Automaten und Umwelten

L. Budach und O. Wilhelm

Mai 2002

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Historische Eckdaten

Informatik in einem sehr allgemeinen Sinn beginnt lange, bevor Dokumentationen dazu erfolgt sind. Das Zählen von Dingen, das Merken von Zahlen mit Hilfe von Kerben auf Stöcken, die Übermittlung von Nachrichten durch Trommeln sind Beispiele für Informationsverarbeitung.

Erste Quellen

Dokumentierte Geschichte der Informatik beginnt mit **Abu Ja'far Mohamed ibn Musa al-Kharizmi**, einem iranischen Gelehrten. Er schrieb die ältesten uns bekannten systematischen Lehrbücher algorithmischen und mathematischen Inhalts. Das um **820** entstandene **Rechenbuch** wurde rund 400 Jahre später ins Lateinische übersetzt. Sein lateinischer Titel „Algorithmi de numero Indorum" und der arabische Titel seines zweiten großen Werkes „Kitab al-muhtasar fi hisab al-gabr w'almuqabalah" lieferten die Wurzeln der Wörter Algorithmus und Algebra.

Um **1500** gibt es in Europa eine deutlich zu beobachtende Auseinandersetzung zwischen den Anhängern des Rechnens auf dem Rechenbrett, dem Abacus, und den Verfechtern des Rechnens mit den von al-Khwarizmi überlieferten arabisch-indischen Zahlen. Die Parteigänger werden **Abacisten** und **Algorithmiker** genannt.

Die technische Revolution beginnt

Um **1600** beginnt eine markante Entwicklung in der Mathematik, die algorithmischen Denk- und Verfahrensweisen den Weg bereitet. Zu den herausragenden Gelehrten gehören:

- **François Viète** (auch Vieta; **1540–1603**; Frankreich): Er benutzt Variablenbezeichner und algebraische Rechenmethoden.
- **John Napier of Merchiston** (auch Neper; **1550–1617**; Schottland): Von ihm stammt eine in 30 Jahren erarbeitete Tafel Briggscher Lofarithmen (Logarithmen zur Basis 10).
- **Simon Stevin** (**1548–1620**): Diesem holländischen Mathematiker und

Ingenieur verdanken wir, dass sich in Europa die Dezimalbrüche durchgesetzt haben.

Die von diesen Gelehrten erarbeiteten Mittel und Methoden naturwissenschaftlicher Arbeit werden von uns alltäglich benutzt oder sind bereits überholt, waren aber zu ihrer Zeit wertvolle Errungenschaften von großer Bedeutung für den wissenschaftlichen Fortschritt.

Bemerkenswert ist auch, dass schon damals funktionsfähige Rechenmaschinen entwickelt wurden:

1623 : Wilhelm Schickhardt (**1592–1635**) stellt eine Rechenuhr vor, die addieren kann. Schickhardt (auch Schickard) war Mathematiker und Orientalist.

Erste Computer werden gebaut

1642: Der französische Mathematiker, Physiker und Philosoph **Blaise Pascal** (**1623–1662**) baut ein Gerät, das addieren und subtrahieren kann. Er nannte es Pascaline und wollte damit seinem Vater, der Steuereinnehmer war, die Arbeit erleichtern.

1672 : Gottfried Wilhelm Leibniz (**1646–1716**) entwickelt eine Maschine mit Staffelwalzen, die auch multiplizieren und dividieren kann.

Dem schwedischen Naturforscher (Biologen) und Arzt **Carl von Linnè** (**1707–1778**) verdanken wir die Idee, Informationen in Form von Baumstrukturen zu ordnen. Die von ihm eingeführte und seither international benutzte Klassifizierung der Pflanzen, aber auch der Kristalle unter Benutzung von Bestimmungstabellen beruht auf diesem für die Informatik so grundlegendem Prinzip.

Den Namen des französischen Barons **Marie Riche de Prony** (**1755–1839**) erwähnen wir an dieser Stelle nicht so sehr wegen seiner Meriten als Gelehrter. Er tat sich in der Wissenschaftsgeschichte als genialer Manager hervor:

Um **1800** bekam er vom Convent den Auftrag, in kurzer Frist neue Logarithmentafeln zu erarbeiten. (Politischer Hintergrund dieser Geschichte war es, das Ansehen des nachrevolutionären Frankreich zu stärken.) Um diesen wegen der Kurzfristigkeit schwierigen Auftrag zu realisieren, orientierte er sich an der Produktionsweise der damals entstehenden Manufakturen. Deren Beschreibung fand er in **Adam Smith**' Buch „An Enquiry on the Wealth of Nations“. De Prony gründete ein „Bureau des Cadastres“, in dem es drei Abteilungen gab. In der ersten waren acht der besten Mathematiker Frankreichs beschäftigt, z.B. auch der berühmte Legendre. Ihre Aufgabe war es, die geeigneten Algorithmen zu ermitteln. In einer weiteren Abteilung arbeiteten 20 Personen, deren Aufgabe es war, die Rechenverfahren derart zu normieren, dass in einer weiteren Abteilung die Beschäftigten — und dort waren es 80 — nur addieren und subtrahieren mussten. Diese „Rechenknechte“ sollen dabei ganz unglaubliche Fertigkeiten entwickelt haben, sie stellten gewissermaßen 80 parallel arbeitende Prozessoren für einfachste, aber ungeheuer schnell ablaufende Prozeduren dar. Eine solche Modularisierung von Prozessen ist ein wichtiges Merkmal moderner Informationsverarbeitung.

Charles Babbage (**1792–1871**) war derjenige, der zum ersten Mal programmierbare Rechenmaschinen konzipierte und entwickelte. Er war englischer Mathematiker, Techniker und Ökonom und baute zum Beispiel vier Jahre vor Helmholtz einen Augenspiegel. Im Jahr 1823 stellte er eine "difference engine" zur Lösung arithmetischer Gleichungen, 1835 eine "analytical engine" vor. Obzwar der erste der beiden Rechner zu Lebzeiten von Babbage nicht funktionsfähig wurde und die praktische Konstruktion des zweiten gar nicht in Angriff genommen wurde, waren sie aber so gut durchdacht, dass nach den Originalplänen im Jahre 1991 eine "difference engine no. 2" in London gebaut werden konnte. Für die Eingabe von Informationen waren Lochkarten vorgesehen, eine bekanntlich noch vor wenigen Jahren durchgängig genutzte Technik.

