen zu simulieren, wird unsere Schranke $9n$ sein. Es mag von Interesse sein, dass Rosenbergs Simulation eng verwandt mit der von Stoss in [69] angegebenen Konstruktion ist, die Mehrkopf-Turingmaschinen durch solche mit nur einem Kopf pro Band in linearer Zeit simuliert, während unsere durch den ersten Beweis einer solchen Simulation durch Hartmanis und Stearns [22] angeregt wurde.

Satz 9 *Eine Folge von n Deque-Operationen kann deterministisch durch drei Keller unter Verwendung von $9n$ Keller-Operationen simuliert werden.*

Beweis. Wie in Rosenbergs Simulation stellen wir die Deque durch zwei Keller S_L und S_R dar, die den aktuellen Inhalt der Deque repräsentieren. Im Gegensatz zu der oben skizzierten Simulation, in der (außer in Zwischenstadien) die Konkatenation der beiden Keller exakt den Inhalt der Deque reflektiert, wobei die Böden aneinandergefügt werden, kann im Rahmen unserer Simulation ein Teil der gespeicherten Information dupliziert oder ungültig sein. Daher kann es erforderlich sein, einige Elemente am Boden der Keller zu entfernen, um eine Darstellung des aktuellen Inhalts der simulierten Deque zu erhalten. Die Zugehörigkeit eines Elements zu einer Sektion, die möglicherweise ungültig oder dupliziert ist, wird hier der Anschaulichkeit wegen durch Farben beschrieben, wobei Elemente dieser Bereiche als „rot“ bezeichnet werden, während alle anderen „grün“ sind. Die Farben können als einzelnes Bit, das den Elementen hinzugefügt wird, dargestellt werden, oder auch durch Aufteilung der Kellerinhalte mit Hilfe von Markierungssymbolen, denn gleichfarbige Elemente treten immer in Blöcken auf. Anfangs sind S_L und S_R leer, wie auch der dritte Keller S_T der (unter Anderem) als Zähler mit dem anfänglichen Stand 0 dient. Einfügungen werden einfach durch Push-Operationen von grünen Elementen auf S_L oder S_R simuliert. Wir beschreiben im Folgen detailiert die Arbeitsweise der Simulation für eine Löschung am linken Ende der Deque. Die korrespondierende Operation am rechten Ende wird in symmetrischer Art und Weise nachgebildet.

Wenn S_L ein grünes Element als oberstes enthält, dann wird dieses einfach entfernt. Wenn das oberste Element von S_L dagegen rot ist, und der Zähler auf dem dritten Keller nicht den Stand 0 hat, dann wird dieses rote Element benutzt und der Zähler dekrementiert. Anderenfalls wird eine Restrukturierung der gespeicherten Information durchgeführt. Alle verbleibenden Elemente von S_L werden entfernt. Die grünen Elemente an der Spitze von S_R werden dort gelöscht und sowohl auf S_L als auch auf S_T kopiert, wobei ihre Farbe in rot verändert wird. Die verbleibenden roten Elemente auf S_R werden – eines nach dem anderen – ebenfalls nach S_L und S_T übertragen. Die nun auf S_T befindlichen Elemente werden zurück nach S_R gebracht (ohne Änderung der Farbe) und ihre Anzahl wird auf S_L oberhalb der gespeicherten roten Elemente gezählt. Schließlich wird dieser Zählerstand auf S_T übertragen. Dann wird die eigentliche Simulation wieder aufgenommen und die Deque-Operation – wenn möglich – ausgeführt.

Um einzusehen, dass die angegebene Simulation korrekt ist, bemerken wir zunächst, dass der aktuelle Inhalt der simulierten Deque sich aus den grünen Elementen auf S_L (von der Kellerspitze in Richtung Boden gelesen), einem Segment von c roten Elementen auf S_L (wobei c der aktuell auf S_T gespeicherte Zählerwert ist), oder dem damit identischen Spiegelbild eines Segmentes roter Elemente der gleichen Länge auf S_R, und schließlich den

grünen Elementen auf S_R (in Richtung vom Boden zur Spitze) ergibt. Die Farben werden hierbei natürlich ignoriert. Diese Invariante wird durch die Push- und Pop-Operationen grüner Elemente aufrecht erhalten. Eine Pop-Operation eines roten Elements verringert den Zählerstand und aktualisiert daher die Länge des roten Segments, das auf den beiden Kellern betrachtet wird. Die Restrukturierung transformiert eine Folge von grünen Elementen in vollständig überlappende rote Segmente.

Jedes Element erhält bei seiner Aufnahme in die Datenstrukturen die Farbe grün und verursacht eine Push-Operation (1 Keller-Operation). Wenn es die simulierte Deque in grüner Farbe verlässt, dann sind die Kosten für dieses Element 2 Keller-Operationen. Anderenfalls ändert sich seine Farbe im Verlauf der Simulation in Rot. Während dies geschieht, wird es von S_L oder S_R auf die anderen Keller bewegt (3 Keller-Operationen), eine der Kopien wird von S_T entfernt und gezählt (3 Keller-Operationen) und der Zählerstand wird auf S_T gebracht (2 Keller-Operationen). Wenn dies Element nie gelöscht wird, dann verbleiben beide Kopien in den Kellern und die Gesamtkosten der Simulation seiner Einfügung sind – wie behauptet – 9 Keller-Operationen. Im anderen Fall wird das Element und möglicherweise auch seine andere Kopie entfernt (2 Keller-Operationen). Zusätzlich wird der Zähler aktualisiert (1 Keller-Operation). Im letzteren Fall ist die Zahl der erforderlichen Keller-Operationen durch 12 beschränkt, allerdings für die Simulation von *zwei* Deque-Operationen. Dies führt zu einem Aufwand von 6 Keller-Operationen pro simulierter Deque-Operation. □

In der beschriebenen Simulation könnten (unter der „Verschwendung“ von Platz auf dem Keller) rote Elemente in der Restrukturierungsphase auf den Kellern verbleiben, was für den schlechtesten Fall n Operationen einsparen würde. Ein Problem wäre allerdings, dass die lineare Beziehung zwischen der aktuellen Anzahl der Elemente auf der simulierten Deque und dem auf den Kellern erforderlichen Platz verloren ginge.

Eine weitere Möglichkeit zur Einsparung von Keller-Operationen ergibt sich aus einer Veränderung der Verwaltung des Zählers. Mit Hilfe eines komplexeren Steuerungsprogramms könnte der Zählerstand linear komprimiert werden, was zu einer Ersparnis von $(3 - \epsilon)n$ Operationen (für jedes $\epsilon > 0$) führen würde. Lässt sich auf den Kellern ein Binärzähler verwalten, dann kann der Aufwand noch weiter reduziert werden. Diese Modifikationen können auch in Rosenbergs Simulation durchgeführt werden.

Kapitel 6

Rücksprünge in einfach verketteten Listen

6.1 Einleitung

In diesem Kapitel untersuchen wir sogenannte Read-Only Programme in in einer Sprache, die gegenüber realen Programmiersprachen stark eingeschränkt ist. Diese Programme erhalten als Eingabe eine einfach verkettete Liste. Ein Read-Only Programm kann weder diese Eingabe verändern, noch zusätzlichen Speicher belegen. Es kann lediglich eine feste Anzahl von Variablen als Zeiger auf die Eingabeliste verwalten. Solche Programme werden beispielsweise von Neil Jones explizit diskutiert. Er schreibt in [37]:

> ... one has a suspicion that such programs will take extra time due to the complexity of "backing up" to inspect an already-seen input.

Hierbei bezieht sich der Mehraufwand auf den Vergleich mit Programmen, die Zweiweg-Zugriff auf die Eingabe haben. Diese Intuition über die Ineffizienz des Prozesses, einen Rücksprung (engl. "backing up") in einer Liste auszuführen, die nicht verändert werden kann, dürfte sich mit der Erwartung vieler Programmierer decken.

Als ein spezielles Problem, an dem diese Ineffizienz von Read-Only Programmen demonstriert werden kann, schlug Jones das Problem der Suche nach einer gegebenen Zeichenkette in einem Text vor [38]. Nach einer Zeichenkette der Länge m in einem Text der Länge s zu suchen, scheint in schlechtesten Fall $\Omega(ms)$ Schritte zu erfordern, wenn weder Zweiweg-Zugriff auf die Eingabe, noch Tabellen oder anderer Hilfsspeicher zur Verfügung stehen.

Das Hauptresultat, das in diesem Kapitel dargestellt wird, ist eine Methode, den Zusatzaufwand für Schritte zu reduzieren, die einem Zeiger innerhalb der Liste in entgegengesetzter Richtung folgen. Genauer gilt, dass für jedes $\epsilon > 0$ die Zeitkomplexität für einen solchen Schritt in einer einfach verketteten Liste der Länge n auf $O(n^\epsilon)$ reduziert werden kann. Es gibt einige vorbereitende Operationen, die linearen Zeitaufwand erfordern, aber da das bloße einmalige Lesen der gesamten Eingabe lineare Zeit erfordert, wird jedes sinnvolle Programm im schlechtesten Fall nicht in der Ordnung seiner Laufzeit beeinflusst. Wir nehmen daher für den Rest dieses Kapitels an, dass die Zeitkomplexität aller zu simulierenden Programme mindestens linear ist.

Zur Vereinfachung der Darstellung konzentrieren wir uns auf eine einzige Liste, aber natürlich können die Konstruktionen getrennt für jede endliche Anzahl von Listen ausgeführt werden.

Als Konsequenz der Simulation erhalten wir eine subquadratische Lösung des oben beschriebenen Problems der Suche nach einer Zeichenkette der Länge m in einem Text aus s Symbolen durch Read-Only Programme, was die naheliegende Schranke $\Omega(ms)$ widerlegt.

6.2 Die Programme

Die hier untersuchten Programme werden in einer imperativen Programmiersprache ohne Möglichkeit der Rekursion formuliert. Sie sind mit einer endlichen Anzahl von Variablen ausgestattet, die als Zeiger in eine lineare Eingabeliste dienen. Die Werte dieser Variablen können durch Zuweisungen (wobei Werte kopiert werden) oder durch Bewegungen in der Liste verändert werden. Nach einer Operation vom letztgenannten Typ zeigt die veränderte Variable auf das nächste Element der Liste. Die Eingabeliste kann nicht verändert werden und die Programme können keinen weiteren Speicher belegen.

In diesem Grundvorrat an Operationen ist keine direkte Möglichkeit vorhanden, einen Zeiger auf das (relativ zu seinem aktuellen Wert) vorherige Element zeigen zu lassen, außer ihm wird der Wert einer anderen Variable zugewiesen, welche auf das gewünschte Element an der vorherigen Position zeigt.

Einige technische Bemerkungen: Zu Beginn der Berechnung zeigen alle Variablen auf den Anfang der Liste. Eine Variable, die durch Vorwärtsbewegungen den Bereich der Eingabeliste verlassen hat, erhält den speziellen Wert `nil`, der von den Programmen erkannt werden kann. In unserer Diskussion verbinden wir häufig die Begriffe „links“ bzw. „rechts“ mit Bewegungen der Zeiger in Richtung Anfang bzw. Ende der Liste.

6.3 Die obere Schranke

Das Ziel dieses Abschnittes ist es, eine effiziente Simulation von Zweiweg-Zugriff auf eine Liste zu entwickeln, die nur Zugriff über Zeiger erlaubt, die sich in nur eine Richtung (hier „rechts“) bewegen können. Die Simulation macht von der Möglichkeit Gebrauch, Wurzelfunktionen der Eingabelänge (annähernd) zu berechnen, eine Technik, die später in Abschnitt 6.3.3 beschrieben wird, und die Gleichheit von Zeigervariablen zu bestimmen. In Abschnitt 6.3.4 werden wir zeigen, wie Zeiger, die auf das gleiche Element verweisen, erkannt werden können. Wir haben eine solche Fähigkeit nicht in das zugrunde liegende Modell aufgenommen. Falls die Möglichkeit, Zeigerwerte zu vergleichen, schon von vornherein vorhanden ist, ist dieser Abschnitt der Konstruktion natürlich überflüssig. Ein vorbereitender Abschnitt 6.3.2 entwickelt Techniken, die die Verwendung von Zeigervariablen als Zähler erlauben, die erhöht, vermindert und verglichen werden können. Auch dieser Teil der Konstruktion wäre überflüssig, wenn die grundlegende Sprache entsprechende Variablentypen zur Verfügung stellen würde. Die Darstellung aller dieser Techniken ist vom Hauptresultat abgetrennt worden, um das zentrale Ergebnis klarer präsentieren zu können. Ohne diese Trennung wären einige Verbesserungen der Effizienz im Hinblick auf die Anzahl der verwendeten Hilfsvariablen möglich, indem direkt auf „interne“ Variablen zugegriffen wird. Dies hätte aber keinen Einfluss auf die asymptotische Zeitkomplexität der Simulation.

6.3.1 Simulation von Rücksprüngen

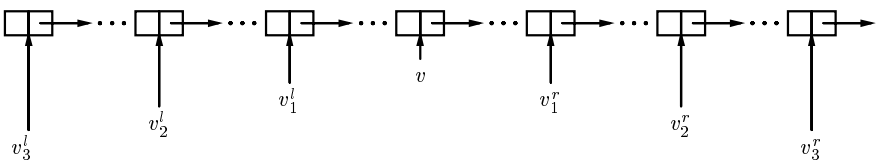
Wir nehmen in diesem Abschnitt an, dass Wurzelfunktionen der Eingabelänge annähernd berechnet werden können (s. Abschnitt 6.3.3) und dass Zeigerwerte verglichen werden können (s. Abschnitt 6.3.4). Sei $k \geq 2$ eine beliebige Konstante und r ein Näherungswert für $\sqrt[k]{n}$.

Sei v eine Zeigervariable des zu simulierenden Programms. Um einen Rücksprung dieser Variable zu simulieren, fügen wir zu v eine Reihe von Hilfsvariablen hinzu. Einige von ihnen werden als Zeiger verwendet, während andere als Zähler dienen. Wir sagen, dass diese Variablen zu v gehören.

Wenn das simulierte Programm den Wert der Variablen v einer anderen Variablen v' zuweist, dann kopiert der Simulator die Werte aller Hilfsvariablen einschließlich der Zähler, die zu v gehören, in die entsprechenden v' zugeordneten Variablen.

Die wichtigsten dieser Hilfsvariablen sind in *Ebenen* $0 \leq j \leq k-1$ angeordnet. Für jede Ebene gibt es sechs Hilfszeiger v_j^u, v_j^s, v_j^l, v_j^c, v_j^r und v_j^t. Die Idee der Simulation ist es, dass v_j^l und v_j^r ein Segment der Liste markieren, in

das v zeigt. Je kleiner j ist, um so näher sind diese Zeiger beieinander, wobei das Segment jeder Ebene vollständig im Segment der nächsten Ebene enthalten ist. Eine typische Situation mit drei Ebenen (wobei Ebene 0 weggelassen worden ist) zeigt die folgende graphische Darstellung.



Zeiger links von v helfen dabei, einen Rücksprung auszuführen, da solche Zeiger vorwärts in Richtung auf v bewegt und auf die Position direkt vor v gebracht werden können. Je näher ein Zeiger an v ist, um so schneller kann diese Operation ausgeführt werden. Auf der anderen Seite ist eine aufwendigere Folge von Operationen nötig, wenn v ein aktuelles Segment verlässt, denn dann müssen die Segmentgrenzen verschoben werden, um die Invariante der Simulation aufrecht zu erhalten. Die Verschachtelung der Segmente hilft, die asymptotischen Kosten dieser Operationen zu begrenzen, wie genauer weiter unten ausgeführt wird.

Für jedes j wird (zu gewissen „Stichzeitpunkten“) der Abstand sowohl zwischen v_j^l und v_j^c, als auch zwischen v_j^c und v_j^r auf r^j gehalten. Hierbei befindet sich v_j^l links und v_j^r rechts von v_j^c. Der Abstand von v_j^l zum Listenanfang wird zu diesen Zeitpunkten ein ganzzahliges Vielfaches von r^j sein.

Zusätzlich zu diesen Eigenschaften werden wir die folgende Invariante aufrecht erhalten: Das Element, auf welches v zeigt, liegt zwischen denen, die durch v_j^l und v_j^r bezeichnet werden (einschließlich dieser beiden Elemente), außer wenn v in einen Suffix von weniger als r^j Symbolen verweist. Wir nehmen an, dass eine Variable h auf den Anfang der Liste zeigt, und dass durch Vergleich dieser Variable mit jedem Zeiger, der sich rückwärts bewegt, sichergestellt ist, dass kein Versuch gemacht wird, den Zeiger von der Liste zu bewegen.

Am Anfang der Berechnung zeigen alle Variablen v_j^l auf den Beginn der Eingabeliste. Für $j = 0, \ldots, k-1$ werden die Variablen v_j^c und v_j^r mit Hilfe von Zählern an die richtigen Positionen gebracht. Diese Initialisierung kann in linearer Zeit durchgeführt werden.

In unserer Erläuterung der Simulation nehmen wir an, dass der zu bewegende Zeiger sich nicht in der Nähe einer der Enden der Liste befindet. Die einfachen Anpassungen, die in diesen Fällen nötig sind, lassen wir weg. Unter dieser Annahme halten wir die zusätzliche Invariante aufrecht, dass die

durch v_j^l und v_j^r markierten Intervalle echt ineinander enthalten sind, d.h., kein Paar von Variablen v_j^l und $v_{j'}^l$ (v_j^r und $v_{j'}^r$) für $j \neq j'$ fällt zusammen.