George Boole(**1815–1864**, England, Mathematiker) hatte die glänzende Idee, einer Aussage „nur“ zwei Wahrheitswerte, nämlich wahr und falsch zuzuordnen, unsere Gedankenwelt gleichsam zu digitalisieren. Damit begründete er, lange bevor die technischen Möglichkeiten bereit standen, die theoretischen Grundlagen für das Innenleben heutiger Computer.

David Hilbert(**1862–1943**) hielt auf dem zweiten internationalen Mathematikerkongress **1900** einen berühmten Vortrag, in dem er 23 ungelöste mathematische Probleme formulierte. Damit gab er den Anstoß für folgenreiche Entwicklungen in verschiedenen Bereichen der Mathematik.

Der Algorithmusbegriff wird präzisiert

Das zehnte Problem war diophantischen Gleichungen gewidmet. Das sind arithmetische Gleichungen, deren Variablen ganzzahlige Koeffizienten haben und deren Lösungen im Bereich der ganzen Zahlen gesucht werden. (Diophant war ein griechischer Mathematiker des Altertums.) Hilbert stellte die Frage nach einem konstruktiven Verfahren, das entscheidet, ob eine beliebige diophantische Gleichung eine Lösung hat oder nicht. Die Lösung des Hilbertschen Problems, und zwar eine negative, wurde erst 1970 von dem russischen Mathematiker Matjasevitch gefunden. In unserem Zusammenhang ist wichtig, dass diese Problemstellung Anlass wurde für eine intensive, internationale Forschung zu der Frage, was man unter einem konstruktiven Verfahren zu verstehen hat. Und in deren Ergebnis gelangte man zu einer endgültigen Präzisierung des Algorithmenbegriffs. Mitte der dreißiger Jahre des zwanzigsten Jahrhunderts erschienen etwa gleichzeitig vier Arbeiten von verschiedenen Autoren, allesamt in der Grundlagenforschung tätige Mathematiker, die – von ganz unterschiedliche Modellen ausgehend – zu Begriffsbildungen kamen, die sich als gleichwertig erwiesen:

- **Alonzo Church** (geb. **1903**, USA):„A note on the Entscheidungsproblem“ (1936)
- **Stephen Cole Kleene**(geb. **1909**, USA): „General recursive functions of natural numbers“ (1936)
- **Emil Leon Post** (**1897–1954**, USA): „Finite combinatory processes“ (1936)

- **Alan Mathison Turing** (**1912–1954**, England): „On computable numbers with an application to the Entscheidungsproblem“ (1937)

Weitere Algorithmenbegriffe, wie z.B. der Markowsche folgten:

Andrej Andrejewitsch Markow (**1903–1979**; russischer Mathematiker): „Über die Darstellung rekursiver Funktionen“ (1949, russisch)

Das weltweite wissenschaftliche Interesse an der Problemstellung wird durch diese internationale Beteiligung belegt.

Die überraschende Gleichwertigkeit der auf völlig unterschiedliche Weise gewonnenen Resultate veranlasste Church zu der nach ihm benannten These von der Gleichwertigkeit konstruktiver Algorithmen.

Das Computerzeitalter beginnt

Mit **Konrad Zuse** (**1910–1995**) beginnt das Zeitalter der modernen Rechner. 1937 hatte er die Z1 fertiggestellt, 1941 die programmierbare Z3. Diese Rechner waren mit mechanischen Relais' ausgestattet.

Während des zweiten Weltkrieges arbeitete man unter dem Druck der politischen Ereignisse in den USA und in England ebenfalls fieberhaft am Bau von Rechnern, die auch bereits zur Lösung kriegswichtiger Probleme eingesetzt wurden. **1946** wurde in den USA die **ENIAC** (Electronic Numerical Integrator and Calculator) vorgestellt. Sie arbeitete mit Elektronenröhren und hatte etwa die Größe eines Tennisplatzes. Man benutzte damals noch – im Gegensatz zur Z1, die schon digital arbeitete – dezimal dargestellte Zahlen.

John von Neumann (**1903–1957**), ein amerikanischer Mathematiker österreichisch-ungarischer Herkunft, der auf vielen Gebieten der Mathematik unseres Jahrhunderts bahnbrechende Arbeiten veröffentlichte, beeinflusste auch die Prinzipien, nach denen Rechner gebaut werden, in so bestimmender Weise, dass man von der „von-Neumannschen Rechnerarchitektur“ spricht. Allerdings wurde das von ihm eingeführte Arbeitsregime eines modernen Rechners schon von Turing theoretisch begründet. Es besteht darin, einem Rechner neben den Daten für einen Algorithmus auch den Algorithmus selbst in Gestalt eines Programms zu übergeben. Das war ein wichtiger Gedanke, um die flexible Einsetzbarkeit von Rechnern zu garantieren.

Eine wichtige physikalische Entdeckung

Nächste wichtige Stufe in der historischen Entwicklung der Informatik war die Entdeckung des Transistoreffekts durch **John Bardeen** (geb.**1908**), **Walter Hausser Brittain** (geb.**1902**) und **William Shokley** (geb.**1910**), die dafür den Nobelpreis für Physik erhielten.

Von nun an setzt eine rasante Entwicklung ein, die zu beschreiben unseren Rahmen sprengen würde. Unser Anliegen wird es sein, die von Turing, Kleene, Post und anderen begründeten Ideen über den Algorithnusbegriff zu systematisieren mit der Absicht, die Informatik als einen wohl begründeten Bestandteil unseres Wissenschaftssystems darzustellen.