Wir beschreiben nun die Prozedur, um das Intervall der Ebene j zu verschieben. Genauer initialisiert die Prozedur die Variablen v_j^u und v_j^t, so dass sie für die Verschiebung des Intervalls $[v_j^l, v_j^r]$ zu einem späteren Zeitpunkt verwendet werden können. Dies wird wie folgt durchgeführt. Ein Zähler wird mit dem Wert r^j belegt. Die Variable v_j^s wird auf den Wert von v_{j+1}^l, die Variable v_j^t wird auf v_j^r gesetzt. In einer ersten Phase wird der Zähler wiederholt vermindert, während v_j^s und v_j^t jedes Mal um eine Position nach links bewegt werden. Wenn der Zähler 0 erreicht, stoppt v_j^t (jetzt in einem Abstand von r^j Positionen von v_j^r), während die Variable v_j^u von v_{j+1}^l parallel mit v_j^s bewegt wird, die weiterhin verschoben wird. Nun bewegen sich beide Zeiger jedes Mal um r Positionen. Diese Phase endet, sobald v_j^s mit v_j^l zusammenfällt. Wir bemerken, dass sich zu diesem Zeitpunkt v_j^u auf der r^j-ten Position links von v_j^l befindet. Als Teil unserer Simulation führen wir diese Folge von Operationen quasi-parallel für verschiedene Intervalle aus. Daher nennen wir die Operationsfolge die *Koroutine* der Ebene j. Jedes Mal, wenn sie aufgerufen wird, führt die Koroutine zwei Schritte aus (worin jeweils zwei Zeiger entweder um zwei oder r Positionen nach rechts bewegt werden, abhängig von der Phase). Daher ist die gesamte Anzahl der erforderlichen Aufrufe, um die Ausführung der Koroutine abzuschließen, durch r^j beschränkt.

Wir beschreiben nun die Simulation der Bewegung von v. Wenn v im nächsten Schritt einen Rückwärtssprung ausführen soll, prüft der Simulator, ob v mit der Variablen v_0^l zusammenfällt. Wenn dies nicht der Fall ist, dann stimmt v entweder mit v_0^c überein, worauf v's neuer Wert der von v_0^l ist, oder v stimmt mit v_0^r überein, was bedeutet, dass v den neuen Wert von v_0^c erhalten kann. Anderenfalls gibt es einen oder mehrere Ebenen $j \geq 0$ mit der Eigenschaft, dass das Intervall, welches durch v_j^l und v_j^r markiert wird, nach links bewegt werden muss, um die Invarianten der Simulation aufrecht zu erhalten.

Unter der Annahme, dass die Koroutine der Ebene j schon durchgeführt worden ist, befindet sich v_j^u in einer Entfernung von r^j Positionen von v_j^l und das Intervall kann verschoben werden, indem v_j^r den alten Wert von v_j^c erhält, v_j^c den alten Wert von v_j^l und v_j^l den Wert von v_j^u. Schließlich wird die Koroutine der Ebene j initialisiert.

Wenn v nach rechts bewegt werden soll, dann schiebt der Simulator in analoger Weise zu der obigen Beschreibung Intervalle nach rechts. Nun spielt v_j^t die Rolle von v_j^u, die auf jeden Fall r^j Positionen von v_j^r entfernt ist, wenn eine Verschiebung notwendig wird.

Wir bemerken, dass mindestens r^j Schritte des simulierten Programms ausgeführt werden müssen, bevor es notwendig wird, ein Intervall der Ebene j

(mit Länge $2r^j$) zu verschieben. Dies gibt der Koroutine der Ebene j gerade genug Zeit, ihre Arbeit vor dem nächsten Ereignis dieser Art abzuschließen.

Für ein gegebenes Programm ist die Anzahl der Koroutinen, die gleichzeitig aktiv sein können, durch das Produkt von k und der Anzahl der Variablen, auf die die Konstruktion angewendet wird, beschränkt. Jede Koroutine arbeitet $O(r)$ Schritte für jeden Schritt des simulierten Programms. Daher erhalten wir bei einer geeigneten Wahl von k den folgenden Satz.

Satz 10 *Sei $\epsilon > 0$. Jede Folge von $t(n) \geq n$ Operationen eines Read-Only Programms mit Zeigern, die Zweiweg-Zugriff erlauben, kann durch ein Programm mit Einweg-Zeigern in $O(t(n)n^\epsilon)$ Schritten simuliert werden.*

Tatsächlich kann, nach einer linearen Inititialisierungsphase, die eigentliche Simulation in $O(n^\epsilon)$ Schritten *pro simuliertem Schritt* durchgeführt werden.

6.3.2 Arithmetik mit Zeigern

Zunächst sei an einige bekannte Techniken zur Simulation von Programmen mit Zählern durch solche mit Zeigern auf eine Liste, welche nur gelesen werden kann, erinnert, siehe [37].

Angenommen, die Eingabeliste ist $(a_n a_{n-1} \ldots a_1)$ und $r \leq n$ ist in Form einer Variablen gegeben, die stets auf die Teilliste $(a_r a_{r-1} \ldots a_1)$ zeigt. Dann können durch den Wert r beschränkte Zähler simuliert werden, wobei eine Variable, die auf die Teilliste $(a_i \ldots a_1)$ zeigt, den Wert i kodiert, eine Variable mit dem Wert `nil` kodiert den Wert 0. Wir sagen kürzer, dass eine Variable auf a_i zeigt statt auf die Teilliste $(a_i \ldots a_1)$.

Ein kodierter Zählerwert $i > 0$ kann um 1 vermindert werden, indem der Zeiger von a_i auf a_{i-1} bewegt wird. Dies erfordert lediglich konstanten Aufwand. Indem dieser Vorgang für zwei Variablen parallel wiederholt ausgeführt wird, können kodierte Zählerstände miteinander in $O(r)$ Schritten verglichen werden. Die Simulation des Erhöhens eines Zählerstandes ist etwas komplizierter. Zunächst wird der gespeicherte Wert i durch $r - i$ ersetzt, indem ein Zeiger bei a_r beginnend parallel mit einem weiteren, der auf a_i zeigt, bewegt wird, bis der letztere das Ende der Liste erreicht. Dann wird der gespeicherte Wert um 1 vermindert (unter der Annahme, dass $i < r$), woraus sich $r - i - 1$ ergibt. Schließlich wird $r - i - 1$ in der gleichen Weise durch $i + 1$ ersetzt. Die Zeitkomplexität der gesamten Operationsfolge zur Simulation einer Erhöhung des kodierten Zählers ist $O(r)$.

Es ist leicht zu sehen, dass auch die Multiplikation oder Division eines kodierten Zählerstandes mit einer Konstanten in Zeit $O(r)$ möglich ist.

6.3.3 Näherungsweise Berechnung von Wurzeln

In diesem Abschnitt entwickeln wir eine Technik, die es ermöglicht, in effizienter Art und Weise die k-te Wurzel aus der Eingabelänge $n > 0$ eines Read-Only Programms angenähert zu berechnen, wobei k eine ganze, positive Zahl ist.

Der Logarithmus der Eingabelänge kann näherungsweise dadurch berechnet werden, dass die Länge wiederholt durch 2 geteilt wird und die Anzahl der Iterationen gezählt wird, bis auf diese Weise der Wert 0 erreicht wird. Dies ist in $O(n + \frac{n}{2} + \frac{n}{4} + \cdots) = O(n)$ Schritten möglich.

Das folgende Programmsegment setzt diese Idee um, wobei $\lfloor \log_2 n \rfloor + 1$ in der Variablen `c` abgelegt wird. Hierfür sind $O(n)$ Operationen ausreichend. Für die Berechnungen weiterer Funktionen, die einem ähnlichen Muster folgen, werden wir keinen Programmcode angeben.

```
c := n;
x := n;
while x > 0 do
  { c := c - 1;
    y := x;
    while y > 0 do
      { y := y - 1;
        y := y - 1;   (* no-op, falls y = 0 *)
        x := x - 1
      }
  };
c := n - c
```

Nachdem $y = (k-1)\lfloor \lfloor \log_2 n \rfloor / k \rfloor$ berechnet worden ist, wiederum in linearer Zeit, wird eine Annäherung $r = \lceil n/2^y \rceil$ von $\sqrt[k]{n}$ in $O(n)$ Schritten berechnet, indem die Eingabelänge wiederholt durch 2 geteilt wird.

So erhalten wir:

$$\begin{aligned} \lceil n/2^y \rceil &\geq 2^{\log_2 n - y} \\ &\geq 2^{\log_2 n - (k-1)/k \cdot \lfloor \log_2 n \rfloor} \\ &\geq 2^{(\log_2 n)/k} \\ &= \sqrt[k]{n} \end{aligned}$$

und

$$\begin{aligned} \lceil n/2^y \rceil &\leq 2^{\log_2 n - y} + 1 \\ &\leq 2^{\log_2 n - (k-1)/k \cdot \log_2 n + k} + 1 \\ &= 2^k \sqrt[k]{n} + 1. \end{aligned}$$

Daher gilt $r = \Theta(\sqrt[k]{n})$.

6.3.4 Vergleich von Zeigern

Wir beschreiben nun, wie ein Programm mit der Fähigkeit, Zeigerwerte zu vergleichen, durch ein Programm ohne diese Eigenschaft simuliert werden kann, wobei der zusätzliche Zeitbedarf pro Operation durch $O(\sqrt[k]{n})$ (für jede Wahl von $k \geq 2$) begrenzt ist.

Wir nehmen an, dass es einen ausreichenden Vorrat an Hilfsvariablen gibt, die die Rolle von Zählern spielen, wie in Abschnitt 6.3.2 beschrieben. Außerdem betrachten wir nur n, die nicht zu klein sind.

Die obere Schranke r für $\sqrt[k]{n}$ wird gemäß Abschnitt 6.3.3 berechnet und über die gesamte Simulation in einer Variablen gespeichert. Für jeden Zeiger des simulierten Programms werden k zusätzliche Zähler $v_1, \ldots, v_k$ mit Werten im Bereich 0 bis einschließlich r eingeführt. Diese Zähler kodieren gemeinsam eine Zahl, nämlich den Abstand zwischen der Position des Zeigers und dem *Anfang* der Liste. Sei $|v_i|$ der Wert von Zähler v_i, dann ist die Zahl, die durch die Gruppe der Hilfszähler kodiert wird, der Wert $\sum_{i=1}^{k} |v_i| \cdot (r+1)^{i-1}$. Da $r \cdot (r+1)^{k-1} > r^k \geq n$, können alle möglichen Abstände auf diese Weise dargestellt werden. Zu Beginn zeigt die simulierte Variable auf den Anfang der Liste und die Zähler werden zurückgesetzt um den Abstand 0 zu repräsentieren. Wenn die simulierte Variable sich nach rechts bewegt, wird der kodierte Abstand erhöht. Diese Erhöhung wird Stufe für Stufe unter Berücksichtigung etwaiger Überträge beginnend mit v_1 durchgeführt.

Die Zeitkomplexität der Erhöhung eines einzelnen v_i ist $O(\sqrt[k]{n})$, weil die Zahl die kodiert wird durch r beschränkt ist (siehe Abschnitt 6.3.2). Die Erhöhung des Abstandes um 1 erfordert höchstens k solche Operationen, was zu einer Zeitkomplexität von $O(\sqrt[k]{n})$ für jede Bewegung eines Zeigers führt.

Für eine Zuweisung erhält ein Zeiger Kopien der Zähler, die zu dem anderen Zeiger gehören. Dies kann in einer konstanten Anzahl von Schritten durchgeführt werden.

Seien zwei Zeiger mit ihren Gruppen von Zählern $v_1, \ldots, v_k$ und $v'_1, \ldots, v'_k$ gegeben. Ihre Werte können in $O(\sqrt[k]{n})$ Schritten verglichen werden, indem jedes v_i mit dem korrespondierenden v'_i verglichen wird (siehe Abschnitt 6.3.2), wobei jeweils Hilfsvariablen verwendet werden, in die ihre aktuellen Werte kopiert werden. Dadurch werden die ursprünglich gespeicherten Zählerstände nicht zerstört.

Die obige Diskussion wird in dem folgenden Satz zusammengefasst, wobei wir $k = 1/\epsilon$ setzen.

Satz 11 *Sei $\epsilon > 0$. Jede elementare Operation eines Programmes mit Zeigern, deren Werte miteinander verglichen werden können, kann durch ein Programm ohne diese Möglichkeit mittels eines Zeitaufwandes von $O(n^\epsilon)$ pro elementarer Operation simuliert werden.*

6.4 Suche nach Zeichenketten

Das Problem, zu entscheiden, ob eine gegebene Zeichenkette p in einem Text t über dem festen Alphabet $\{0,1\}$ vorkommt, kann in der folgenden Weise als Problem der Erkennung einer Sprache formalisiert werden:

$$M = \{p\#t \mid p,t \in \{0,1\}^*, \quad t = t_1 p t_2\}$$

Für Anwendungen wäre es natürlich erforderlich, mehr Information als nur die bloße Anwesenheit der Zeichenkette p zu erkennen. Beispielsweise könnte bei positiver Antwort die Lage der ersten im Text vorhandenen Zeichenkette p zurückgeben werden, oder die Menge aller Positionen von Kopien der Zeichenkette. Da eine untere Schranke für die Erkennung von M sich auf diese komplexeren Fragestellungen überträgt, und andererseits Lösungen des Erkennungsproblems im Normalfall konstruktiv sind und mindestens eine Position von p liefern, scheint die Untersuchung von M dennoch gerechtfertigt.

Unter der Annahme, dass die Eingabe in dem Sinne wohlgeformt ist, dass nur ein Symbol # darin enthalten ist, bezeichnen wir die Länge der Zeichenkette mit $m = |p|$, die Länge des Textes mit $s = |t|$, und die Länge der Eingabe mit $n = m + s + 1$. Zu beachten ist, dass die Eingabe – trotz dieser Unterscheidung – eine Zeichenkette ist, die dem Programm in unserem Modell als Liste von Symbolen vorliegt.

Die vielleicht am nächsten liegende Methode, um p in t aufzufinden, versucht p mit aufeinander folgenden Teilen von t abzugleichen. Diese Methode hat eine quadratische Laufzeit, wie aus dem Beispiel $p = 0^{m-1}1$ und $t = 0^s$ hervorgeht. Hierbei wird eine Vergleichsrichtung von links nach rechts angenommen. Andere deterministische Strategien zur Auswahl der zu vergleichenden Zeichen könnten durch entsprechende Anordnung der 1 in p auf analoge Weise zu einer quadratischen Laufzeit gezwungen werden. Diese Methode mit Laufzeit $\Theta(ms) = \Theta(n^2)$ lässt sich durch Read-Only Programme implementieren.

Auf der anderen Seite existieren wohlbekannte Algorithmen zur Suche von Zeichenketten in Texten, die nur lineare Laufzeit benötigen, siehe z.B. [40].

Die bekanntesten Ansätze zur linearen Suche nach Zeichenketten benutzen allerdings Tabellen mit einer Größe, die proportional zur Länge m der

Zeichenkette ist. Somit ist es nicht möglich, diese Verfahren durch Programme zu realisieren, die keinen zusätzlichen Speicher verwenden. Eine Ausnahme bildet der Algorithmus von Galil und Seiferas [17], der bei linearer Laufzeit mit einer konstanten Anzahl von Zeigern in den Eingabetext auskommt. Diese Zeiger können aber – im Gegensatz zu den in unserem Modell vorhandenen – sowohl vorwärts als auch in Richtung des Textanfangs bewegt werden.

Mit Hilfe der effizienten Technik, um Rückwärtssprünge in einer einfach verketteten Liste auszuführen, erhalten wir aus [17]:

Korollar 4 *Für jedes $\epsilon > 0$ kann das Problem der Suche nach einer Zeichenkette in einem Text durch ein Read-Only Programm in $O(n^{1+\epsilon})$ Schritten gelöst werden.*

6.5 Eine untere Schranke

Die obere Schranke $O(t(n)n^{\epsilon})$, die wir für die Simulation von Zweiweg-Zugriff zeigen konnten, legt einen linearen Zusammenhang zwischen der Anzahl der zusätzlich erforderlichen Zeiger und $1/\epsilon$ nahe. Die folgende untere Schranke zeigt tatsächlich eine solche Beziehung.

Einige Bemerkungen im Bezug auf die Art der Simulation, die hier betrachtet wird, sind erforderlich. Wie im Falle der oberen Schranke erlauben wir dem Simulator eine Vorbereitungsphase, in der der Simulator seine Zeiger in für ihn optimaler Weise platzieren darf. Da das Interesse auf eine schnelle Simulation gerichtet ist, wäre es nicht sinnvolle, eine unangemessen lange Vorbereitungsphase zu erlauben. Tatsächlich ist die untere Schranke aber unabhängig von der Länge dieser Vorbereitung und nutzt für den Beweis der Schranke für die Anzahl an erforderlichen Hilfsvariablen nur die Annahme, dass die folgende Phase, in der eine Ausgabe erzeugt wird, schnell ausgeführt wird.

Wir nennen für den Zweck dieses Beweises eine Simulation $b(n)$ *zeitbeschränkt*, falls das Folgende gilt. Für ein Programm, das m Ausgabesymbole innerhalb von $t(n)$ Schritten erzeugt, kann durch den Simulator innerhalb $b(n) \cdot t(n)$ Schritten die gleiche Folge von Ausgabesymbolen erzeugt werden, wobei die Zeit beginnend mit dem ersten Symbol der Ausgabe gemessen wird. Hierbei ist zu beachten, dass nicht gefordert wird, die Schranke $b(n)$ für jedes einzelne Symbol einzuhalten, was eine stärkere Einschränkung der Simulationverfahren bedeuten würde, sondern lediglich, dass die Zeitschranke in einem amortisierten oder gemittelten Sinne eingehalten wird. Der Simulator kann beispielsweise in einem frühen Stadium der Simulation Schritte einsparen, die er dann später einsetzen darf.