1.2 Exemplarisches zum Algorithmusbegriff

Unser Verhandlungsgegenstand ist die Informatik. Was das eigentlich ist, können wir für Uneingeweihte hier nur ungefähr sagen. Zunächst ist festzustellen, dass in den USA diese Wissenschaftsdisziplin unter dem Namen **Computer Science** entstanden ist. Einer der ganz Großen auf diesem Gebiet, Donald E. Knuth, "definiert" deshalb :

Was ist Informatik

> *"I believe that Computer Science is the study of Algorithms."* [1]

Hartmanis formuliert folgendermaßen:

> *"Computer Science befasst sich mit vom Menschen geschaffenen Objekten. Wir müssen zuerst erdenken oder erbauen, was wir dann untersuchen wollen. Wir müssen die intellektuellen Werkzeuge entwickeln, mit denen wir nicht nur das Existierende erforschen, sondern auch das Mögliche studieren können."*

In Europa hat man den Begriffsinhalt dieser Wissenschaftsdisziplin erweitert und auch einen anderen Namen, eben Informatik gewählt. Die Académie Française hat folgende prägnante Charakterisierung publiziert:

> *„L' Informatique: Science de traitement rationel, notamment par machines automatiques, de l' information consideré, comme le support des connaissances humaines et de communications, dans les domaines techniques, économiques et sociales"* [2]

Das folgende Zitat von Jacob T. Schwartz soll die Abgrenzung zur Mathematik beschreiben:

> *"Das Einfache im Komplexen finden, das Endliche im Unendlichen– das scheint mir keine schlechte Beschreibung der Aufgabe und des Wesens der Mathematik zu sein. Dagegen ist die Informatik wohl so zu charakterisieren, dass sie mit ihren Mitteln Schneisen in die Zone des Komplexen schlägt."*

Algorithmen im Alltag

Die algorithmischen Handlungsmechanismen, die immer wieder als Charakteristikum der Informatik hervorgehoben werden, sind durchaus Grunbestand menschlicher Tätigkeit. Wir wollen das belegen.

In einem Buch des Titels "Wissensspeicher Holztechnik" findet man z.B. die folgende Anweisung zum Schleifen von Holz:

- Das Werkstück ist einzuspannen.

[1] Ich glaube, dass Computer Science das Studium von Algorithmen ist.

[2] Informatik: Wissenschaft von der rationellen Verarbeitung und dem Transport von Infomation als Träger menschliche Wissens– insbesondere durch automatische Maschinen–in Technik, Ökonomie und Gesellschaft.

- Falls nach der Bearbeitung die Holztextur sichtbar sein soll, so ist der Schleifblock parallel zum Faserverlauf zu führen, sonst
  - a) den Schleifblock quer und
  - b) parallel zur Faser führen.
- Die Flächen werden gewässert.
- Trocknen der Flächen
- Wiederhole das Schleifen.
- Zum Abschluss werden die Kanten gebrochen.

Damit ist ein gut strukturiertes Handlungsprogramm geliefert, das eine Wiederholungsanweisung enthält, ein Konstrukt, welches jeder Programmierer als Routine benutzt.

Der folgende Auszug aus einem Kochbuch mag als Beispiel für parallel auszuführende Handlungen dienen:

> Während die Suppe zehn Minuten dünstet, das Fleisch in Würfel schneiden und mit Zwiebeln zusammen anbraten.

Im folgenden Beispiel findet sich eine Form von Nichtdeterminismus:

> Kartoffelbrei oder Salzkartoffeln können als Beilage dienen, aber auch die etwas kräftiger schmeckenden Bratkartoffeln oder Pommes frites.

Zum Abschluss dieser Beispiele für "Algorithmen im Alltag" ein Zitat aus einem Radwanderbuch:

> Ab hier beginnt Golm. Wir radeln auf den Reiherberg zu, folgen dann aber dem Lauf der Hauptstraße, biegen aber schon nach 1km rechts ab

Algorithmen sind also in unserem täglichen Leben etwas höchst Selbstverständliches und haben zunächst gar keinen Zusammenhang zu Rechnern. Wie so oft in der Mathematik und den Naturwissenschaften wird die Sachlage dann komplizierter, wenn wir eine wissenschaftlich brauchbare Konkretisierung des Gegenstandes brauchen. Um das vorzubereiten, noch drei Beispiele mehr formaler Natur, weil sie besser geeignet sind, Wesentliches zu beleuchten.

Es gibt keine Lösung

Im ersten Beispiel betrachten wir ein Schachbrett, auf welchem die diagonal gegenüber liegenden Felder a8 und h1 fehlen (vgl. Abbildung 1.1). Dieses reduzierte Schachbrett soll mit einem Parkett aus Dominosteinen belegt werden. Man gebe einen Algorithmus an, der zu einer Belegung dieser Art führt.

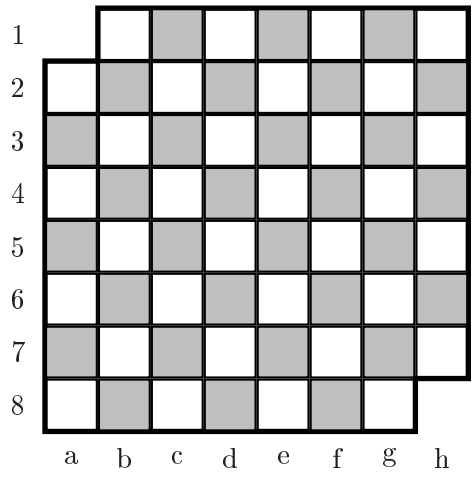


Abbildung 1.1: Gibt es eine Parkettierung mit Dominosteinen?

Es zeigt sich, dass ein solcher Algorithmus gar nicht existieren kann, weil nämlich das Problem nicht lösbar ist. Man braucht also den Versuch einer Parkettierung gar nicht zu beginnen. Warum?

Angenommen, es gäbe eine Lösung. Dann könnte man die Dominosteine derart schwarz und weiß einfärben, dass jede schwarze bzw. weiße Dominohälfte auf einem gleichfarbigen Schachfeld läge. Dann hätte man 31 schwarze und 31 weiße Dominohälften, was aber nicht mit den Anzahlen der schwarzen und weißen Felder des Schachbrettes übereinstimmt.

Das zweite Beispiel ist mathematischer Natur. Wir betrachten die Reste ganzer Zahlen bei der Division durch eine Primzahl p. In den gängigen Programmiersprachen schreibt man $r := a \; mod \; p$, um auszudrücken, dass der Variablen r der Rest bei der Division von a durch p zugewiesen wird. Sei nun $a \in \mathbb{N}$ und $0 < a < p$ [3]. Dazu gibt es stets ein $\hat{a} \in \mathbb{N}$ mit $0 < a < p$ derart, dass das Produkt aus a und $\hat{a}$ den Rest 1 hat.

Zum Beispiel ist

$$2 \cdot 4 \bmod 7 = 1,$$
$$3 \cdot 5 \bmod 7 = 1, \text{ usw.}$$

Man nennt $\hat{a}$ das zu a **arithmetisch Inverse** bezüglich p.

[3] $\mathbb{N}$ bezeichnet die Menge der natürlichen Zahlen

Der Beweis der Existenz von $\hat{a}$ ergibt sich nach dem so genannten Schubkastenprinzip. O.b.d.A. [4] darf $1 < a < p$ angenommen werden, weil der Fall $a = 1$ trivial ist.

Nun ergibt sich, dass die $p - 1$ Zahlen $a \cdot i \bmod p$ mit $1 \leq i \leq p - 1$ alle sowohl positiv als auch paarweise verschieden sind.

Wenn nämlich $a \cdot i \bmod p = 0$ wäre, gäbe es eine natürliche Zahl y mit $a \cdot i = p \cdot y$. p wäre also Teiler entweder von a oder von i. Das ist nicht möglich, weil beide kleiner als p sind.

Dass die Zahlen alle paarweise verschieden sind, ergibt sich ebenfalls indirekt: Wäre $i \neq j$ – o.B.d.A darf $i > j$ angenommen werden – und $a \cdot i \bmod p = a \cdot j \bmod p$, so gäbe es natürliche Zahlen u, v mit

$$a \cdot i = p \cdot u + r$$
$$a \cdot j = p \cdot v + r,$$

woraus sich

$$a \cdot (i - j) = p \cdot (u - v)$$

ergibt.

Offenbar ist $u \neq v$, weil man anderenfalls $i = j$ erhält. Damit ist wiederum p Teiler entweder von a oder von $i - j$ im Widerspruch zur Wahl dieser Zahlen.

Jetzt kann man von dem Schubfachprinzip Gebrauch machen: da in der Menge

$$M_a = \{n \in \mathbb{N} \mid n = a \cdot i;\ i \in \mathbb{N};\ 1 \leq i < p\}$$

genau $p - 1$ natürliche Zahlen zwischen 0 und p vorkommen, muss die 1 dabei sein.

Es liegt also ein lösbares Problem vor. Der angegebene Beweis liefert allerdings kein brauchbares Verfahren, um das Inverse auch effektiv zu berechnen. Wenn p eine große Primzahl ist, kann die Ermittlung der Reste recht aufwendig sein. Wir haben hier also den Fall eines gewiss lösbaren Problems, die Suche nach einem guten Algorithmus ist aber eine neue Frage.

Es gibt einen effizienten Algorithmus

In diesem Beispiel bietet sich ein Verfahren an, das nach Euklid benannt ist und das wir nur exemplarisch beschreiben. Exakte mathematische Begründung und Programmierung, womöglich rekursiv, sind eine interessante Übung.

Sei $p = 53$ und $a = 11$. Schrittweises Dividieren mit Rest ergibt

$$53 = 4 \cdot 11 + 9$$
$$11 = 1 \cdot 9 + 2$$
$$9 = 4 \cdot 2 + 1$$

[4] Ohne Beschränkung der Allgemeinheit

Dieser letzte Rest ist gesetzmäßig. Er kommt immer vor, wie man das Beispiel auch wählt (warum?). Bei dieser Zeile wird abgebrochen und der Rest 1, rückwärts rechnend, linear aus p und a zusammengesetzt:

$$\begin{aligned} 1 &= 9 - 4 \cdot 2 = 9 - 4 \cdot (11 - 1 \cdot 9) = \ldots \\ &= 5 \cdot 53 - 24 \cdot 11 \text{ oder} \\ -24 \cdot 11 &= -5 \cdot 53 + 1 \end{aligned}$$

Addiert man auf beiden Seiten $53 \cdot 11$, ergibt sich

$$29 \cdot 11 = 6 \cdot 53 + 1, \text{ demnach } \hat{a} = 29$$

Tests mit diesem Algorithmus zeigen, dass er sehr effektiv ist.

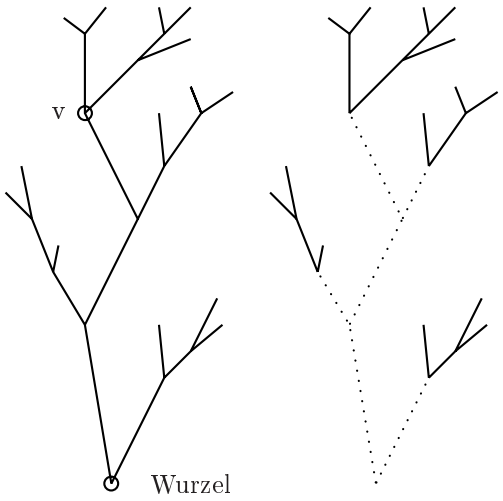


Abbildung 1.2: Wer gewinnt?

Ein Algorithmus existiert gewiss

Das letzte Beispiel steht für den Fall, dass man weiß, es gibt einen Algorithmus, ohne ihn zu kennen. Für das in Abbildung 1.2 angedeutete Zweipersonenspiel gilt die Regel, dass jeder der zwei Spieler abwechselnd in einem Baumgraphen einen Knoten belegt, was zur Folge hat, dass jedes Mal der gesamte darunter liegende Pfad bis zur Wurzel abstirbt. Gewonnen hat, wer den letzten Knoten besetzt.

Es handelt sich um ein so genanntes Nullsummenspiel, und seit den grundlegenden Arbeiten von John von Neumann ist bekannt, dass es für solche Spiele immer eine Gewinnstrategie gibt.

Man kann nun zeigen, dass in unserem Falle der Spieler gewinnt, welcher beginnt, natürlich unter der Voraussetzung, dass er einer Gewinnstrategie folgt.