Wie nehmen weiterhin an, dass sowohl die Elemente, die in der Eingabeliste gespeichert werden, als auch die Ausgabe über einem festen Alphabet, konkret $\{0, 1\}$, gebildet werden. In diesem Sinne ist unsere untere Schranke stärker als eine Schranke, die einen unendlichen Vorrat an Symbolen voraussetzt, wie z.B. in [59]. Dieser qualitative Unterschied wird daran deutlich, dass ein Programm über einem endlichen Alphabet ein Teilstück der Eingabe in seinem „inneren" Zustand speichern kann, was bei einem unendlichen Eingabevorrat unmöglich ist. Wir erlauben dem Simulator ebenfalls, sowohl Zeigervariablen als auch Zähler zu verwenden, die nur im Bezug auf ihre Kapazität beschränkt sind. Wir nehmen an, dass die Werte der verwendeten Zähler im Bereich 0 bis n liegen. Tatsächlich könnten wir auch eine polynomielle obere Schranke zulassen. Wir erlauben jede Art von Operationen auf den Werten der Zähler. Dies erhöht wiederum die Mächtigkeit des Modells, das für die untere Schranke untersucht wird, im Vergleich zu den im Beweis der oberen Schranke verwendeten Fähigkeiten.

Lemma 5 *Das Read-Only Programm C sei mit k Variablen (Zeigern oder Zählern) ausgestattet und arbeite auf der Eingabe w. Sei s eine passende Konstante, die nur von C und dem Kompressionsverfahren abhängt. Weiter sei x eine Teilkette von w mit Länge $|x| = p$ und $p > s + 2\log s + 1 + (k + 2)(2\log|w| + 1)$. Falls C das Spiegelbild von x ausgibt und währenddessen kein Zeiger auf den Bereich von x zeigt, dann kann w komprimiert werden.*

Beweis. Die folgende Information wird aneinandergehängt, um eine komprimierte Kodierung von w zu erhalten:

- eine selbst-begrenzende Kodierung des Programmtextes von C (s Symbole),
- die Instruktionsnummer der Operation von C, die zu dem Zeitpunkt ausgeführt wird, wenn C mit der Ausgabe des Spiegelbildes von x beginnt ($\leq 2\log s + 1$ Symbole),
- der Wert der Variablen von C zu diesem Zeitpunkt, wobei Zeiger durch die Position in der Eingabeliste identifiziert werden ($\leq 2k\log|w| + 1$ Symbole),
- die Position von x in w ($\leq 2\log|w| + 1$ Symbole),
- die Länge von x ($\leq 2\log|w| + 1$ Symbole), wobei all diese numerischen Daten in selbst-begrenzender Binärkodierung gespeichert werden,
- w ohne die Teilkette x ($|w| - p$ Symbole).

Der folgende Algorithmus gewinnt w aus den obigen Daten zurück und wird ihnen (in $O(1)$ Bits kodiert) vorangestellt:

- Lege eine Liste u der Länge $|w|$ an und speichere die $|w|-p$ Symbole von w, die aus der obigen Kodierung bekannt sind, in ihnen. Die übrigen x Positionen werden vorläufig beliebig belegt.
- Setze k Variablen auf ihre gespeicherten Werte bzw. Positionen in u.
- Simuliere C beginnend von der gespeicherten Instruktionsnummer und halte die folgenden $|x|$ Ausgabesymbole fest.
- Füge C's gespiegelte Ausgabe in u an der korrekten Position ein.

Wir halten fest, dass die Simulation von C auf u dieselbe Ausgabe wie bei der Ausführung auf w erzeugt, da innerhalb dieser Phase der Berechnung von C kein Zeiger auf den Bereich zugreift, der in w von x eingenommen wird und in dem w und u voneinander abweichen.

Die zusätzliche Information, um x wiederherzustellen, hat eine Länge die gemäß den Voraussetzungen des Lemmas kürzer ist als die von x. Daher ist w komprimierbar. □

Satz 12 *Jede allgemeine Simulation von Zweiweg-Zugriff durch ein Read-Only Programm, die $O(n^{1/k})$ zeitbeschränkt ist, erfordert mindestens $k-1$ Variablen.*

Beweis. Wir beweisen den Satz, indem wir ein spezielles Programm mit Zweiweg-Zugriff beschreiben, so dass jedes Programm mit Einweg-Zugriff, das die gleiche Aufgabe innerhalb der Zeitschranke $O(n^{1/k})$ löst, die Behauptung erfüllen muss. Das Programm erhält eine lineare Liste w von $n \geq 1$ Symbolen $w = (a_1 a_2 \cdots a_n)$, wobei $a_i \in \{0,1\}$, und gibt die gespiegelte Liste $w^R = (a_n a_{n-1} \cdots a_1)$, aus. Dies erfordert (unter Verwendung von Zweiweg-Zugriff) lediglich lineare Zeit, tatsächlich kann die Ausgabe in konstanter Zeit pro Ausgabesymbol erzeugt werden, sobald die Ausgabephase startet. Wir betrachten nun ein Read-Only Programm C, das eine Liste mit Hilfe von Ein-Weg Zeigern spiegelt und $O(n^{1/k})$ zeitbeschränkt ist. Nach unserer Definition heißt dies: Für eine Konstante c gilt, dass C höchstens $cn^{1/k} \cdot m$ Zeit benötigt, um die ersten m Ausgabesymbole zu erzeugen, wobei wir mit dem Zählen der Schritte bei der ersten Ausgabe von C beginnen. Den Teil der Berechnung, der mit der ersten Ausgabeoperation beginnt, nennen wir *Spiegelung*.

Um das Resultat zu beweisen, wählen wir als Eingabe eine inkompressible Folge von Symbolen mit der Länge $n = p^k$, wobei $p > (2c+2)^{k-2}$ die Bedingungen von Lemma 5 erfüllt.

Wir zeigen durch vollständige Induktion, dass auf gewisse Regionen der Eingabe zu Beginn der Spiegelung mindestens ein Zeiger verweisen muss. Genauer befinden sich mindestens i Zeiger in einem Suffix der Eingabe mit der Länge $(2c+2)^i p^{i+1}$ wenn die Spiegelung beginnt.

Für $i = 0$ gilt diese Aussage trivialerweise. Für $i \geq 1$ konzentrieren wir uns auf w's Teilfolge x der Länge p, die vor dem bereits betrachteten Suffix der Länge $L_i = (2c+2)^{i-1}p^i$ liegt (und nach Annahme mindestens $i-1$ Zeiger enthält). Da dieser Abschnitt der Eingabe die Voraussetzungen des Lemmas 5 erfüllt, kann seine Ausgabe nicht erfolgen, ohne dass eine Zeigervariable auf ihn zeigt. Um den auf x folgenden Suffix zu spiegeln darf C höchstens $cn^{1/k} \cdot L_i = cp(2c+2)^{i-1}p^i$ Schritte verwenden, und innerhalb von höchstens $cp(2c+2)^{i-1}p^i + cp^2$ Schritten muss die Ausgabe von x^R abgeschlossen sein. Da die Zeiger, die sich rechts von x befinden, nicht direkt nach links bewegt werden können, muss ein Zeiger, der sich zunächst links von x befunden hat, die Basis für die Adressierung von x bilden. Daher kann dieser Zeiger nicht weiter als durch die obige Zeitschranke angegeben wird von x entfernt sein. Also gibt es mindestens i Zeiger, die auf einen Suffix der Länge

$$\begin{aligned} & cp(2c+2)^{i-1}p^i + cp^2 + p + (2c+2)^{i-1}p^i \\ & \qquad \leq \quad (c+1)(2c+2)^{i-1}p^{i+1} + cp^2 + p \\ & \qquad \leq \quad (2c+1)(2c+2)^{i-1}p^{i+1} + p \\ & \qquad \leq \quad (2c+2)^i p^{i+1} \end{aligned}$$

verweisen.

Nach Wahl von p haben wir $(2c+2)^{k-2}p^{k-1} \leq p^k - p = n - p$. Das obige Argument zeigt, dass mindestens $k-2$ Variablen auf einem Suffix der Länge $n-p$ erforderlich sind, ein weiterer Zeiger ist nötig, um Zugriff auf die ersten p Symbole zu ermöglichen. Dies schließt den Beweis ab. □

6.6 Eine untere Schranke für stärkere Datentypen

Berechnungsmodelle, die zur Beschreibung und Analyse praktisch relevanter Algorithmen dienen, erlauben im Allgemeinen flexiblere Datentypen als nur Zeichen, Zeigervariablen und beschränkte Zähler. Typische Beispiele sind ganze Zahlen einer bestimmten „Wortlänge“, die größer als $\Theta(\log n)$ sein

kann, unbeschränkte ganze Zahlen oder reelle Zahlen. Es wird auch oft angenommen, dass ein Programm eine Gruppe von primitiven Operationen, so wie z.B. arithmetische Operationen, in Einheitskostenmaß ausführen kann. Dies bedeutet, dass – unabhängig von der Größe der Operanden – jede Operation nur einen Schritt erfordert.

Algorithmen auf Datenstrukturen werden normalerweise in einer von den tatsächlich in der Struktur gespeicherten Daten unabhängigen Weise beschrieben. Z.B. kann eine verkettete Liste in jeder Zelle ein Datum eines beliebigen Typs enthalten. Wenn man aber annimmt, dass ein Programm die Möglichkeit hat, stärkere Datentypen zu verwenden, dann müssen Beweise für untere Schranken dies berücksichtigen. Offensichtlich kann ein komplexer Datentyp mit Operationen, die jeweils nur einen Schritt erfordern, den Beweis einer unteren Schranke, die für ein schwächeres Modell gilt, unmöglich machen. In diesem Abschnitt zeigen wir, dass unsere untere Schranke für Programme gilt, die stärkere Datentypen verarbeiten können, sofern diese Daten die Eigenschaft einer *allgemeinen Inkompressibilität* erfüllen. Diese Eigenschaft verallgemeinert die Inkompressibilität von Zeichenketten. Sie wurde in [3] mit der Absicht eingeführt, Beweise für untere Schranken, die auf der Inkompressibilität von Zeichenketten beruhen, auf andere Datentypen zu verallgemeinern. Wir geben einen Überblick über die Ergebnisse aus [3] und zeigen dann, wie sie verwendet werden können, um die untere Schranke zu verallgemeinern.

6.6.1 Allgemeine Inkompressibilität

Ein Datentyp wird in [3] als ein Tripel $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$ definiert, wobei $\mathcal{D}$ der *Grundbereich* ist, $\mathcal{F}$ die Menge der *primitiven Operationen* und $<$ die Ordnungsrelation für Vergleichsoperationen. Ein Berechnungsprozess, der auf einem n-Tupel von Werten aus $\mathcal{D}$ beruht, kann als $\mathcal{F}$ *Entscheidungsbaum* betrachtet werden. In solch einem Baum steht jeder verzweigende Knoten für einen Vergleich von Werten, die ausgehend von der Eingabe durch eine endliche Kombination von Operationen aus $\mathcal{F}$ berechnet werden. Blätter geben die Ausgabe der Berechnung an. Es gibt keine Beschränkung der Operationen, die zur Berechnung von Resultaten verwendet werden können, oder der Vergleiche, solange die Berechnung endlich ist.

Wir halten fest, dass solche Bäume eine (nicht-uniforme) Abstraktion von Programmen bilden, die auf Eingaben aus $\mathcal{D}$ operieren, und dass sie *genuin zeitbeschränkt* sind, ein Ausdruck, der von Meyer auf der Heide in [48] eingeführt wurde, um anzudeuten, dass das Programm eine Zeitschranke besitzt, die nur von der Anzahl der Eingaben abhängt, aber nicht von deren Wert. Da wir nur daran interessiert sind, eine Zeitschranke zu besitzen, aber nicht

an ihrem genauen Wert, sind die Details des Berechnungsmodells (oder der Programmiersprache) von geringer Bedeutung. Eine informelle Beschreibung des Algorithmus, die klar zeitbeschränkt ist, genügt.

Ein Datentyp $\mathcal{T}$ ist *inkompressibel*, wenn für kein $n > 0$ ein Paar f_n, g_n von Funktionen existiert, die durch $\mathcal{F}$-Entscheidungsbäume berechnet werden, so dass f_n den Bereich $\mathcal{D}^n$ in $\mathcal{D}^{n-1}$ abbildet und g_n die inverse Funktion von f_n ist.

Diese Eigenschaft wurde in [3] für die folgenden Datentypen bewiesen:

1. Binärwörter mit einer festen Länge und einer beliebigen Menge von Operationen.

2. Reelle Zahlen, mit einer beliebigen Menge von Operationen, die stückweise stetige Funktionen berechnen. Beispiele sind algebraische Operationen, die Floor-Funktion und viele andere bekannte Funktionen.

3. Unbeschränkte ganze Zahlen mit den Operationen Addition, Subtraktion, Multiplikation, bitweisen Boole'schen Operationen und beschränkten Schiebeoperationen.

Sei P ein Programm, das auf der Eingabe x operiert, eine Folge von Eingabewerten aus $\mathcal{D}$. Wir bezeichnen die Ausgabe von P bei Eingabe x mit $[\![\mathrm{P}]\!]x$ (die Ausgabe ist ebenfalls eine Folge).

6.6.2 Erweiterung der unteren Schranke

Wir erweitern nun die Klasse der Read-Only Programme, indem wir den Programmen erlauben, eine endliche Anzahl von Variablen eines gegebenen Datentyps $\mathcal{T}$ zu benutzen, auf das es die primitiven Operationen des Datentyps mit Einheitskosten anwenden kann. Das Problem, mit dessen Hilfe die untere Schranke bewiesen wird, ist die Spiegelung einer Liste von Elementen aus $\mathcal{T}$. Die Idee ist es, die Argumente aus dem Beweis von Satz 12 zu benutzten, wobei das Konzept einer *inkompressiblen Zeichenkette* durch eine *inkompressible endliche Folge* von Elementen aus $\mathcal{T}$ ersetzt wird, was im Folgenden definiert wird.

Definition 1 *Seien K, D zwei genuin zeitbeschränkte Programme (informell: Kodierer und Dekodierer), welche den Datentyp $\mathcal{T}$ benutzen. Sei $x \in \mathcal{D}^n$. Wir sagen, dass x durch (K, D) komprimiert wird, falls es eine Folge von $y \in \mathcal{D}^m$ mit $m < n$ gibt, so dass $[\![K]\!]x = y$ und $[\![D]\!]y = x$.*

Wenn x nicht durch (K, D) komprimiert wird, nennen wir es inkompressibel *für dieses Paar von Programmen.*

Zu beachten ist, dass Kodierer und Dekodierer spezifiziert werden müssen, um behaupten zu können, dass ein x inkompressibel ist. Die folgende Aussage ist eine direkte Konsequenz der Definition:

Beobachtung 2 *Wenn $\mathcal{T}$ ein inkompressibler Datentyp ist, dann gilt für jedes Paar von zeitbeschränkten Programmen und für jedes $n > 0$, dass es Folgen der Länge n gibt, die inkompressibel für die gegebenen Programme sind.*

Um die erweiterte Version von Satz 12 zu zeigen, verallgemeinern wir zunächst Lemma 5. Die Verallgemeinerung besteht darin, dem Programm C zu erlauben, auf Variablen des Typs $\mathcal{T}$ mit den dazugehörigen Operationen zu arbeiten und als Eingabe eine Folge von Werten aus dem Bereich $\mathcal{T}$ zu lesen. Wir nehmen ebenfalls an, dass C zeitbeschränkt ist.

Um das Lemma zu beweisen, müssen wir zeigen, dass ein Paar von Programmen (K, D) existiert, die in der Lage sind, eine Eingabe w zu komprimieren. Die einzige Abweichung von der ursprünglichen Behauptung ist, dass nun Kodierer und Dekodierer als zeitbeschränkte Programme gegeben werden müssen. Daher erläutern wir lediglich die Unterschiede, ansonsten ist der Beweis identisch.

Den Dekodieralgorithmus haben wir schon im Rahmen des Beweises im letzten Abschnitt beschrieben. Wir müssen lediglich sicherstellen, dass er zeitbeschränkt ist. Dies folgt aus einer Zeitbeschränkung des Programms C.

Der Kodierungsalgorithmus arbeitet wie folgt. Seine Eingabe ist die Liste w. Das Programm berechnet $p_0 = s + 2\log s + 2 + (k+2)(2\log|w| + 1)$. Es markiert einen Abschnitt x von w mit Länge p_0 (anfangs die ersten p_0 Positionen von w) und führt C aus, während es die Ausgabe des Programms und die Positionen seiner Zeiger überwacht und nach einer Situation wie im Lemma beschrieben sucht (Ausgabe der Spiegelung von x, ohne dass Zeigerwerte in seinem Bereich liegen). Wenn so eine Situation gefunden wird, dann erzeugt der Kodierer die im letzten Abschnitt beschriebene Ausgabe. Falls nicht, schiebt er den als x bezeichneten Abschnitt um eine Position vorwärts und wiederholt den Versuch. Unter der Voraussetzung des Lemmas wird die gesuchte Situation schließlich gefunden.

Der Kodierer kann somit erfolgreich eine Kodierung ausgeben, deren Länge im letzten Abschnitt berechnet wurde (es muss nur überall „Symbol“ durch „Datenelement“ ersetzt werden). Dort wurde auch gezeigt, dass die Länge unter $|w|$ liegt. Schließlich halten wir fest, dass der Kodierer zeitbeschränkt ist, da dies auf C zutrifft.

Wir können die Beschreibung des Kodierers erweitern, indem er eine beliebige Ausgabe erzeugt, falls die im Lemma beschriebene Situation nicht ein-

tritt. Das macht die Beschreibung des Programms vollständig, unabhängig von den Bedingungen des Lemmas.