Das kann man indirekt beweisen. Spieler (A) sei derjenige, welcher beginnt, (B) der andere. Angenommen, der Spieler (B) gewinnt, wenn er einer Gewinnstrategie folgt. Setzt nun Spieler (A) auf die Wurzel des Baumes, so wird Spieler (B) in Erwiderung auf einen Knoten v setzen (vgl. Abb. 1.2). Dieser Knoten und alle darunter liegenden sterben ab. Diese Spielsituation, die nach zwei Zügen eingetreten ist, kann Spieler (A) aber zu Beginn eines neuen Spiels mit dem gleichen Baum auch in nur einem Zug erreichen, indem er gleich auf v setzt. Er ist dann nach diesem Zug in der gleichen Lage wie Spieler (B) nach dem zweiten Zug im ersten Spiel, kann also die Strategie des Spielers (B) benutzen und siegt. Die Voraussetzung, dass Spieler (B) gewinnt, hat zu einem Widerspruch geführt, weil ja nur einer gewinnen kann.

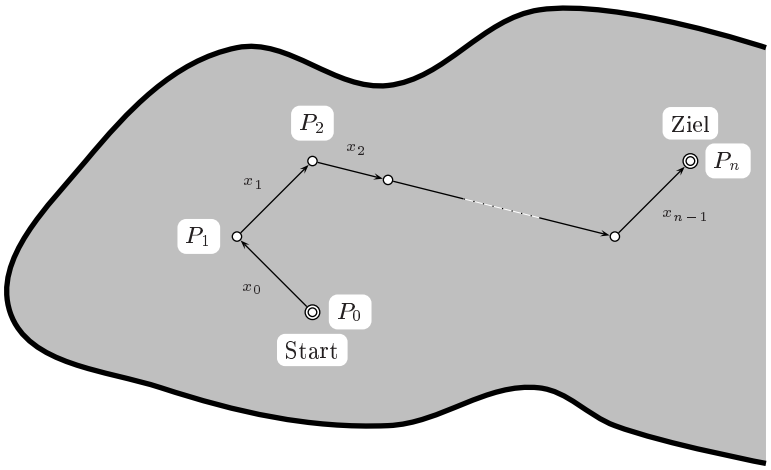


Abbildung 1.3: Ein Lösungsraum

Umwelt und Automat

Die drei Beispiele sind Prototypen so genannter Probleme. Davon ausgehend, wollen wir ein Problemlösungsmodellïnformal beschreiben. Die genaue Ausarbeitung dieses Konzeptes ist wesentlicher Inhalt der nachfolgenden Kapitel. Wenn ein Problem gestellt wird, ist immer eine gewisse Ausgangssituation, ein Startpunkt gegeben. Ferner wird ein Ziel formuliert, das zu erreichen ist, und es werden Möglichkeiten angeboten, die es unter Umständen gestatten, einen Weg vom Start zum Ziel zu finden. Das alles wollen wir unter dem Begriff einer Umwelt zusammenfassen. Sie soll aus einer im allgemeinen unendlichen Menge von Punkten bestehen. In einem Startpunkt hat man einen gewissen Einblick, der es ermöglicht, Entscheidungen zu treffen und Aktionen x_i auszulösen, welche zu neuen Punkten und neuen Einblicken führen und möglicherweise endlich zum Zielpunkt. Die Bestimmung des Weges vom Start zum Ziel, mit anderen Worten die Problemlösung zu finden, wird Aufgabe eines Automaten sein. Dabei wollen wir annehmen, dass ein Automat nur über begrenzte Ressourcen verfügt. Er soll nur eine endliche Zahl von Aktionen auszuführen in der Lage sein und

auch nur eine endliche Zahl von Wahrnehmungen aufnehmen können. Überdies werden wir verlangen, dass ein Automat ein begrenzter Gedächtnis, einen endlichen Speicher besitzt. Das entspricht unseren Möglichkeiten: wir können noch so komplizierte Geräte erdenken und bauen oder Computer mit den komfortabelsten Speichern und vielen schnell arbeitenden Prozessoren konstruieren, sie werden immer mit endlichen Mitteln auskommen müssen. Unbeschränkt große Umwelten mit endlichen Mitteln zu untersuchen, das ist unser Denkprinzip. Dieses Modell wird in den folgenden Kapiteln genau beschrieben und fortlaufend benutzt. Es wird sich als ein sehr brauchbares didaktisches Hilfsmittel erweisen und Wege zu Neuem öffnen.

Aufgaben 1

1. *Beweis der Korrektheit des euklidischen Algorithmus zur Berechnung des arithmetisch Inversen einer natürlichen Zahl bezüglich einer Primzahl, am Beispiel erläutert auf Seite 12*

2. *Welche Strategie muss in dem Nimmspiel auf Seite 13 der beginnende Spieler anwenden, um zu gewinnen?*

3. *In einem würfelförmigen Kasten der Kantenlänge $3d$ sind 27 Äpfel gleicher Größe des Durchmessers d untergebracht. Im innersten Apfel sitzt eine gierige Made. Gelingt es diesem Wurm, sich durch alle Äpfel hindurch zu fressen, wenn sie jeden Apfel nur ein einziges Mal aufsucht und von einem zum nächsten Apfel nur dort hinüber wechselt, wo sie sich berühren? Die Antwort ist zu beweisen!*

Kapitel 2

Automaten und Umwelten

2.1 Endliche deterministische Automaten

2.1.1 Was ist ein Automat?

Informale Beschreibung

Geräte, die man üblicherweise Automaten nennt, erhalten Informationen, welche zu Aktionen bzw. Ausgaben führen. Dabei müssen die Eingaben den Ausgaben in sinnvoller Weise zugeordnet werden. Ein Getränkeautomat hat z.B. zu prüfen, ob eine eingeworfene Münze gültig ist, dem Preis der gewünschten Ware genau entspricht, ob weitere Münzen erforderlich sind oder ob eventuell Wechselgeld auszugeben ist. Das sind vier verschiedene Situationen, die als Zustände bezeichnet werden.

Unter den Zuständen gibt es solche, in denen der Automat stehen bleibt. Er beendet seine Arbeit und gibt eine Marke aus. Das sind seine Endzustände, und die Marken liefern geeignete Informationen, z.B. dass die Arbeit abgeschlossen ist. Bevor eine genaue Definition formuliert wird, fügen wir einige Notationsvereinbarungen für zwei mathematische Begriffe ein:

Funktionen bzw. Abbildungen; Relationen

$$\begin{aligned} f : A &\xrightarrow{\supset} B \\ a &\mapsto f(a) = b \end{aligned}$$

So wird eine **partielle Funktion** von A nach B notiert. Das Attribut "partiell" drückt aus, dass es Elemente $a \in A$ gibt, für welche kein Funktionswert existiert. Die Funktion ist für solche Elemente nicht definiert. In diesem Falle schreiben wir $f(a) = \perp$. Die Menge der Elemente von A, zu denen ein Funktionswert existiert, soll **Definitionsbereich von f** heißen und wird $dom\ f$ abgekürzt [1]:

$$dom\ f = \{a \in A \mid f(a) \neq \perp\}$$

Die Abbildung f von A nach B ist also *partiell*, wenn ihr Definitionsbereich eine echte Teilmenge von A ist:

$$dom(f) \subset A$$

Ist $dom(f) = A$, so heißt f **total**, und man schreibt $f : A \longrightarrow B$. Die Menge aller Funktionswerte wollen wir Bild von f nennen und mit $im\ f$ bezeichnen [2]:

$$im(f) = \{b \in B \mid \text{es gibt ein } a \in A \text{ mit } b = f(a)\}$$

Später wird es nötig sein, die entsprechenden Begriffe im Zusammenhang mit Relationen zur Hand zu haben:

Wenn $R \subseteq A \times B$ eine Relation zwischen gewissen Elementen der Mengen A und B ist, so soll

$$dom\ R = \{a \in A \mid \text{ es gibt ein } b \in B \text{ mit } (a, b) \in R\}$$

und

$$im\ R = \{b \in B \mid \text{ es gibt ein } a \in A \text{ mit } (a, b) \in R\}$$

sein.

Freie Vereinigung von Mengen

Es ist häufig erforderlich, Mengen A und B mit nicht leerem Durchschnitt zu vereinigen, jedoch so, dass in der Vereinigungsmenge ersichtlich ist, welche Elemente aus A oder B sind. Zum Beispiel können 0 und 1 aus A Zahlen sein, wogegen 0 und 1 aus B Wahrheitswerte bezeichnen. Um diese gleich bezeichneten, jedoch mit unterschiedlichen Bedeutungen verknüpften Elemente in der Vereinigungsmenge auseinander halten zu können, muss man sie entsprechend markieren. Das kann folgendermaßen geschehen:

$$A \sqcup B := (A \times \{\alpha\}) \cup (B \times \{\beta\})$$

$\sqcup$ ist das Operationszeichen für die freie, $\cup$ für die gewöhnliche Vereinigung von Mengen. ":=" soll als "ist per Definition gleich" gelesen werden. Diese Bezeichnung wollen wir im weiteren aber auch so verwenden, wie es in Programmiersprachen üblich ist, nämlich als Zuweisungsvorschrift für die Belegung von Variablen. Z.B. bedeutet $x := 3x + 5$, dass aus einer Belegung der Variablen x mit 4 die neue Belegung 17 geworden ist.

[2] image (engl.) = Bild

Definition 2.1.1 *Ein **endlicher deterministischer Automat** ist ein Tupel*

$$\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

GENAUE BEGRIFFSBESTIMMUNG

.

*Darin sind S, X, Y und M endliche Mengen, deren Elemente in dieser Reihenfolge **Zustände, Aktionen, Einblicke** und **Marken** heißen. Y bzw. X werden auch Eingabe- bzw. Ausgabealphabet genannt. M enthält stets das Element go:*

$$M = M' \sqcup \{go\}$$

.

*$s_0 \in S$ heißt **initialer** oder **Startzustand**.*

*μ nennen wir **Markierungsfunktion**. Durch sie werden die **aktiven** und die **terminalen** oder **Endzustände** charakterisiert:*

$$\mu : S \longrightarrow M$$
$$act\ (\mathfrak{A}) := \{s \in S \mid \mu(s) \in M'\})$$
$$term\ (\mathfrak{A}) := S \backslash act\ (\mathfrak{A}) = \{s \in S \mid \mu(s) \in M'\}$$

*δ und λ heißen **Zustandsüberführungs-** und **Ausgabefunktion***

$$\delta : act\ (\mathfrak{A}) \times Y \longrightarrow S \qquad \lambda : act\ (\mathfrak{A}) \times Y \longrightarrow X$$
$$(s, y) \mapsto s' = \delta(s, y) \qquad (s, y) \mapsto x = \lambda(s, y)$$

Die Funktionen δ und λ sind für aktive Zustände definiert. Für terminale Zustände liefern sie keine Werte. Das entspricht der oben gegebenen informalen Beschreibung. Dieser so definiert Automatentyp heißt **endlich**, weil alle beteiligten Mengen endlich sind, und er heißt **deterministisch**, weil alle Aktivitäten durch Funktionen, also eindeutig geregelt sind. Statt Automat wird auch die Bezeichnung Prozessor verwendet. Ein Automat mit nur einem Endzustand ist ein **Aktor**. Hat er zwei terminale Zustände, denen durch μ Wahrheitswerte zugewiesen sind, etwa *true* und $false$, so ist das ein **boolescher Automat**.

Dieses so definierte Modell eines Automaten $\mathfrak{A}$ beginnt zu leben, wenn es Informationen aus einer Menge Y empfängt, gibt Befehle aus einer Menge X ab und stirbt, wenn er einen Endzustand erreicht - was nicht immer eintreten muss - und liefert eine letzte Information aus einer Menge M. Diese vereinfachte Beschreibung entspricht der symbolischen Darstellung

$$Y \longrightarrow \boxed{\mathfrak{A}} \longrightarrow X$$
$$\downarrow$$
$$M$$

die wir auch benutzen wollen, um Automaten zu kennzeichnen. Da bei einem Aktor die Angabe einer Endmarke in der Regel keine Bedeutung hat, wir das Symbol

$$Y \longrightarrow \boxed{\mathfrak{A}} \longrightarrow X$$

benutzt.