Beweis der unteren Schranke. Angenommen, wir wissen dass ein Programm C, wie oben beschrieben, existiert, das unter Verwendung von Einweg-Zeigern eine Liste spiegelt und höchstens $cn^{1/k} \cdot m$ Schritte benötigt, um m Ausgaben zu erzeugen (wiederum beginnt die Zählung der Schritte mit der Ausgabe des ersten Symbols durch C). Für die Phase, die der ersten Ausgabeoperation vorausgeht, nehmen wir nur an, dass sie genuin zeitbeschränkt ist. Diese Annahmen reichen aus, um Programme für Kodierer und Dekodierer wie oben beschrieben zu erhalten. Gemäß Beobachtung 2 ist es möglich, eine Eingabe w zu wählen, die für die gegebenen Programme inkompressibel ist. Der Rest des Beweises gilt unverändert.

Kapitel 7

Äquivalenz von Köpfen und Zählern für deterministische endliche Automaten über beschränkten Sprachen

7.1 Einleitung

In diesem Kapitel zeigen wir, dass beschränkte Zähler und Köpfe (d.h., Zeiger in eine nicht veränderbare Eingabezeichenkette) für deterministische endliche Automaten äquivalent sind, falls die betrachtete Sprachklasse auf die *beschränkten* Sprachen begrenzt wird. Hier wird Äquivalenz im Bezug auf die Klasse der akzeptierten Sprachen verstanden. Anders als im Kapitel 6 interessieren wir uns daher hier nicht für die Zeitkomplexität der Simulation. Die Ergebnisse der beiden Kapitel sind gewissermaßen unvergleichbar. Während hier bei fester Zahl der Variablen die Beschreibungsmächtigkeit der Modelle verglichen wird, galt dort die Aufmerksamkeit der Zeitkomplexität, wobei eine Erhöhung der Anzahl an Variablen zur Effizienzsteigerung in Kauf genommen wurde. Dies rechtfertigt auch die unterschiedliche Nomenklatur in der Darstellung. Während dort von Programmen die Rede war, werden wir hier die traditionelle Sprechweise aus der Theorie formaler Sprachen benutzen, welche Programme als endliche Automaten sieht, die andere Komponenten kontrollieren. Dies erleichtert es, die hier gewonnenen Einsichten mit Ergebnissen und offenen Fragen der Literatur zu verbinden. Es sollte nie in Vergessenheit geraten, dass es sich um Prinzip äquivalente Begriffe handelt.

Das in diesem Kapitel erzielte Simulationsergebnis, das die Äquivalenz

von deterministischen endlichen Automaten mit $k \geq 2$ Köpfen und solchen mit nur einem Eingabekopf und $k-1$ beschränkten Zählern über beschränkten Sprachen zeigt, ist in einem gewissen Sinne bestmöglich, denn es ist bekannt, dass deterministische endliche Automaten mit zwei Köpfen eine *unbeschränkte* Sprache entscheiden können, die von keinem deterministischen Automaten mit einem Zähler akzeptiert wird [11]. Die Verallgemeinerung von beschränkten auf unbeschränkte Sprachen ist somit nicht möglich.

Mit Hilfe eines zweidimensionalen Automaten wurde als Spezialfall die Äquivalenz von deterministischen Automaten mit einem Zähler und solchen mit zwei Köpfen über einem einelementigen Eingabealphabet von Morita e.a. in [52] gezeigt, siehe auch [53].

In unserer Darstellung werden wir eine Technik entwickeln, die die Verallgemeinerung der Äquivalenzresultate für eine Reihe von Automatentypen von strikt beschränkten auf beschränkte Sprachen erlaubt. Als weitere Anwendung können wir das Resultat von Ibarra et al. [30] von strikt beschränkten auf beschränkte Sprachen erweitern. Dieses Ergebnis zeigt die Äquivalenz zwischen deterministischen Automaten mit einem umkehrbeschränkten Keller und solchen mit einem Zähler.

Eine wesentliche Motivation dieser Untersuchung ist die Frage, welchen Einfluss die Fähigkeiten eines Berechenbarkeitsmodells auf seine Mächtigkeit haben. Es gibt aber auch andere, eher technische Aspekte. Ein Spezialfall beschränkter Sprachen sind Sprachen über einem einelementigen Alphabet. Unsere Simulation macht eine getrennte Untersuchung der Hierarchie von deterministischen Zählerautomaten mit einer steigenden Anzahl von Zählern über einem einelementigen Alphabet überflüssig. Der Hierarchiesatz für Zählerautomaten in Theorem 3 von [50] folgt aus dem entsprechenden Resultat für Mehrkopfautomaten in Theorem 1 von [50]. Mehrkopfautomaten scheinen ein robusteres Modell als Zählerautomaten gegenüber leichten Variationen der erlaubten Operationen zu sein. Beispielweise gibt es mehrere natürlich erscheinende Definitionen eines Zählers, der durch die Eingabelänge beschränkt wird [63]. Es könnte zunächst einfach gefordert werden, dass ein Zähler im Verlauf einer Rechnung niemals überläuft. Dies hat den Nachteil, dass diese Eigenschaft unentscheidbar ist. Alternativ könnte die Rechnung des Automaten im Fall des Überlaufs blockieren, wobei dann Akzeptierung vom aktuellen Zustand abhängen könnte, eine Fehlermeldung könnte signalisiert werden, wobei der betroffene Zähler undefiniert oder einfach unverändert bleiben könnte. Im Wesentlichen entspricht die letzte Variante, die einen aktiven Test der Bedingung „Zähler auf Maximalwert“ erlaubt, der Definition eines Automaten mit einfachen Köpfen (simple multi-head automaton), siehe z.B. [11]. All diese möglicherweise inäquivalenten Varianten können leicht durch einen Automaten, der generell mit Köpfen ausgestattet ist, simuliert

werden. Daher sind sie, gemäß dem Hauptresultat dieses Kapitels, mindestens für beschränkte Sprachen alle äquivalent.

7.2 Definitionen

Eine formale Sprache wird *strikt beschränkt* genannt, wenn sie für ein festes $m \geq 0$ eine Teilmenge von $a_1^* a_2^* \cdots a_m^*$ für paarweise verschiedene Symbole $a_1, a_2, \ldots, a_m$ ist. Eingabewörter werden den Automaten zwischen Endmarkierungen vorgelegt, und es wird sich als bequem erweisen, diesen Markierungen die Namen a_0 (linker Endmarker) und a_{m+1} (rechter Endmarker) zu geben. Eine maximale Folge von Eingabesymbolen eines Typs a_i bezeichnen wir als *Block*. Eine Sprache ist *beschränkt*, falls sie für ein festes $m \geq 0$ eine Teilmenge von $w_1^* w_2^* \cdots w_m^*$ für (nicht notwendig verschiedene) Wörter $w_1, w_2 \ldots, w_m$ ist. Wir nennen dann $w_1^* w_2^* \cdots w_m^*$ auch die *Schranke* der Sprache. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass keines der Wörter leer ist. Leider ist die Verwendung der Begriffe „strikt beschränkt“ und „beschränkt“ (bzw. ihrer englischen Gegenstücke „strictly bounded“ und „bounded“) schwankend, und häufig wird durch „beschränkt“ lediglich der Spezialfall „strikt beschränkt“ bezeichnet.

Die Berechenbarkeitsmodelle, die hier untersucht werden, sind *k Kopf Automaten*, *beschränkte Zählerautomaten mit k Zählern* und *Registermaschinen mit k beschränkten Registern*, siehe [50]. Die beiden erstgenannten Modell verfügen über Eingabebänder, von denen nur gelesen werden kann, und die wie oben beschrieben durch Endmarkierungen begrenzt sind. Der k Kopf Automat greift auf die Eingabe mittels k Köpfen oder Zeigern zu, die unabhängig voneinander vorwärts oder rückwärts bewegt werden können. Diese Köpfe geben der endlichen Kontrolle jeweils an, welches Eingabesymbol sie lesen. In diesem Modell ist es nicht direkt möglich, festzustellen, ob zwei Köpfe dasselbe Feld lesen. Im Gegensatz zu dem k Kopf Automaten besitzt ein Automat mit beschränkten Zählern nur einen Eingabekopf, der in der eben beschriebenen Weise arbeitet. Die übrigen Speicher sind Zähler mit den Operationen Inkrement und Dekrement um jeweils eine Stufe und einem Test auf Null. Die Kapazität der Zähler ist jeweils auf die Eingabelänge beschränkt, was dem Automatenmodell seinen Namen gibt. Zu beachten ist, dass ein solcher Automat mit k Zählern als weiteren Speicher die Position des Eingabekopfes hat. Wir werden daher Automaten mit k Zählern und solche mit $k+1$ Köpfen vergleichen.

Die Automaten beginnen ihre Arbeit mit allen Köpfen auf dem ersten Symbol der Eingabe, falls sie nicht leer ist, anderenfalls auf a_0. Alle Zähler haben anfangs den Wert Null.

Eine Registermaschine erhält als Eingabe eine Zahl im ersten Register, alle anderen Register werden anfangs auf Null gesetzt. Wir vergleichen Registermaschinen mit anderen Typen von Automaten, indem wir nichtnegative ganze Zahlen und Zeichenketten über einem einelementigen Alphabet in der üblichen Weise identifizieren. Alle Maschinen haben deterministische und nichtdeterministische Varianten und akzeptieren eine Eingabe durch Übergang in einen Endzustand.

7.3 Die Ergebnisse

Zunächst soll die Problematik dargestellt werden, welche die Übertragung der Simulation eines Automatenmodells durch ein zweites erschwert, wenn wir an der Verallgemeinerung von strikt beschränkten Sprachen zu beschränkten Sprachen interessiert sind.

Ein Wort $v \in \Sigma^*$ in der Form

$$v = w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m}$$

zu schreiben, suggeriert, dass durch Anwendung eines Parsingschemas, z.B. einer endlichen Vorausschau, die Grenzen zwischen den m Segmenten der Eingabe erkannt werden können. Wir geben einige Beispiele, an denen deutlich wird, dass die Situation komplizierter ist.

Es ist naheliegend, für eine strikt beschränkte Sprache $L \subseteq a_1^* a_2^* \cdots a_m^*$ die Sprache

$$L' = \{w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m} \mid a_1^{x_1} a_2^{x_2} \cdots a_m^{x_m} \in L\}$$

zu betrachten, wobei $w_1, w_2, \ldots, w_m$ nun beliebige Wörter über einem endlichen Alphabet Σ sein dürfen. Es ist im Allgemeinen nicht wahr, dass ein Automatenmodell von der Art, wie sie im Weiteren untersucht wird, L genau dann erkennen kann, wenn dies für L' gilt. In folgendem Spezialfall ist dies sogar explizit falsch. Sei $m = 2$ und

$$L = \{a_1^x a_2^x \mid x \geq 0\}.$$

Offensichtlich ist L keine reguläre Sprache. Wählen wir aber $w_1 = w_2 = a$, dann erhalten wir gemäß der obigen Beziehung die Sprache

$$L' = \{w_1^{x_1} w_2^{x_2} \mid a_1^{x_1} a_2^{x_2} \in L\} = \{a^{2x} \mid x \geq 0\}.$$

Dies ist eine reguläre Sprache. Ein Automatenmodell, das die weiter unten formulierten Anforderungen erfüllt, ist das des deterministischen endlichen Zweiweg-Automaten. Dieser Automatentyp akzeptiert genau die regulären

Sprachen [62, 68]. Somit haben wir den Widerspruch, dass L' von einem Automaten dieses Typs akzeptiert wird, nicht aber L.

Ein insofern weniger aussagekräftiges Beispiel, als es auf ein offenes Problem führt, ist die Sprache

$$\{a^n b^{n^2-n} \mid n \geq 1\}.$$

Eine triviale Modifikation des Algorithmus aus [55] zeigt, dass diese Sprache in der Klasse der von deterministischen Zweiweg-Automaten mit einem Zähler akzeptierten Sprachen (2DC) liegt. Wenn wir wiederum jedes Symbol in ein a übergehen lassen, dann erhalten wir

$$\{a^{n^2} \mid n \geq 1\},$$

eine Sprache, deren Mitgliedschaft in 2DC offen ist. Es ist sogar eine nichttriviale Aufgabe, einen Algorithmus für eine entsprechende Maschine mit einem Keller statt eines Zählers zu entwickeln [51]. Daher ist eher anzunehmen, dass auch hier keine direkte Korrespondenz zwischen strikt beschränkten und beschränkten Sprachen in der oben definierten Weise existiert.

Die eben besprochenen Beispiele beruhten darauf, dass die Wörter in der Folge $w_1, w_2 \ldots, w_m$, die die Schranke der betrachteten Sprache bildeten, nicht verschieden waren. Aber selbst für verschiedene Wörter, die die Blöcke bilden, ist die Zerlegung in Segmente problematisch.

Sei $w_1 = a$, $w_2 = aba$, und $w_3 = aab$. Nun betrachten wir die Folgen von Wörtern $v_n = (aab)^n a$ und $v'_n = v_n ab$. All diese Wörter sind Elemente von $w_1^* w_2^* w_3^*$. Aber während $v_n = w_1 w_2^n$, müssen wir $v'_n = w_3^{n+1}$ schreiben. Daher reicht keine endliche Vorausschau ausgehend von der ersten Position in v_n oder v'_n aus, um zwischen den beiden Fällen zu unterscheiden. Die Situation wird noch komplexer, wenn wir einen vierten Block mit $w_4 = a$ einführen, da dann v_n in zwei verschiedenen Arten in $v_n = w_1 w_2^n = w_3^n w_4$ zerlegt werden kann.

Wir lösen die letztgenannte Problematik, indem wir eine eindeutige, gierige Faktorisierung für jedes Wort einer beschränkten Sprache einführen. Sei $v = w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m}$ so, dass für $v = w_1^{x'_1} w_2^{x'_2} \cdots w_m^{x'_m}$ gilt

$$\left(\forall j \leq m : x_j = x'_j\right) \vee \exists i < m : \left(\forall j \leq i : x_j = x'_j\right) \wedge x_{i+1} > x'_{i+1}.$$

Dies bedeutet, dass die gierige Faktorisierung maximal im Bezug auf w_1, w_2, …, w_m ist, wenn wir die lexikographische Ordnung der Vektoren von Exponenten betrachten. Hier ist es wichtig, dass keines der Wörter leer ist.

Sei C eine Klasse von Maschinen, die mit einer endlichen Anzahl von Zweiweg-Köpfen ausgestattet sind. Diese Köpfe sollen sich auf der Eingabe

frei bewegen können, dürfen sie aber nicht verändern. Diese Beschreibung ist keine formale Definition, aber es dürfte in jedem Fall klar sein, ob eine Klasse von Maschinen diese Charakterisierung erfüllt. Im Besonderen werden endliche Mehrkopf-Zweiweg-Automaten und Turingmaschinen mit separatem platzbeschränkten Arbeitsband von dieser Beschreibung erfasst, während beispielsweise Einweg-Automaten, Maschinen mit Umkehrbeschränkung auf der Eingabe und Realzeit-Maschinen die Bedingung nicht erfüllen. Klassen, die die Bedingung erfüllen, werden wir *anpassungsfähige Zweiweg-Maschinen* nennen.

Sei $w_1^* w_2^* \cdots w_m^*$ die Schranke einer Sprache. Wir definieren als $L_i = w_i w_i^* w_{i+1}^* \cdots w_m^*$ für $1 \leq i \leq m$ diejenigen Wörter, die eine Faktorisierung gemäß der Schranke erlauben, welche in nichttrivialer Weise mit w_i beginnt.

Diese Darstellung zeigt sofort:

Beobachtung 3 *Die Sprachen L_i sind regulär.*

Im nächsten Schritt definieren wir die Sprachen

$$\begin{aligned} R_i \;=\; \{ & w_1^{x_1} \cdots w_{i-1}^{x_{i-1}} w_i^{x_i} = v \\ & \mid\; x_1, \ldots, x_i \text{ sind die Exponenten} \\ & \quad \text{der gierigen Faktorisierung von } v,\; x_i \geq 1\} \end{aligned}$$

für $1 \leq i \leq m$, bestehend aus den Wörtern, die eine gierige Faktorisierung mit einem w_i als letztem Teilwort erlauben.

Lemma 6 *Die Sprachen R_i sind regulär.*

Beweis. Wir beschreiben deterministische Zweiweg-Automaten M_i, welche die Sprachen R_i akzeptieren. Diese Automaten, die bereits oben erwähnt wurden, sind mit einem Kopf ausgestattet, der linkes und rechtes Ende der Eingabe erkennen kann, und charakterisieren bekanntermaßen die regulären Sprachen [62, 68]. Für jedes $1 \leq j \leq m$ sei N_j ein vollständiger deterministischer endlicher Einweg-Automat, der L_j in einer *rechts-links* Kopfbewegung über die gesamte Eingabe akzeptiert. Gemäß den Abschlusseigenschaften der regulären Sprachen ist L_j^R (die Spiegelsprache zu L_j) regulär, und aus einem Automaten für L_j^R kann ein Automat mit der gewünschten Eigenschaft durch Umkehrung der Leserichtung gewonnen werden. Der Automat M_i beginnt seine Arbeit, indem er seinen Eingabekopf an das rechte Ende der Eingabe bewegt, den Kopf von rechts nach links über die Eingabe führt, und dabei parallele, vollständige Läufe der Automaten N_j simuliert. Er hält die Zustände der Automaten N_j am Ende ihrer Durchläufe fest und initialisiert eine Variable p mit dem Wert 0. Nun wiederholt M_i die folgenden

Schritte. Er ermittelt den kleinsten Index q mit der Eigenschaft, dass $q \geq p$ und N_q einen akzeptierenden Zustand einnimmt. Falls kein solcher Automat existiert, lehnt M_i die Eingabe ab. Anderenfalls aktualisiert M_i den Wert von p auf q. Er bewegt seinen Kopf über das Wort w_q und simuliert jeden der Automaten N_j rückwärts, um den Zustand von N_j vor dem Lesen von w_q zu ermitteln. Diese Rückwärtssimulation wird gemäß Lemma 3 von [27] durchgeführt. Diese Technik wird ebenfalls in dem Beweis von Lemma 4 aus [32] erläutert.