2.1.2 Darstellung endlicher Automaten durch Graphen

Ein einfaches Beispiel

Wir zeigen die Methodik an einem Beispiel

$$\begin{aligned} S &= \{s_0, s_1, s_2, t_0, t_1, t_2\} \\ X &= \{0, -1\} \\ Y &= \{0, 1\} \\ M &= \{0, 1, 2, go\} \\ \mu(s_0) &= \mu(s_1) = \mu(s_2) = go \\ \mu(t_0) &= 0; \qquad \mu(t_1) = 1; \qquad \mu(t_2) = 2 \end{aligned}$$

δ und λ sind den folgenden Tabellen zu entnehmen:

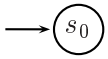
δ	0	1
s_0	t_0	s_1
s_1	t_1	s_2
s_2	t_2	s_0

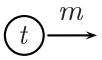
λ	0	1
s_0	0	-1
s_1	0	-1
s_2	0	-1

Mit diesen Festlegungen ist ein Automat gemäß der Definition gegeben. Sei *mod_3* der Name dieses Prozessors. Man bemerkt, dass $act(mod\_3) = \{s_0, s_1, s_2\}$ ist.

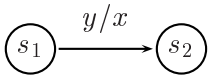
Diese Form der Darstellung ist aber sehr unanschaulich. Deshalb wird vorzugsweise mit der folgenden Symbolik gearbeitet:

Automaten-bilder

- Startzustand: 

- ein Endzustand t mit der Marke $m = \mu(t)$: 

- ein Übergang von s_1 nach $s_2 = \delta(s_1, y)$ mit der Ausgabe $x = \lambda(s_1, y)$:



Mit diesen Vereinbarungen kann der Prozessor in Gestalt eines Graphen angegeben werden (vgl. Abbildung 2.1), dessen Knoten die Elemente von S sind, weshalb wir auch vom **Zustandsgraphen** des Automaten reden.

Ein solches Bild vermittelt eine weniger abstrakte Vorstellung von einem Automaten als die obige Definition. Dennoch muss man sich klar darüber sein, dass auch damit ein Begriff ohne inhaltliche Bedeutung erklärt ist. Es handelt sich lediglich um Mengen und gewisse Funktionen über diesen Mengen. Eine Semantik ist damit noch nicht verbunden. Später, wenn definiert ist, in welchen Umwelten der in diesem Beispiel vorgestellte Automat arbeiten kann und in welcher Weise er die Punkte dieser Umwelten beeinflusst, werden wir sehen, dass er richtig rechnen kann.

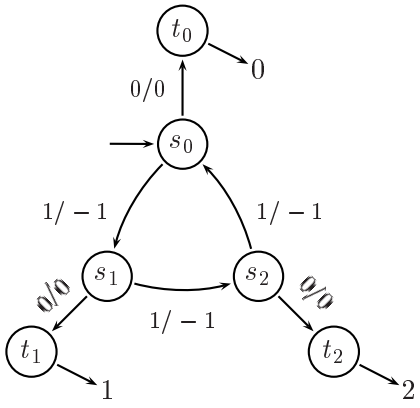


Abbildung 2.1: *mod_3* als Zustandsgraph

2.1.3 Verknüpfung von Automaten

Konstruktion von Automaten nach dem Baukastenprinzip

Es ist ein nahe liegender Gedanke, Automaten miteinander zu verbinden. Wenn ein Prozessor $\mathfrak{A}_1$ einen terminalen Zustand t erreicht hat, kann ein weiterer Automat $\mathfrak{A}_2$ seine Arbeit beginnen und eine Folgeaufgabe lösen. Dazu muss t mit dem Startzustand von $\mathfrak{A}_2$ identifiziert werden. Die Zustandsüberführung ist an der Schnittstelle in geeigneter Weise zu definieren.

Betrachten wir den einfachsten Fall, dass $\mathfrak{A}_1$ und $\mathfrak{A}_2$ Aktoren sind:

$$\mathfrak{A}_i = (S_i, s_{0i}, X, Y, M_i, \delta_i, \lambda_i, \mu_i); \qquad i = 1, 2$$

t_1 und t_2 seien die terminalen Zustände. Daraus entsteht ein neuer Aktor:

$$\mathfrak{A} = \mathfrak{A}_1; \mathfrak{A}_2 = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

vermöge der folgenden Definition:

$$S := (S_1 \sqcup S_2) \backslash \{t_1\}; \quad s_0 := s_{01}; \quad M := M_2$$

$$term\, \mathfrak{A} := \{t_2\}$$

$$\mu(s) := \begin{cases} \mu_2(s) & \text{falls } s = t_2 \\ go & \text{sonst} \end{cases}$$

$$\lambda := \begin{cases} \lambda_1(s, y) & \text{falls } s \in act\, \mathfrak{A}_1 \\ \lambda_2(s, y) & \text{falls } s \in act\, \mathfrak{A}_2 \end{cases}$$

$$\delta(s, y) := \begin{cases} \delta_1(s, y) & \text{für } s \in act\, \mathfrak{A}_1 \text{ und } \delta_1(s, y) \neq t_1 \\ s_{02} & \text{für } s \in act\, \mathfrak{A}_1 \text{ und } \delta_1(s, y) = t_1 \\ \delta_2(s, y) & \text{für } s \in act\, \mathfrak{A}_2 \end{cases}$$

Offenbar ist $\mathfrak{A}$ wieder ein endlicher, deterministischer Automat, genauer ein Aktor, den wir schematisch in der Form $\rightarrow \mathfrak{A}_1 \rightarrow \mathfrak{A}_2 \rightarrow \circ$ angeben können.

Konstrukte einer zukünftigen Automatensprache

Bei den nachfolgenden Verknüpfungen muss in analoger Weise verfahren werden. Die ausführliche Darstellung wie im vorangegangenen Beispiel ersparen wir uns hier. Es wird auf einen allgemeinen Satz über R-Schemata im Abschnitt 2.3.3 verwiesen.