Schließlich erreicht M_i (unter der Annahme, dass der Automat nicht ablehnen musste) den linken Endmarker. Er akzeptiert genau dann, wenn $q = i$. Zu beachten ist, dass M_i Schritt für Schritt die gierige Faktorisierung der Eingabe konstruiert hat und dass q den Index des letzten Wortes w_q in dieser Faktorisierung angibt. □

Bemerkung: Ein anderer Beweis kann gegeben werden, indem ausgenutzt wird, dass die von alternierenden endlichen Automaten akzeptierten Sprachen regulär sind.

Satz 13 *Seien C_1 und C_2 Klassen anpassungsfähiger Zweiweg-Maschinen. Wenn die Klasse strikt beschränkter Sprachen, die durch Maschinen der Klasse C_1 akzeptiert werden, in der durch C_2 akzeptierten enthalten ist, dann gilt eine entsprechende Inklusion für beschränkte Sprachen.*

Beweis. Sei M_1 eine Maschine aus C_1 mit einer Eingabe, die in der Schranke liegt. Wir erinnern zunächst an die Konstruktion, die für Zweiweg-Maschinen Abschluss unter inversen Homomorphismen zeigt. Hier ist der Homomorphismus h durch $h(a_i) = w_i$ für $1 \leq i \leq m$, $h(a_0) = a_0$ und $h(a_{m+1}) = a_{m+1}$ gegeben. Wir transformieren M_1 in $\hat{M}_1$ mit der Eigenschaft, dass $\hat{M}_1$ ein Wort v genau dann akzeptiert, wenn M_1 das Wort $h(v)$ akzeptiert, im Besonderen akzeptiert $\hat{M}_1$ ein Wort $a_1^{x_1} a_2^{x_2} \cdots a_m^{x_m}$ genau dann, wenn M_1 das Wort $w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m}$ akzeptiert.

Wir konzentrieren uns auf einen Kopf von $\hat{M}_1$ und nehmen an, dass er das Symbol a_i liest. Dann wird M_1 mit seinem korrespondierenden Kopf auf w_i simuliert und $\hat{M}_1$ hält in seiner endlichen Kontrolle eine der $|w_i|$ Positionen von M_1's Kopf auf w_i fest.

Während die simulierten Schritte den Kopf von M_1 im Bereich von w_i lassen, bewegt sich der Kopf von $\hat{M}_1$ nicht und das Symbol, das M_1 liest, wird durch die endliche Kontrolle von $\hat{M}_i$ zur Verfügung gestellt. Im Verlauf der Simulation verlässt der simulierte Kopf den Bereich von w_i möglicherweise am rechten oder linken Ende. Dann bewegt $\hat{M}_1$ den entsprechenden Kopf in die gleiche Richtung, hält das neu gelesene Symbol a_j und die ursprüngliche Position des simulierten Kopfes in der endlichen Kontrolle fest, und setzt die

Simulation fort. Die Startkonfiguration von $\hat{M}_1$ ist so, dass alle Köpfe das erste Eingabesymbol lesen.

Wir zeigen im nächsten Schritt, wie $\hat{M}_1$'s Arbeit auf $a_1^{x_1} a_2^{x_2} \cdots a_m^{x_m}$ simuliert werden kann, wenn die tatsächliche Eingabe $w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m}$ ist und die Exponenten diejenigen aus der gierigen Faktorisierung der Eingabe sind. Sollte die Eingabe keine Faktorisierung der Eingabe gemäß der Schranke zulassen (was auf Grund der Regularität der Schranke leicht zu testen ist), dann wird sie abgelehnt. Wir können von nun an also annehmen, dass die Eingabe sich in die durch die Schranke gegebene Blockstruktur einteilen lässt. Für jeden Kopf und jeden Block initialisiert der Simulator zwei vollständige deterministische endliche Automaten. Der erste akzeptiert L_i und liest die Eingabe von rechts nach links, während der andere R_i in der üblichen Weise von links nach rechts akzeptiert. Um den Automaten für L_i zu initialisieren, wird die gesamte Eingabe gelesen. In jedem Schritt aktualisiert der Simulator die Zustände dieser Automaten, um die Zustände zu reflektieren, die sie erreichen würden, wenn sie alle Wörter der Blockstruktur lesen würden, mit Ausnahme des Wortes, das von dem jeweiligen Kopf gelesen wird. Nehmen wir also an, dass die Eingabe die Form $w_1^{x_1} w_2^{x_2} \cdots w_i^{x_i} \cdots w_m^{x_m}$ hat und ein Kopf das $j$te Teilwort w_i liest ($1 \leq j \leq x_i$), dann wäre der Automat für R_i in dem Zustand nach Lesen von $w_1^{x_1} w_2^{x_2} \cdots w_i^{j-1}$, während der zu L_i gehörende Automat in den Zustand nach Lesen von $w_i^{x_i - j} \cdots w_m^{x_m}$ (von rechts nach links) einnehmen würde. Indem die aktuellen Zustände dieser simulierten Automaten auf akzeptierende untersucht werden, kann der Übergang von einem Block zu einem benachbarten erkannt werden. Wenn ein Kopf der (simulierten) Maschine $\hat{M}_1$ von einem Symbol a_q auf ein Symbol a_p bewegt wird, dann wird w_q in die Automaten für die L_i eingegeben, während die Automaten für R_i rückwärts simuliert werden, um ihre Zustände vor dem Lesen von w_p zu bestimmen (unter der Annahme, dass $p > 0$). Dies ist nach Lemma 3 von [27] in einem „Ausflug“ des Kopfes möglich, der seine ursprüngliche Position nicht zerstört. Ein entsprechender Prozess mit vertauschten Rollen von „links“ und „rechts“ wird für Rechtsbewegungen durchgeführt.

Schließlich wird die Komponenten des Simulators, die auf $\hat{M}_1$ beruht, durch einen äquivalenten Automaten aus $\hat{M}_2$ aus C_2 ersetzt. Der resultierende Automat M_2 ist in der Klasse C_2 und akzeptiert eine Eingabe genau dann, wenn M_1 dies tut. □

Korollar 5 *Wenn die Klassen der strikt beschränkten Sprachen, die durch Maschinen der Klassen C_1 und C_2 akzeptiert werden, identisch sind, dann gilt dies auch für die Klassen beschränkter Sprachen.*

Das Äquivalenzresultat aus [30] für strikt beschränkte Sprachen ergibt nun:

Korollar 6 *Deterministische umkehrbeschränkte Kellerautomaten und deterministische umkehrbeschränkte Automaten mit einem Zähler sind über beschränkten Sprachen äquivalent.*

Anmerkung: Diese Verallgemeinerung wurde, ohne Angabe einer Konstruktion, in [31, S. 361] angekündigt. In der Zeitschriftenversion [30] wird nur die schwächere Aussage über strikt beschränkte Sprachen bewiesen, wobei letztere als beschränkte Sprachen bezeichnet werden.

Auf der Basis der Ideen von Geffert [18, 19] wurde die Technik des Beweises des Satzes von Savitch in [73, Theorem 10.2.2] auf strikt beschränkte Sprachen und sublogarithmische Platzschranken angewendet. Dies kann nun weiter verallgemeinert werden.

Korollar 7 *Sei L eine Teilmenge von $w_1^* \cdots w_m^*$ und $s(n)$ eine Platzschranke. Wenn $L \in NSPACE(s(n))$ dann gilt $L \in DSPACE(s^2(n))$.*

In ähnlicher Weise kann die Technik zur Komplementierung für nichtdeterministische platzbeschränkte Turingmaschinen auf beliebige Platzschranken ausgedehnt werden [18, 19] [73, Theorem 7.4.2] (wir nehmen hier starke Platzkomplexität an).

Korollar 8 *Sei L eine Teilmenge von $w_1^* \cdots w_m^*$ und $s(n)$ eine Platzschranke. Wenn $L \in NSPACE(s(n))$, dann $\overline{L} \in NSPACE(s(n))$, wobei $\overline{L}$ das Komplement von L bezeichnet.*

Beweis. Sei M_1 eine Turingmaschine, die L mit der gegebenen Platzschranke akzeptiert. Die Sprache

$$L' = \{a_1^{x_1} a_2^{x_2} \cdots a_m^{x_m} \mid w_1^{x_1} w_2^{x_2} \cdots w_m^{x_m} \notin L\}$$

kann von einer Turingmaschine $\hat{M}_1$ mit dem gleichen Platzbedarf auf Eingabe v simuliert werden, wie ihn M_1 auf $h(v)$ hat. Nun wird $\hat{M}_1$ wie im Beweis von Satz 13 simuliert und zusätzlich die reguläre Sprache der nicht „wohlgeformten“ Zeichenketten akzeptiert, um

$$\overline{L} = h(L') \cup \overline{w_1^* w_2^* \cdots w_m^*}$$

zu erhalten. □

Wir kommen nun zur Hauptanwendung der oben entwickelten Technik, nämlich der Äquivalenz zwischen Köpfen und Zählern für deterministische endliche Automaten.

Um die Darstellung im Folgenden zu vereinfachen, nehmen wir ohne Beschränkung der Allgemeinheit an, dass Mehrkopf-Automaten in jedem Schritt

ihrer Berechnung genau einen Kopf bewegen. Angenommen, ein Mehrkopf-Automat arbeitet auf einer Eingabe aus einer strikt beschränkten Sprache. Wir nennen jeden Schritt, in dem ein Kopf von einem Block in einen benachbarten übergeht, ein *Ereignis*. Wir sagen, dass die Kopfbewegung in diesem Schritt das Ereignis *auslöst*.

Lemma 7 *Sei die Eingabe eines deterministischen Mehrkopf-Automaten im strikten Sinne beschränkt. Es ist möglich zu bestimmen, ob ein Kopf das zeitlich nächste Ereignis auslösen kann (unter der Voraussetzung, dass dies kein anderer Kopf tut), indem ein Eingabesegment mit von der Eingabe unabhängiger Länge um den Kopf herum inspiziert wird (abgesehen von Verkleinerungen des Segments an den Eingabeenden). Wenn der Kopf das nächste Ereignis auslösen kann, dann wird ebenfalls ermittelt, an welcher Blockgrenze dies geschieht.*

Beweis. Habe der Automat r Zustände. Wenn der Kopf sich mindestens r Positionen von seiner ursprünglichen Position innerhalb des gleichen Blocks entfernt hat, bevor das nächste Ereignis eintritt, dann muss mindestens ein Zustand an zwei verschiedenen Positionen vorgekommen sein. Der Automat, der in dieser Phase die gleiche Information von seinen Köpfen erhält, setzt seine Arbeit in einer Schleife fort, bis das nächste Ereignis eintritt (vorausgesetzt, es gibt ein solches). Daher ist es ausreichend, den Automaten in einem Eingabesegment der Länge $2r - 1$ um den betrachteten Kopf herum zu simulieren (oder in einem kleineren Segment, falls diese Länge die Entfernung zu einer Blockgrenze übersteigt). Wir nehmen an, dass kein anderer Kopf das Ereignis auslöst. Daher sind höchstens $2r^2 - r$ partielle Konfigurationen bestehend aus Zustand, durch die Köpfe gelesenen Symbolen und Positionen des Kopfes möglich, bevor der betrachtete Kopf entweder das Segment verlässt, der Automat in eine Schleife gerät (wobei der Kopf sich möglicherweise in Richtung einer Blockgrenze bewegt) oder ein anderer Kopf ein Ereignis auslöst. In den ersten beiden Fällen kann die Blockgrenze, an der das nächste Ereignis (möglicherweise) stattfindet, aus der Simulation erschlossen werden. □

Satz 14 *Deterministische endliche Automaten mit k Zweiweg-Köpfen und deterministische beschränkte Zählerautomaten mit $k - 1$ Zählern über strikt beschränkten Sprachen sind äquivalent.*

Beweis. Es sollte klar sein, dass jeder beschränkte Zählerautomat mit $k - 1$ Zählern durch einen deterministischen Automaten mit k Köpfen simuliert werden kann, und dies unabhängig von der Struktur der Eingabezeichenkette.

Der Mehrkopfautomat simuliert den Eingabekopf des Zählerautomaten mit der Hilfe eines seiner Köpfe und nutzt seine übrigen $k-1$ Köpfe, um die Zählerstände des Zählerautomaten als Abstände der Köpfe von der linken Endmarkierung zu kodieren.

Für die umgekehrte Richtung der Simulation beschreiben wir einen beschränkten Zählerautomaten C, der die Arbeit eines gegebenen Mehrkopfautomaten M nachbildet. Um Verwirrung bezüglich der Köpfe der beiden Automaten zu vermeiden, nennen wir den (einzigen) Kopf von C seinen Zeiger.

Wir zerlegen die Berechnung des Mehrkopfautomaten M in Intervalle. Jedes Intervall beginnt mit mindestens einem Kopf auf einem Eingabefeld, das direkt einer Blockgrenze benachbart ist. Einer dieser Köpfe wird durch den Zeiger von C repräsentiert. Zu beachten ist, dass die Anfangskonfiguration diese Eigenschaft hat. jedes Intervall (außer dem letzten) endet, wenn das nächste Ereignis eintritt. Nach diesem nächsten Ereignis ist der Automat wieder in einer Konfiguration, in der das nächste Intervall starten kann.

Der Zählerautomat C aktualisiert stets die Symbole, die durch die (simulierten) Köpfe von M gelesen werden (anfangs das erste Eingabesymbol oder die rechte Endmarkierung) und hält diese Information in der endlichen Kontrolle. Die Zähler und der Zeiger von C werden den Köpfen von M zugeordnet, aber diese Zuordnung kann sich im Verlauf der Simulation ändern und wird ebenfalls von C gespeichert. Ein Zähler, der einem Kopf zugeordnet ist, speichert die Anzahl der Symbole innerhalb des Blocks, der durch den Kopf gelesen wird, und die sich links oder rechts von seiner Position befinden. Der Automat C speichert ebenfalls, welche dieser beiden Zahlen (die selbstverständlich durch die Eingabelänge beschränkt sind) gespeichert wird.

Wir beginnen unsere Beschreibung des Algorithmus, der durch C ausgeführt wird, in einer Konfiguration mit der Eigenschaft, dass mindestens ein Kopf von M einer Blockgrenze direkt benachbart ist. Einer dieser Köpfe wird direkt durch Cs Zeiger repräsentiert. Zunächst bewegt C den Zeiger auf jeden Block, der von einem Kopf von M besucht wird. Diese Blöcke können wegen der strikten Beschränkung der Eingabe aus den gespeicherten Symbolen abgelesen werden, denn Symbole verschiedener Blöcke sind nach Definition verschieden. Für jeden dieser relevanten Blöcke bewegt C den Zeiger an die Blockgrenze, die durch den Typ der für den betrachteten Kopf gespeicherten Entfernung angegeben wird. Genauer wird der Zeiger für einen Abstand von der linken Blockgrenze an diese Grenze bewegt, und umgekehrt an die rechte Grenze, falls dieser Abstand gespeichert worden ist. Solange der entsprechende Zähler noch nicht den Stand Null erreicht hat, bewegt C den Kopf von der Blockgrenze weg und dekrementiert für jeden Schritt den Zähler. Auf diese Weise bewegt C den Zeiger auf die Position, den der

betrachtete Kopf von M in dieser Phase der Simulation einnehmen würde. Dann ermittelt C, ob es sich bei diesem Kopf um einen solchen handelt, der das nächste Ereignis auslösen könnte. Dies ist gemäß Lemma 7 möglich. Zu diesem Zweck wird eine Umgebung des Zeigers von maximal $2r-1$ Symbolen gelesen, ohne dass dabei die Kopfposition zerstört wird. Der Test, ob ein solches Ereignis möglich ist, wird vollständig in der endlichen Kontrolle durchgeführt.

Nehmen wir an, dass der gerade untersuchte Kopf in der Lage ist, das nächste Ereignis an einer der Blockgrenzen auszulösen. Dann speichert C den Abstand des Kopfes von der entsprechenden Grenze und die Lage (rechts oder links) der Grenze relativ zur Kopfposition. Wenn der Block die Form $a_i^{x_i}$ hat und der Kopf ist auf Position $n \geq 1$ innerhalb des Blocks, dann bewegt der Zählerautomat seinen Zeiger auf die Grenze zu, an der das Ereignis eintritt, und misst durch paralleles Erhöhen des dem Kopf zugeordneten Zählers den Abstand zur Blockgrenze. Somit ist der Zählerstand letztlich entweder $n-1$ oder $x_i - n$. Wenn kein Ereignis durch den gerade betrachteten Kopf ausgelöst werden kann, dann wird einer der beiden Abstände gespeichert, z.B. der zum linken Rand.

Diese Operationen werden für jeden Kopf ausgeführt. Schließlich bewegt C seinen Zeiger zurück auf die ursprüngliche Position, was möglich ist, da sich diese Position neben einer Blockgrenze befand und daher in der endlichen Kontrolle gespeichert werden kann. Danach beginnt C, den Mehrkopfautomaten Schritt für Schritt zu simulieren. Hierbei werden Kopfbewegungen verschieden umgesetzt, je nachdem, ob der entsprende Kopf durch den Zeiger oder durch einen Zähler dargestellt werden. Die Simulation des durch den Zeiger repräsentierten Kopfes besteht einfach in direkt korrespondierenden Bewegungen. Für die Zähler ist die Simulation etwas komplizierter. Bei einer Rechtsbewegung wird der Zähler inkrementiert, wenn er einen Abstand zu einer links von dem Kopf liegenden Blockgrenze darstellt, und dekrementiert, falls es sich um den Abstand zu einer rechten Grenze handelt. Umgekehrt wird bei eine Linksbewegung der Abstand zu einer links von der aktuellen Kopfposition liegenden Grenze dekrementiert und zu einer rechts von der Position liegenden Grenze inkrementiert.

Wenn M in eine akzeptierende Konfiguration gerät, dann akzeptiert auch C. Wenn der Zeiger seinen Block verlässt, beginnt ein neues Intervall. Wenn ein Zähler den Wert Null hat und er in der nächsten Operation dekrementiert werden soll, dann wird diese Operation nicht ausgeführt. Stattdessen wird die aktuelle Position des Zeigers in diesem (leeren) Zähler festgehalten und die Funktion dieses Zählers und des Zeigers werden vertauscht. Das Symbol, das von dem Kopf gelesen wird dem nun der Zähler zugeordnet ist, und der Zustand von M werden aktualisiert. Ein neues Intervall kann beginnen.

Die Anfangskonfiguration von C besteht darin, dass alle Zähler leer sind und die in der endlichen Kontrolle gespeicherte Information dem ersten Eingabesymbol entspricht. Die Zuordnung von Zählern zu Köpfen ist beliebig, alle Zähler speichern den Abstand zur linken Blockgrenze. □

Mit Satz 13 erhalten wir:

Korollar 9 *Deterministische endliche Mehrkopf-Automaten mit k Zweiweg-Köpfen und deterministische beschränkte Zählerautomaten mit $k-1$ Zählern sind über beschränkten Sprachen äquivalent.*

In Lemma 5 aus [50] wird eine Simulation von k Kopf Automaten über einem einelementigen Alphabet durch Registermaschinen mit $k+1$ Registern beschrieben. Diese Simulation ist sehr speziell, da sie nur auf Eingabewörter angewendet werden kann, deren Länge eine Zweierpotenz ist. Wir verallgemeinern diese Simulation hier auf Wörter über einem einelementigen Alphabet mit einer beliebigen Länge.

Satz 15 *Jeder deterministische (nichtdeterministische) k Kopf Automat mit einem einelementigen Alphabet kann durch eine deterministische (nichtdeterministische) $k+1$ Registermaschine simuliert werden. Hierbei darf der Mehrkopfautomat sogar die Möglichkeit haben, Kopfpositionen auf Gleichheit zu testen.*

Beweis. Zunächst normalisieren wir den Mehrkopfautomaten mit der Möglichkeit zum Positionsvergleich in einer Weise, dass seine Köpfe immer in der gleichen, festen Reihenfolge auf dem Eingabeband erscheinen (falls einige Köpfe dasselbe Feld lesen, dann lassen wir unter ihnen jede Reihenfolge zu, womit diese Eigenschaft aufrecht erhalten werden kann). Die Normalisierung der Automaten bedeutet keine Einschränkung der Allgemeinheit, da die Rollen der Köpf innerhalb der endlichen Kontrolle vertauscht werden können.

Die Registermaschine, welche die Simulation eines k Kopf Automaten mit Hilfe von $k+1$ Registern durchführen soll, speichert in den Registern die Abstände der Köpfe zueinander bzw. zu einer benachbarten Endmarkierung. Unter dem Abstand verstehen wir hier die Anzahl der Schritte, die notwendig sind, um einen Kopf durch Rechtsbewegungen auf die Position eines anderen Kopfes oder auf eine Endmarkierung zu bewegen. Register 1 speichert den Abstand des letzten Kopfes zur rechten Endmarkierung. Wenn ein Kopf bewegt wird, werden durch die Registermaschine die beiden betroffenen Abstände aktualisiert. Ein kleines technisches Problem ist der Abstand zur linken Endmarkierung, der gemäß der obigen Definition -1 sein sollte.

Die Registermaschine speichert die Information, ob der bzw. die Köpfe, die sich am weitesten links befinden, die Endmarkierung lesen, in ihrer endlichen Kontrolle. Auf diese Weise kann die Zählerkapazität auf die Eingabelänge begrenzt werden. □

Kapitel 8

Separationsresultate für Rebound-Automaten

Das Ziel dieses Kapitels ist es, für Varianten eines Berechenbarkeitsmodell, das Sprachen mit logarithmischer Platzkomplexität charakterisiert, Separationsergebnisse zu erzielen.

8.1 Einleitung

Ein klassisches Ergebnis in der Theorie zweidimensionaler Automaten besagt, dass für endliche Automaten Nichtdeterminismus mächtiger als Determinismus ist. Diese Trennung, die einen wichtigen Unterschied zu eindimensionalen Automaten bedeutet (wo beide Arbeitsweisen genau die regulären Sprachen charakterisieren), wurde durch Blum und Hewitt [4] gezeigt, ein Beweis ist ebenfalls in Rosenfelds Monographie angegeben [66, Theorem 4.3.4].

Ein Problem bei der Untersuchung zwei- oder höherdimensionaler Berechenbarkeitsmodelle mit variierenden Eigenschaften ist es allerdings, dass Klassen von akzeptierten Eingabemengen nicht direkt mit den eindimensionalen Gegenstücken verglichen werden können. Dies motivierte die Beschränkung auf *Rebound-Automaten*. Auf die Übersetzung dieses Ausdrucks, der sich etwa mit „Rückprall“ wiedergeben ließe, wird hier verzichtet. Solche Automaten arbeiten, wie die von Blum und Hewitt untersuchten, auf einer quadratischen Eingabefläche. Allerdings enthält hier nur die erste Zeile die Eingabe, alle anderen Felder sind mit Leersymbolen belegt. Der Rand ist für den Automaten erkennbar und die „abprallende“ Bewegung des Kopfes, nachdem der Rand erreicht worden ist, dürfte bei der Namensgebung Pate gestanden haben. Auf diese Weise definiert ein Rebound-Automat eine akzeptierte Sprache eindimensionaler Wörter.

Deterministische Rebound-Automaten wurden von Sugata, Umeo und Morita eingeführt und erstmals untersucht. Eine Erweiterung auf nichtdeterministische Automaten wird bereits in ihrem frühen Artikel [72] kurz erwähnt. Die Frage, ob die aus der Definition folgende Inklusion der Klassen von Sprachen, die durch deterministische und nichtdeterministische Automaten erkannt werden, echt ist, tauchte explizit in [34] als offenes Problem auf und wurde in den Veröffentlichungen [76, 56] erneut formuliert. Diese Frage wird nicht durch die von Blum und Hewitt eingeführte Separationstechnik beantwortet, denn das von ihnen verwendete Argument beruht gerade darauf, dass ein Feld außerhalb der erste Zeile der Eingabe markiert ist.

Das Hauptresultat dieses Kapitels ist eine Trennung von Determinismus und Nichtdeterminismus für Rebound-Automaten, wodurch dieses lange offene Problem gelöst wird. Tatsächlich erhalten wir dadurch ebenfalls einen neuen Beweis für die von Blum und Hewitt [4] angegebene Trennung, denn durch Einbeziehung der Leersymbole ergibt sich eine Menge von zweidimensionalen Eingaben. Würde diese von einem deterministischen zweidimensionalen Automaten akzeptiert, dann auch von einem deterministischen Rebound-Automaten, was somit unmöglich ist.

Desweiteren definieren wir eine Sprache, die durch einen deterministischen Einweg-Zählerautomaten akzeptiert werden kann, aber nicht durch einen (sogar nichtdeterministischen) Rebound-Automaten, womit ein weiteres offenes Problem aus [34] gelöst wird. Da diese Sprache im Durchschnitt von durch deterministische Einweg-Zählerautomaten akzeptierten Sprachen und deterministisch linear kontextfreien Sprachen liegt, erhalten wir auch eine Verschärfung der Trennungen aus [67, 56] und [34], wo nichtdeterministische Rebound-Automaten von Zweiweg-Zählerautomaten und kontextfreien Sprachen getrennt wurden.

8.2 Die generelle Strategie

Eine Technik, die sich auf viele Varianten von zweidimensionalen Automaten anwenden lässt, ist eine Analyse des Flusses von Information über die Grenzen zwischen zwei Bereichen der Eingabe oder auch Gruppen von solchen Bereichen. In unseren Beweisen nehmen wir an, dass eine Aufzählung der Grenzen zwischen den Feldern der Eingabe festgelegt wird, beispielsweise von links nach rechts und vom unteren Teil der Eingabe nach oben. Dann ist es möglich, eine Teilmenge dieser Grenzen zu spezifizieren, die wir als *Interface* der Bereiche bezeichnen. Wenn die Eingabe genügend lang ist, können die Elemente des Interfaces festgelegt werden, unabhängig von der genauen Länge.

Bei der Trennung zweier Berechenbarkeitsmodelle A und B sind die wichtigen Schritte die Auswahl einer trennenden Sprache L und eines Interfaces mit den folgenden Eigenschaften:

1. Modell A kann eine ausreichende Menge an Information über das Interface transportieren, um L entscheiden zu können.

2. Modell B kann dies nicht leisten.

Nachdem eine trennende Sprache L festgelegt ist und wir einen Algorithmus zur Akzeptierung von L durch eine Maschine der Klasse A angegeben haben, müssen wir daher ein Interface (abhängig von der Größe der Maschine vom Typ B) finden und begründen, dass der Automat der Klasse B auf mindestens einer Eingabe nicht in der Lage ist, eine korrekte Enteschеidung zu treffen. Die werden wir erreichen, indem wir Teile von Eingaben entlang dem Interface zusammenfügen, so dass sich ein Element des Komplements von L ergibt.

Zur Vereinfachung unserer Diskussion nehmen wir ohne Beschränkung der Allgemeinheit an, dass alle Rebound-Automaten, wenn sie akzeptieren, die Position des Kopfes auf dem oberen linken Feld der Eingabe ist.

8.3 Trennung von Determinismus und Nichtdeterminismus für Rebound-Automaten

Die Sprache V, die in der Trennung von deterministischen und nichtdeterministischen Rebound-Automaten verwendet wird, ist wie folgt definiert:

$$\begin{aligned} V \;=\; & \{a^j b^k 0^{7\cdot 2^r} y_{2^r-j-k} \cdots y_1 \mid 2^{r+3} - j - k \geq (2j+1)2^k, \quad y_{(2j+1)2^k} = 1, \\ & \forall 1 \leq i \leq 2^r - j - k : y_i \in \{0,1\}\} \end{aligned}$$

Lemma 8 *Die Sprache V kann von einem nichtdeterministischen Rebound-Automaten akzeptiert werden.*

Beweis. Wir skizzieren einen Algorithmus, der von einem nichtdeterministischen Rebound-Automaten A ausgeführt werden kann um V zu akzeptieren. Im Folgenden werden wir oft Zahlen erwähnen, die durch Positionen des Eingabekopfes auf der ersten Zeile der Eingabe repräsentiert werden. Wir verwenden an verschiedenen Stellen die Technik, solche Zahlenwerte durch eine Konstante zu dividieren oder mit einer solchen zu multiplizieren. Dies geschieht, indem der Kopf diagonal über die Eingabe bewegt wird, wobei das

Verhältnis aus vertikalen und horizontalen Schritten in einem Zyklus dieser Bewegungen so bemessen wird, dass sich der beabsichtigte Faktor ergibt. Ein eventueller Divisionsrest kann in der endlichen Kontrolle des Automaten gespeichert werden.

Zuerst prüft A, ob die erste Zeile der Eingabe a's, gefolgt von b's und dann eine Folge von Symbolen 0 und 1 enthält.

Er überprüft dann, dass seine Eingabelänge eine Zweierpotenz ist und die Form $m = 2^{r+3}$ hat. Dies kann erreicht werden, indem Spaltenpositionen wiederholt durch 2 geteilt werden. Wenn dabei schließlich 1 erhalten wird und die Anzahl der Iterationen mindestens 3 ist, wird diese Eigenschaft erfüllt.

Danach bewegt A seinen Kopf auf die am weitesten links stehende 1 (wenn vorhanden), subtrahiert $m - \frac{1}{8}m = 7 \cdot 2^r$ von der Spaltenposition und prüft, dass das gelesene Symbol 0 ist.

Dann berechnet A die Zahlen $p_i = (2j+1)2^i$ als Spaltenpositionen für $i = 0, 1, 2, \ldots$ (solange dies möglich ist) und benutzt diese Zahlen, um auf die Symbole y_{p_i} zuzugreifen. Für ein nichtdeterministisch ausgewähltes p_i mit der Eigenschaft, dass $y_{p_i} = 1$, stoppt der Automat dieses Vorgehen (wenn kein solches p_i existiert, lehnt A ab). Zu beachten ist, dass $p_i \leq 2^r$. Nun addiert A die Zahl $\frac{1}{2}m = 2^{r+2}$ zur Spaltenposition und teilt wiederholt durch 2 bis ein Divisionsrest 1 erscheint.

Danach multipliziert A mit 2, inkrementiert um 3 und multipliziert wiederholt mit 2 bis ein Wert größer oder gleich 2^{r+2} erreicht wird. Danach subtrahiert A den Wert 2^{r+2} und teilt durch 2. Diese Folge von Operationen wird wiederholt bis die letzte Division einen von Null verschiedenen Rest liefert.

Die arithmetischen Operationen, die mit der Position des Eingabekopfes durchgeführt werden, transformieren $(2j+1)2^i$ in $(2(j+1)+1)2^{i-1}$ für jedes $i \geq 1$. Für $j = 0$ ergibt sich durch die erste Transformation $3 \cdot 2^{i-1} < 2^{r+2}$, für $j > 0$ haben wir $(2(j+1)+1)2^{i-1} = ((2j+1)+2)2^{i-1} \leq (2j+1)2^i < 2^{r+2}$. Daher wird das Zählen korrekt durchgeführt.

Der letzte Wert, der als A's Kopfposition berechnet wird, ist $j + i + 1$. Daher kann A diesen Wert benutzen, um sicherzustellen, dass $p_k = (2j+1)2^k$ korrekt erraten wurde. Dies kann dadurch überprüft werden, dass das Symbol an Position $j + i + 1$ die am weitesten links befindliche 0 ist.

Insgesamt prüft A syntaktische Korrektheit und dass das Flag y_{p_k} eine 1 ist. □

Lemma 9 *Die Sprache V kann durch keinen deterministischen Rebound-Automaten akzeptiert werden.*

Beweis. Wir nehmen an, dass ein Rebound-Automat R mit einer Zustandsmenge Q ist, der V akzeptiert. Abhängig von der Anzahl der Zustände von

R werden wir eine Menge von möglichen Eingabezeichenketten mit der Eigenschaft auswählen, dass R nicht jede von ihnen korrekt akzeptieren oder ablehnen kann. Daher kann kein Automat existieren, der V akzeptiert.

Wir konzentrieren uns auf Eingabeketten, die einen Präfix $w = a^j b^k 0^{n-j-k}$ für ein $n \geq j + k$ besitzen. Die Anzahl der verschiedenen Möglichkeiten, wie ein solcher Präfix beschaffen sein kann, ist

$$\frac{n^2 + 3n + 2}{2} \geq \frac{n^2}{2}.$$

Wir spalten:

$*$	$*$	$*$	$\cdot$	$*$	$\cdot$	$*$	$*$
$*$	x_1	x_2	$\cdot$	x_n	$\cdot$	x_m	$*$
$*$			$\cdot$		$\cdot$		$*$
$*$			$\cdot$		$\cdot$		$*$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$*$			$\cdot$		$\cdot$		$*$
$*$	$*$	$*$	$\cdot$		$\cdot$	$*$	$*$

in

$*$	$*$	$*$	$\cdot$	$*$
$*$	x_1	x_2	$\cdot$	x_n

					$\cdot$	$*$	$*$
					$\cdot$	x_m	$*$
$*$			$\cdot$		$\cdot$		$*$
$*$			$\cdot$		$\cdot$		$*$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$
$*$			$\cdot$		$\cdot$		$*$
$*$	$*$	$*$	$\cdot$		$\cdot$	$*$	$*$

wobei jedes $x_i \in \{a, b, 0, 1\}$, n fest ist und m variiert. Sei $v = x_{n+1} \cdots x_m$ (die gesamte Eingabe ist somit wv).

Das Interface I besteht aus den $n+3$ Grenzen zwischen Eingabezellen, die die beiden Teile der Eingabe trennen. Wir definieren die partielle Funktion $C_v : I \times Q \to I \times Q$ durch

$$C_v(i, p) = (i', q)$$

falls R zu Position i' von I zurückkehrt und in Zustand q übergeht, wenn er w's Bereich der Eingabe an Position i von I verlässt und dabei in Zustand p übergeht. Wir bemerken, dass wenn $C_v = C_{v'}$ entweder wv und wv' beide akzeptiert werden oder beide von R abgelehnt werden. Dies gilt, weil die Segmente einer akzeptierenden Berechnung auf wv, die innerhalb von w liegen, mit den Anteilen außerhalb kombiniert werden können, um eine akzeptierende Berechnung auf wv' zu erhalten, und umgekehrt.

Die Anzahl verschiedener partieller Funktionen C_v für einen Automaten mit $s = |Q|$ Zuständen ist durch $(s(n+3)+1)^{s(n+3)}$ beschränkt. Da ein Automat, der eine nicht-triviale Sprache erkennt, mindestens einen akzeptierenden und einen nicht-akzeptierenden Zustand benötigt, können wir $s \geq 2$ annehmen.

Über die Kodierung als $(2j+1)2^k$ definiert jedes Wort $v \in \{0,1\}^*$ eine Menge $M_v \subseteq \{(j,k) \mid j,k \leq n\}$ in dem Sinne, dass $(j,k) \in M_v$ genau dann, wenn $a^j b^k 0^{n-j-k} v \in V$.

Um zu zeigen, dass der Automat R nicht existieren kann, reicht es aus, ein n mit der Eigenschaft zu finden, dass die Anzahl der Mengen M_v größer als die der partiellen Funktionen C_v ist. Denn dann gibt es v, v' mit $C_v = C_{v'}$ für $M_v \neq M_{v'}$ und wir können ein $(j,k) \in (M_v \setminus M_{v'}) \cup (M_{v'} \setminus M_v)$ auswählen. Nun akzeptiert R entweder sowohl $a^j b^k 0^{n-j-k} v$ als auch $a^j b^k 0^{n-j-k} v'$, oder keines dieser Wörter, im Gegensatz zur Definition von V.

Wir behaupten, dass $n = s^8$ die geforderte Eigenschaft hat, weil:

$$\begin{aligned}
(s(s^8+3)) \cdot \log_2(s(s^8+3)+1) &\leq 2s^9 \cdot \log_2(2s^9+1) \\
&\leq 2s^9 \cdot \log_2(4s^9) \\
&\leq 2s^9 \cdot (2 + 9\log_2 s) \\
&\leq 20s^{10} \\
&< 32s^{10} \\
&\leq \frac{s^6}{2} s^{10} \\
&= \frac{(s^8)^2}{2}
\end{aligned}$$

Daher ist $(s(n+3)+1)^{s(n+3)}$ (die obere Schranke für die Anzahl verschiedener C_v) kleiner als $2^{n^2/2}$ (die untere Schranke für die Anzahl verschiedener Mengen M_v). □

Durch Kombination der beiden Lemmata erhalten wir eine Lösung des offenen Problems aus [34, 76, 56]:

Satz 16 *Die Klasse der durch deterministische Rebound-Automaten akzeptierten Sprachen ist echt in der durch nichtdeterministische akzeptierten enthalten.*

8.4 Trennung von Rebound-Automaten und Einweg-Zählerautomaten

Wir geben zunächst die Definition der Sprache $W \subseteq \{0,1,\$,\#\}^*$ für die Trennung von Zählerautomaten und Rebound-Automaten an. Diese Sprache ist eine Teilmenge der regulären Sprache $W' = ((0+1)\$0^*\$)^*\#(0+1)^*$ und wird folgendermaßen definiert:

$$\begin{aligned} W \;=\; \{ & x_1\$0^{m_1}\$\cdots\$x_n\$0^{m_n}\$\#y_0\cdots y_r \mid \\ & \forall i \le n : x_i \in \{0,1\}, \\ & \forall i \le r : y_i \in \{0,1\}, \\ & y_p = 1, \text{ wobei } p = \sum_{i\in\{1,\ldots,n\}:x_i=1} m_i \} \end{aligned}$$

Die Intuition hinter dieser Definition ist, dass Flags x_i Zahlen m_i auswählen, die dann zu einer Summe ℓ aufaddiert werden. Das Segment nach # kodiert eine Menge von Zahlen $M = \{i \mid y_i = 1\}$ mit der Eigenschaft $\ell \in M$.

Lemma 10 *Die Sprache W kann von einem deterministischen Einweg-Zählerautomaten mit einem Zähler akzeptiert werden, wobei der Automat nur eine Umkehrung der Zählrichtung auf dem Zähler macht.*

Beweis. Wir beschreiben einen Algorithmus, der durch eine Automaten A von der im Lemma geforderten Art ausgeführt werden kann. Dabei nehmen wir stillschweigend an, dass A die Mitgliedschaft in der regulären Obermenge W' von W prüft.

Beginnend mit einem leeren Zähler addiert A jedes m_i zu dem Wert, der auf dem Zähler gespeichert ist, falls das zugehörige x_i den Wert 1 hat. Dies kann ohne Umkehrung der Zählrichtung geschehen. Nachdem das Symbol # gelesen wurde, akzeptiert der Automat genau dann, wenn er eine 1 liest und der Zähler den Wert Null hat. Für jedes in dieser Phase verarbeitete Symbol dekrementiert der Automat seinen Zähler. Wenn der Zähler dabei negativ werden würde oder das Eingabeende erreicht wird, ohne dass der Zähler den Wert Null annimmt, wird die Eingabe abgelehnt.

Vor dem Lesen des # wird der Zähler in jedem Schritt entweder inkrementiert oder unverändert gelassen, danach dekrementiert. Somit führt A höchstens eine Umkehrung durch. □

Lemma 11 *Die Sprache W kann nicht durch einen nichtdeterministischen Rebound-Automaten akzeptiert werden.*

Beweis. Wir führen die Annahme, dass es einen nichtdeterministischen Rebound-Automaten R, der in der Lage ist, W zu akzeptieren, zu einem Widerspruch.

Für jedes $n \geq 1$ konzentrieren wir uns auf Eingaben einer bestimmten Klasse, nämlich solchen der Form:

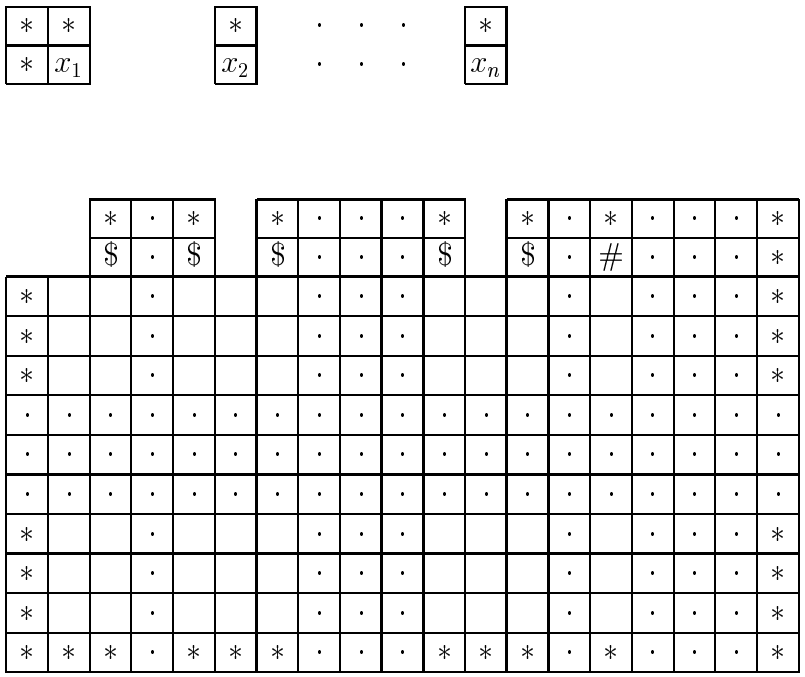
$$x_1\$0^{2^0}\$\cdots\$x_n\$0^{2^{n-1}}\$\#y_0\cdots y_r \tag{8.1}$$

mit $x_1, \ldots, x_n \in \{0, 1\}$ und $r = 2^n - 1$.

Wir zerlegen eine Eingabe

*	*	*	·	*	*	*	·	·	·	*	*	*	·	*	·	·	·	*
*	x_1	$	·	$	x_2	$	·	·	·	$	x_n	$	·	#	·	·	·	*
*			·				·	·	·				·		·	·	·	*
*			·				·	·	·				·		·	·	·	*
*			·				·	·	·				·		·	·	·	*
·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
*			·				·	·	·				·		·	·	·	*
*			·				·	·	·				·		·	·	·	*
*			·				·	·	·				·		·	·	·	*
*	*	*	·	*	*	*	·	·	·	*	*	*	·	*	·	·	·	*

mit $x_1, \ldots, x_n \in \{0, 1\}$ in



Das Interface I zwischen den Bereichen, welche die x_i enthalten, und dem Rest der Eingabe hat die Größe $5n-1$. Angenommen, der Rebound-Automat R hat die Menge innerer Zustände Q mit $s = |Q| \geq 2$. Für eine feste Eingabe werden die möglichen Rechnungen zwischen zwei „Besuchen“ von Bereichen der x durch die Relation $C \subseteq (I \times Q) \times (I \times Q)$ festgelegt, wobei $((i,p),(j,q)) \in C$ genau dann, wenn R einen der $x_1, \ldots, x_n$ enthaltenden Bereiche an Position i des Interface mit einem Übergang in Zustand p verlassen kann und ein solches später mit einem Übergang in Zustand q an der Position j wieder betritt.

Eingaben der Form (8.1) mit denselben $x_1, \ldots, x_n$ und derselben Relation C werden entweder alle akzeptiert, oder keines wird in der Sprache liegen, die durch R definiert ist.

Es gibt höchstens $2^{s^2(5n-1)^2}$ verschiedene Relationen, die für Automaten auf Eingaben der Form (8.1) auftreten können. Wir argumentieren nun, dass diese Anzahl nicht ausreichend ist, um W korrekt zu akzeptieren.

Die Folge $x_1, \ldots, x_n$ legt eine Zahl in der Menge $M = \{0, \ldots, 2^n - 1\}$ fest. Jede Teilmenge von M kann durch eine Folge $y_0 \cdots y_r$ spezifiziert werden. Es gibt 2^{2^n} Teilmengen von M. Für $n \geq 12s$ übersteigt diese Zahl die Anzahl möglicher Relationen. Daher existieren zwei Teilmengen $M_1, M_2 \subseteq M$ mit $M_1 \neq M_2$, welchen die gleiche Relation C (wie oben definiert) zugeordnet ist. Wir wählen eine Zahl $\ell \in (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$ und $x_1, \ldots, x_n$ so, dass sie ℓ festlegen. Dann kombinieren wir diese Teile mit Zeichenketten, die M_1 und M_2 kodieren. Nun akzeptiert R entweder beide Eingabe oder keine,

was der Definition von W widerspricht. Wir schließen, dass kein Rebound-Automat, der W akzeptiert, existieren kann. □

Rebound-Automaten akzeptieren Sprachen wie

$$P = \{w\$w^R \mid w \in \{0,1\}^*\},$$

wobei w^R das Spiegelbild von w ist [72]. Durch Betrachtung der möglichen Konfigurationen, die ein Zählerautomat erreichen kann, wenn er das \$ liest, ist es klar, dass kein deterministischer Einweg-Zählerautomat mit einem Zähler P akzeptieren kann. Diese Aussage folgt auch aus der allgemeineren Tatsache, dass Einweg-Zählerautomaten (mit mehreren Zählern) exponentielle Zeit benötigen, um P zu akzeptieren [13] und Einweg-Automaten mit einem Zähler in linearer Zeit arbeiten.

Aus den beiden vorigen Lemmata und dieser Tatsache kann der folgende Satz geschlossen werden.

Satz 17 *Die Klasse der durch nichtdeterministische Rebound-Automaten akzeptierten Sprachen ist unvergleichbar mit der Klasse der durch deterministische Einweg-Automaten mit einem Zähler akzeptierten Sprachen.*

Wir erhalten somit auch eine positive Antwort auf die Frage in [34], ob es eine Sprache gibt, die von einem nichtdeterministischen Einweg-Zählerautomaten akzeptiert wird, aber von keinem nichtdeterministischen Rebound-Automaten (dort wird der Begriff „on-line“ für die Einweg-Beschränkung benutzt).

Da die Sprache W durch einen deterministischen Einweg-Zählerautomaten akzeptiert wird, der nur eine Umkehrung auf seinem Zähler ausführt, handelt es sich um eine deterministisch linear kontextfreie Sprache. Wir sind daher ebenfalls in der Lage, das Resultat aus [34] zu verstärken, dass eine kontextfreie Sprache existiert, die von keinem Rebound-Automaten akzeptiert wird (die trennende Sprache aus [34] ist linear, aber nicht deterministisch kontextfrei).

Satz 18 *Es existiert eine deterministisch linear kontextfreie Sprache, die von keinem nichtdeterministischen Rebound-Automaten akzeptiert wird.*

In Kombination mit der Simulation von deterministischen Rebound-Automaten durch deterministische Zweiweg-Zählerautomaten [52] erhalten wir auch eines der Hauptresultate aus [56].

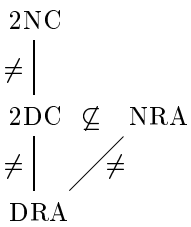
Korollar 10 ([56]) *Die Klasse der durch deterministische Rebound-Automaten akzeptierten Sprachen ist echt in der Klasse der durch deterministische Zweiweg-Automaten mit einem Zähler akzeptierten Sprachen enthalten.*

Rebound-Automaten sind durch Hinzufügen einer endlichen Anzahl von Markierungen oder „Steinchen“ erweitert worden, die im Verlauf der Rechnung auf Zellen abgelegt und später erkannt, wieder entfernt und neu verteilt werden können. Es ist bekannt, dass k Markierungen in der Lage sind, $2k$ Köpfe eines zweidimensionalen Automaten zu simulieren [72]. Wir bemerken, dass die Trennung von Zählerautomaten und Rebound-Automaten sich auf Automaten mit zwei Köpfen überträgt, da ein Kopf einen durch die Eingabelänge beschränkten Zähler simulieren kann. Die führt uns auf einen alternativen Beweis von Theorem 2.2 aus [34].

Korollar 11 ([34]) *Es existiert eine Sprache, die durch einen deterministischen Rebound-Automaten mit einer Markierung akzeptiert werden kann, aber durch keinen nichtdeterministischen Rebound-Automaten ohne Markierungen.*

Die Trennung der deterministischen Modell folgt auch aus dem Hauptresultat von [11].

Einige Beziehungen zwischen den in diesem Kapitel diskutierten Sprachklassen sind in dem folgenden Diagramm dargestellt, wobei 2NC und 2DC die von deterministischen und nichtdeterministischen Zweiweg-Automaten mit einem Zähler akzeptierten Sprachen bezeichnen. Die Klassen NRA und DRA stehen entsprechend für deterministische und nichtdeterministische Rebound-Automaten. Eine Kante zwischen Klasse X oberhalb von Klasse Y bedeutet $Y \subseteq X$. Die Trennung von 2NC und 2DC wurde von Chrobak bewiesen [8].



Kapitel 9

Abschließende Bemerkungen und Ausblick

Wir haben in dieser Arbeit Ergebnisse vorgestellt, die einerseits die zeiteffiziente gegenseitige Simulation von mächtigen Datenstrukturen wie Kellern oder Queues, und andererseits den Vergleich von Modellen mit logarithmischem Speicherplatz betrafen. An der Nahtstelle dieser beiden Untersuchungsgebiete befindet sich Kapitel 6, in dem ein Modell der letzteren Klasse im Hinblick auf seine Zeiteffizienz untersucht wurde. Im Folgenden wollen wir aus der Vielzahl der möglichen Fragestellungen für weitere Untersuchungen einige herausgreifen, die besonders interessant erscheinen.

Im Bereich der zeiteffizienten Simulationen (Kapitel 2–5) bestehen noch zahlreiche Lücken zwischen unteren und oberen Schranken, wie aus der Übersicht hervorgeht. Beispielsweise konnte die obere Schranke für die deterministische Simulation eines umkehrbeschränkten Kellers und die nichtdeterministische Simulation eines allgemeinen Kellers durch eine Queue jeweils auf $O(n^{3/2})$ gesenkt werden, die bekannte untere Schranke aber ist $\Omega(n^{4/3}/\log n)$. Es ist zu vermuten, dass eine Verfeinerung der Technik die untere Schranke heben kann. Besser ist die Situation bei der nichtdeterministischen Simulation einer Deque durch zwei Keller. In Kapitel 5 konnten wir hier eine Lösung in linearer Zeit erreichen. Offen blieb allerdings, ob die Simulation in Realzeit möglich ist. Ein ähnliches Problem stellt sich bei der deterministischen Nachbildung einer Queue durch Keller. Für eine Simulation in linearer Zeit sind zwei Keller ausreichend, während die bekannte Simulation in Realzeit mindestens 6 Keller erfordert [26]. Immerhin zeigt das Ergebnis aus [36], dass drei Keller eine absolute untere Schranke bilden. Auch die Möglichkeiten zur Realzeitsimulation durch Varianten der Deque sind noch weitgehend unerforscht.

In Kapitel 6 konnten wir zeigen, wie der Aufwand zur Simulation von Zei-

gern, die in zwei Richtungen auf einer linearen Liste bewegt werden können, durch solche ohne die Möglichkeit zu Rückwärtsbewegungen reduziert werden kann. Auch eine asymptotisch passende untere Schranke (im Bezug auf die erforderliche Zahl von Hilfsvariablen) konnte gezeigt werden. Eine andere Interpretation dieser Fragestellung ist der Vergleich von logarithmisch platzbeschränkten off-line Turingmaschinen mit mehreren Einweg-Eingabeköpfen, die jeweils auf die Position eines anderen Kopfes „springen“ können, mit einem Modell, dass keine Einschränkung der Bewegungsrichtung macht.

Es sollte beachtet werden, dass eine entsprechende untere Schranke nicht notwendigerweise für *rekursive* Read-Only Programme gilt, jedenfalls nicht für die Aufgabe, auf der unser Beweis der unteren Schranke beruhte. Durch eine rekursive Funktion kann eine lineare Liste auf einfache Art und Weise gespiegelt werden. Daher ist eine positive Antwort auf die durch Neil Jones in [38] aufgeworfene Frage möglich, ob es eine Lösung des Problems der Suche nach einer Zeichenkette in einem Text in linearer Zeit durch ein rekursives Read-Only Programm gibt. Möglicherweise kann dies sogar durch eine allgemeine Simulation von Zweiweg-Zeigern gezeigt werden.

Eine weitere Schwäche der bisher erzielten Ergebnisse ist es, dass die untere Schranke auf der Unmöglichkeit beruht, eine bestimmte *Ausgabe* in einer gegebene Zeitschranke zu erzeugen. Es scheint, dass eine entsprechende Schranke auch für Erkennungsprobleme gelten sollte. Ein Problem, das sich hier auf Grund unseres Beweises anbietet, ist die Erkennung von Palindromen über $\{0, 1\}$.

Kapitel 7 zeigte die Äquivalenz von zwei Modellen auf strikt beschränkten Eingaben. Eine Technik zur Verallgemeinerung von Simulationen dieser Art auf (generelle) beschränkte Sprachen wurde auf diese und andere Simulationen angewendet. Somit konnte ein Problem der Zerlegung der Eingabe von der eigentlichen Simulation des internen Speichers für eine umfassende Klasse von Berechenbarkeitsmodellen getrennt werden.

Die Äquivalenz von Köpfen und Zählern hat zur Folge, dass auch das formal zwischen diesen beiden Speichermedien liegende Konzept der „einfachen“ Köpfe (simple heads) zu einem für beschränkte Sprachen äquivalenten Modell führt. Einfache Köpfe können nur zwischen Endmarkern und Eingabesymbolen unterscheiden, nicht aber zwischen verschiedenen Typen von Eingabezeichen. Einfache Köpfe lassen sich auch als Zähler mit der Möglichkeit zum Vergleich mit der Eingabelänge (plus eins) interpretieren. Es bleibt offen, ob eine analoge Äquivalenz zwischen zwei oder allen drei Varianten für allgemeine Sprachen gilt, oder ob entsprechend der Vermutung aus [52, 53] einfache Köpfe stärker als Zähler sind.[1] Schon das Problem, eine Sprache

[1] Die n-beschränkten Zähler aus [52, 71] sind in der Lage, von 0 bis n zu zählen und

zu finden, die als Kandidat für die Trennung eines einfachen Kopfes von einem Zähler dienen könnte, scheint schwierig zu sein. Zur Erkennung von $L_4 = \{ww \mid w \in \{0,1\}^*\}$ wurden in [71], die Möglichkeiten eines einfachen Kopfes ausgeschöpft. Dies ist aber nicht notwendig. Ein Zählerautomat kann das Gleiche leisten, indem er korrespondierende Positionen der Eingabe auf dem Zähler berechnet.

Erfolgversprechend erscheint es, die Simulation von Köpfen durch Zähler auf Köpfe mit der Möglichkeit zum Positionsvergleich (sensing heads) zu erweitern. Dies soll in einer überarbeiteten Fassung des Kapitels erfolgen.

In Kapitel 8 konnte nachgewiesen werden, dass Nichtdeterminismus die Mächtigkeit von endlichen Rebound-Automaten erhöht. Die verwendete trennende Sprache, wie auch die in dem zweiten Hauptresultat des Kapitels und die in einer früheren Veröffentlichung angegebene [56] sind weit davon entfernt, in natürlicher Weise die Beschränkung eines Rebound-Automaten bei seinem Zugriff auf die Eingabe zu erfassen. Ein typischeres Beispiel, das Zählerautomaten und Rebound-Automaten zu trennen scheint, ist die Sprache

$$L = \{w \mid w \in \{0,1\}^*, |w|_0 = |w|_1\}$$

(hier bezeichnet $|w|_x$ die Anzahl der Vorkommen des Symbols x in w). Ein deterministischer Automat mit einem Zähler kann diese Sprache in offensichtlicher Weise akzeptieren. Sugata, Umeo und Morita [72] vermuten, dass L nicht durch einen Rebound-Automaten akzeptiert werden kann, aber es ist kein Beweis dieser Vermutung bekannt.

In ähnlicher Weise wie zweidimensionale endliche Automaten sind deterministische und nichtdeterministische mit einer zusätzlichen Markierung getrennt worden [33]. Hier besteht die trennende Menge aus solchen Eingaben, die sich in identische obere und untere Hälfte aufteilen lassen (tatsächlich sind auch nicht-quadratische Eingabe zugelassen, aber der Beweis kann auch mit quadratischen Eingabe gerader Seitenlänge durchgeführt werden). Die Möglichkeit, nicht-leere Symbole an beliebige Stellen der Eingabe zu platzieren, scheint hier wesentlich zu sein, aber möglicherweise können deterministische und nichtdeterministische Rebound-Automaten mit Markierungen ebenfalls getrennt werden.

Eine weitere offene Frage ist es, ob eindimensionale nichtdeterministische Zweiweg-Automaten mit einem Zähler in der Lage sind, nichtdeterministische Rebound-Automaten zu simulieren. Die für die deterministischen Varianten

können auf Gleichheit mit diesen Werten getestet werden. Es ist leicht zu sehen, dass sie die gleiche Mächtigkeit wie einfache Köpfe besitzen. Die Klasse deterministischer Zweiweg-Automaten mit k solche Zählern wird in [52, 71] mit $C(k)$ bezeichnet, während Automaten mit k unbeschränkten Zählern die Klasse $D(k)$ bilden.

in [52] angegebene Technik lässt sich nicht übertragen.

Literaturverzeichnis

[1] K. Ayers. Deque automata and a subfamily of context-sensitive languages which contains all semilinear bounded languages. *Theoretical Computer Science*, 40:163–174, 1985.

[2] J. M. Barzdin'. Complexity of recognition of symmetry on Turing machines. *Problemy Kibernet.*, 15:245–248, 1965. Russisch, MR 36#1326.

[3] A. M. Ben-Amram, Z. Galil. On pointers versus addresses. *Journal of the Association for Computing Machinery*, 39:617–648, 1992.

[4] M. Blum, C. Hewitt. Automata on a 2-dimensional tape. In *Proceedings of the 8th Annual Symposium on Switching and Automata Theory, Austin, 1967*, S. 155–160, 1967.

[5] R. V. Book, S. A. Greibach. Quasi-realtime languages. *Mathematical Systems Theory*, 4:97–111, 1970.

[6] F. J. Brandenburg. Multiple equality sets and Post machines. *Journal of Computer and System Sciences*, 21:292–316, 1980.

[7] F. J. Brandenburg. A note on: 'Deque automata and a subfamily of context-sensitive languages which contains all semilinear bounded languages' (by K. Ayers). *Theoretical Computer Science*, 52(3):341–342, 1987.

[8] M. Chrobak. Variations on the technique of Duris and Galil. *Journal of Computer and System Sciences*, 30:77–85, 1985.

[9] T.-R. Chuang, B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, S. 289–298, 1993.

[10] M. Dietzfelbinger, M. Hühne. Matching upper and lower bounds for simulations of several tapes on one multidimensional tape. In P. S.

Thiagarajan (Hrsg.), *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science (FST& TCS)*, Band 880 von Lecture Notes in Computer Science, S. 24–35, Berlin-Heidelberg-New York, 1994. Springer.

[11] P. Ďuriš, Z. Galil. Fooling a two way automaton or one pushdown store is better than one counter for two way machines. *Theoretical Computer Science*, 21:39–53, 1982.

[12] R. A. Finkel, J. L. Bentley. Quad trees, a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[13] P. C. Fischer, A. R. Meyer, A. L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 2:265–283, 1968.

[14] P. C. Fischer, A. R. Meyer, A. L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the Association for Computing Machinery*, 19:590–607, 1972.

[15] Z. Galil, R. Kannan, E. Szemeredi. On nontrivial separators for k-page graphs and simulations by nondeterministic one-tape Turing machines. In *Proceedings of the 18th ACM Symposium on Theory of Computing (STOC), Berkely, California*, S. 39–49, 1986.

[16] Z. Galil, R. Kannan, E. Szemerédi. On nontrivial separators for k-page graphs and simulations by nondeterministic one-tape Turing machines. *Journal of Computer and System Sciences*, 38:134–149, 1989.

[17] Z. Galil, J. Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.

[18] V. Geffert. Nondeterministic computations in sublogarithmic space and space constructability. *SIAM Journal on Computing*, 20:484–498, 1991.

[19] V. Geffert. Tally versions of the Savitch and Immerman-Szelepcsényi Theorems for sublogarithmic space. *SIAM Journal on Computing*, 22:102–113, 1993.

[20] D. Y. Grigor'ev. Imbedding theorems for Turing machines of different dimensions and Kolmogorov's algorithms. *Dokl. Akad. Nauk SSSR*, 234:15–18, 1977. Russisch, Übersetzung in Soviet Math. Dokl., 18:588–592, 1977.

[21] D. Y. Grigor'ev. Time complexity of multidimensional Turing machines. *Zapiski Nauchnykh Seminarov Leningradskogo Otdelniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR*, 88:47–55, 1979. Russisch, Übersetzung in J. Soviet Mathematics, 20:2290–2295, 1982.

[22] J. Hartmanis, R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[23] F. C. Hennie. One-tape, off-line Turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[24] F. C. Hennie. On-line Turing machine computations. *IEEE Transactions on Electronic Computers*, EC-15:35–44, 1966.

[25] F. C. Hennie, R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the Association for Computing Machinery*, 13:533–546, 1966.

[26] R. Hood, R. Melville. Real time queue operations in pure Lisp. *Information Processing Letters*, 13:50–54, 1981.

[27] J. E. Hopcroft, J. D. Ullman. An approach to a unified theory of automata. *Bell System Technical Journal*, 46:1793–1829, 1967.

[28] M. Hühne. On the power of several queues. *Theoretical Computer Science*, 113:75–91, 1993.

[29] M. Hühne. *Concrete Complexity Theory: Studies of the Impact of the Storage Device on the Efficiency of Computations.* PhD thesis, Univ. Dortmund, 1996.

[30] O. H. Ibarra, T. Jiang, N. Tran, H. Wang. On the equivalence of two-way pushdown automata and counter machines over bounded languages. *International Journal of Foundations of Computer Science*, 4:135–146, 1993.

[31] O. H. Ibarra, T. Jiang, N. Tran, H. Wang. On the equivalence of two-way pushdown automata and counter machines over bounded languages. In P. Enjalbert, A. Finkel, K. W. Wagner (Hrsg.), *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, Band 665 von Lecture Notes in Computer Science, S. 354–364, 1993.

[32] O. H. Ibarra, S. M. Kim, L. E. Rosier. Some characterizations of multihead finite automata. *Information and Control*, 67:114–125, 1985.

[33] K. Inoue, I. Takanami. A survey of two-dimensional automata theory. *Information Sciences*, 55:99–121, 1991.

[34] K. Inoue, I. Takanami, H. Taniguchi. A note on rebound automata. *Information Sciences*, 26:87–93, 1982.

[35] T. Jiang, J. I. Seiferas, P. M. B. Vitányi. Erratum: "Two heads are better than two tapes". *Journal of the Association for Computing Machinery*, 44:632, 1997.

[36] T. Jiang, J. I. Seiferas, P. M. B. Vitányi. Two heads are better than two tapes. *Journal of the Association for Computing Machinery*, 44:237–256, 1997. Siehe auch [35].

[37] N. D. Jones. *Computability and Complexity — From a Programming Perspective*. MIT Press, Cambridge, Mass., London, England, 1997.

[38] N. D. Jones. Must fast pattern matchers be impure? Unterlagen zum Dagstuhl Seminar 9823 "Programs: Improvements, Complexity, and Meanings", June 8–12, 1998.

[39] D. E. Knuth. *The Art of Computer Programming*, Band 1. Addison-Wesley, Reading Mass., 3rd edition, 1997.

[40] D. E. Knuth, J. H. Morris, V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[41] B. L. Leong, J. I. Seiferas. New real-time simulations of multihead tape units. *Journal of the Association for Computing Machinery*, 28:166–180, 1981.

[42] M. Li. Simulating two pushdown stores by one tape in $O(n^{1.5}\sqrt{\log n})$ time. *Journal of Computer and System Sciences*, 37:101–116, 1988.

[43] M. Li, L. Longpré, P. Vitányi. The power of the queue. *SIAM Journal on Computing*, 21:697–712, 1992.

[44] M. Li, P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, Berlin-Heidelberg-New York, 1993.

[45] M. Li, P. M. B. Vitányi. Tape versus queue and stacks: The lower bounds. *Information and Computation*, 78:56–85, 1988.

[46] M. C. Loui. Simulations among multidimensional Turing machines. *Theoretical Computer Science*, 21:145–161, 1982.

[47] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.

[48] F. Meyer auf der Heide. On genuinely time bounded computations. In B. Monien, R. Cori (Hrsg.), *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, Band 349 von Lecture Notes in Computer Science, S. 1–16, 1989.

[49] B. Monien. About the derivation languages of grammars and machines. In M. Steinby (Hrsg.), *Proceedings of the 4th International Colloquium on Automata, Languages and Programming (ICALP), Turku, 1977*, Band 52 von Lecture Notes in Computer Science, S. 337–351, 1977.

[50] B. Monien. Two-way multihead automata over a one-letter alphabet. *R.A.I.R.O. — Informatique Théorique et Applications*, 14:67–82, 1980.

[51] B. Monien. Deterministic two-way one-head pushdown automata are very powerful. *Information Processing Letters*, 18:239–242, 1984.

[52] K. Morita, K. Sugata, H. Umeo. Computation complexity of n-bounded counter automaton and multidimensional rebound automaton. *Systems • Computers • Controls*, 8:80–87, 1977. Übersetzung von Denshi Tsushin Gakkai Ronbunshi (IECE of Japan Trans.) 60-D:283–290, 1977 (Japanisch).

[53] K. Morita, K. Sugata, H. Umeo. Computational complexity of n-bounded counter automaton and multi-dimensional rebound automaton. *IECE of Japan Trans.*, 60-E:226–227, 1977. Zusammenfassung von [52].

[54] W. J. Paul. On-line simulation of $k + 1$ tapes by k tapes requires nonlinear time. *Information and Control*, 53:1–8, 1982.

[55] H. Petersen. Two-way one-counter automata accepting bounded languages. *SIGACT News*, 25(3):102–105, 1994.

[56] H. Petersen. Fooling rebound automata. In M. Kutyłowski, L. Pacholski, T. Wierzbicki (Hrsg.), *Proceedings of the 24th Symposium on Mathematical Foundations of Computer Science (MFCS), Szklarska Poreba, 1999*, Band 1672 von Lecture Notes in Computer Science, S. 241–250, Berlin-Heidelberg-New York, 1999. Springer.

[57] H. Petersen. Stacks versus deques. In J. Wang (Hrsg.), *Proceedings of the 7th Annual International Computing and Combinatorics Conference*

(COCOON), Guilin, 2001, Band 2108 von Lecture Notes in Computer Science, S. 218–227, Berlin-Heidelberg-New York, 2001. Springer.

[58] H. Petersen, J. M. Robson. Efficient simulations by queue machines. In K. G. Larsen, S. Skyum, G. Winskel (Hrsg.), *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP), Aalborg, 1998*, Band 1443 von Lecture Notes in Computer Science, S. 884–895, 1998.

[59] N. Pippenger. Pure versus impure Lisp. *ACM Transactions on Programming Languages and Systems*, 19:223–238, 1997.

[60] N. Pippenger, M. J. Fischer. Relations among complexity measures. *Journal of the Association for Computing Machinery*, 26:361–381, 1979.

[61] E. L. Post. Formal reductions of the classical combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.

[62] M. O. Rabin, D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[63] R. W. Ritchie, F. N. Springsteel. Language recognition by marking automata. *Information and Control*, 20:313–330, 1972.

[64] B. Rosenberg. Simulating a stack by queues. In *Proceedings of the XIX Latinamerican Conference on Computer Science*, Band 1, S. 3–13, 1993.

[65] B. Rosenberg. Fast nondeterministic recognition of context-free languages using two queues. *Information Processing Letters*, 67:91–93, 1998.

[66] A. Rosenfeld. *Picture Languages.* Academic Press, New York, 1979.

[67] M. Sakamoto, K. Inoue, I. Takanami. A two-way nondeterministic one-counter language not accepted by nondeterministic rebound automata. *IECE of Japan Trans.*, 73-E:879–881, 1990.

[68] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3:198–200, 1959.

[69] H.-J. Stoß. k-Band Simulation von k-Kopf Turingmaschinen. (k-tape simulation of k-head Turing machines). *Computing*, 6:309–317, 1970.

[70] H.-J. Stoß. Zwei-Band Simulation von Turingmaschinen. (A two-tape simulation of Turing machines). *Computing*, 7:222–235, 1971.

[71] K. Sugata, K. Morita, H. Umeo. The language accepted by an n-bounded multicounter automaton and its computing ability. *Systems • Computers • Controls*, 8:71–79, 1977. Übersetzung von Denshi Tsushin Gakkai Ronbunshi (IECE of Japan Trans.) 60-D:275–282, 1977 (Japanisch).

[72] K. Sugata, H. Umeo, K. Morita. The language accepted by a rebound automaton and its computing ability. *Electronics and Communications in Japan*, 60-A:11–18, 1977.

[73] A. Szepietowski. *Turing Machines with Sublogarithmic Space*. Band 843 von Lecture Notes in Computer Science. Springer, Berlin-Heidelberg-New York, 1994.

[74] R. Vollmar. Über einen Automaten mit Pufferspeicherung (On an automaton with buffer-tape). *Computing*, 5:57–70, 1970.

[75] K. Wagner, G. Wechsung. *Computational Complexity*. Mathematics and its Applications. D. Reidel Publishing Company, Dordrecht, 1986.

[76] L. Zhang, T. Okozaki, K. Inoue, A. Ito, Y. Wang. A note on probabilistic rebound automata. *IEICE Trans. Inf. & Syst.*, E81-D:1045–1052, 1998.