$\mathfrak{A}$, $\mathfrak{A}_1$ und $\mathfrak{A}_2$ sind wieder Aktoren, $\mathfrak{B}$ sei ein boolescher Prozessor. Bei allen Automaten sind X und Y dieselben Mengen. Daraus werden die in Abbildung 2.3 dargestellten Automaten (a) bis (d) zusammengesetzt. Sie werden in naheliegender Weise wie folgt benannt:

(a) if $\mathfrak{B}$ then $\mathfrak{A}$ end if

(b) if $\mathfrak{B}$ then $\mathfrak{A}_1$ else $\mathfrak{A}_2$ end if

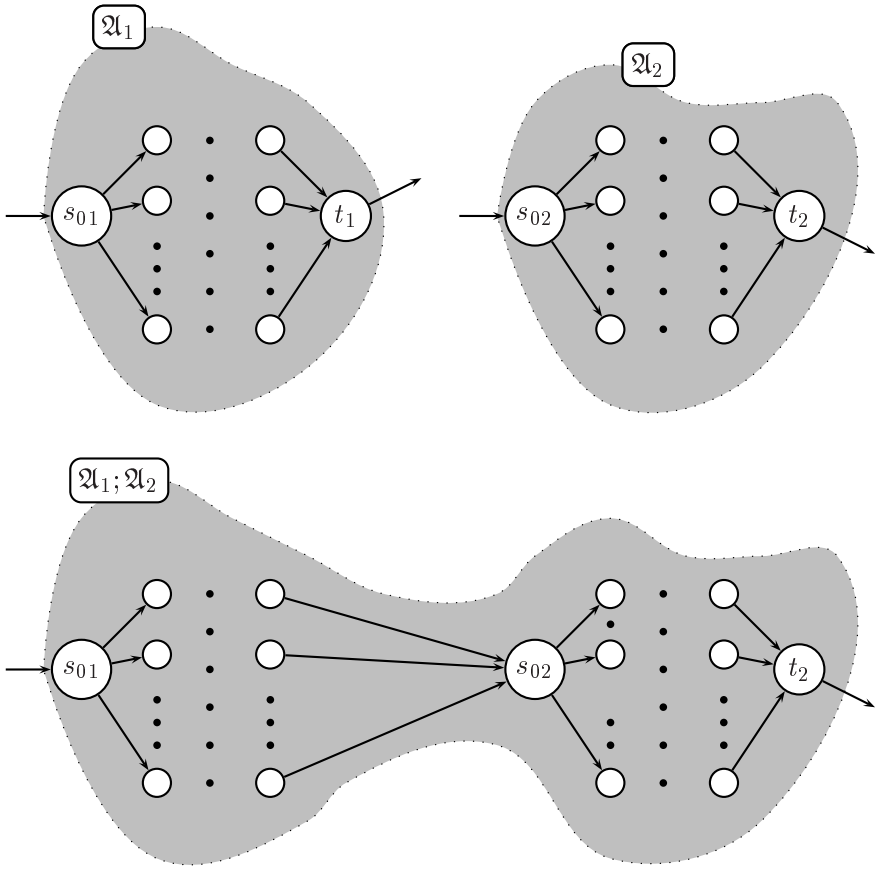


Abbildung 2.2: Verknüpfung zweier Aktoren

(c) while $\mathfrak{B}$ do $\mathfrak{A}$ end while

(d) repeat $\mathfrak{A}$ until $\mathfrak{B}$

Dies sind – ausgenommen der Automat (b) – wieder Aktoren. Die $if-then-else-end\ if$-Verknüpfung kann je nach Verwendungszweck ein oder auch zwei terminale Zustände haben. Wir werden darauf zurück kommen, wenn auf der Grundlage dieser Konstruktionen eine sprachliche Darstellung von Automaten erläutert wird.

Abschließend zu diesem Abschnitt noch eine Konstruktion mit booleschen Automaten, die oft nützlich ist. Wenn $\mathfrak{B}$ ein solcher ist, so möchte man bisweilen die Ausgabe der Wahrheitswerte vertauschen. Diesen "*negierten*" Automaten werden wir *not* $\mathfrak{B}$ nennen.

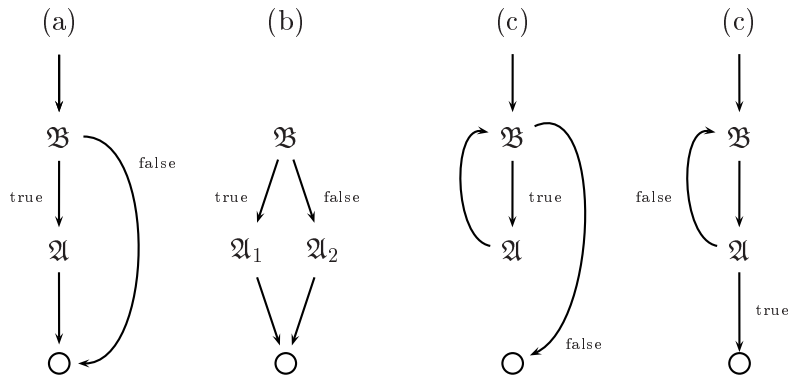


Abbildung 2.3: Die wichtigsten Verknüpfungen von Automaten

Definition 2.1.2 *Ist*

$$\mathfrak{B} = (S, s_0, X, Y, \{m_1, m_2\}, \delta, \lambda, \mu)$$

ein boolescher Automat mit

$$term\ \mathfrak{B} = \{t_1, t_2\}\ und\ \mu(t_1) = m_1;\ \mu(t_2) = m_2,$$

so ist

$$not\ \mathfrak{B} = (S, s_0, X, Y, \{m_1, m_2\}, \delta, \lambda, \bar{\mu})$$

definiert durch

$$term\ \mathfrak{B} = \{t_1, t_2\}\ und\ \bar{\mu}(t_1) = m_2;\ \bar{\mu}(t_2) = m_1$$

2.1.4 Gleichwertige Automaten

Die Abbildung 2.4 zeigt einen Automaten, dessen Zustand s_1 von Startzustand s_0 aus nicht erreichbar ist. Offenbar kann man s_1 samt der Pfeile, welche von diesem Knoten ausgehen, streichen. Wie kann man diese Vermutung rechtfertigen?

Zur sprachlichen Verständigung führen wir einen Begriff ein, der noch oft benötigt wird.

Σ sei eine beliebige endliche Menge, die in dem nachfolgenden Kontext Alphabet genannt wird, die Elemente von Σ heißen Buchstaben. Zum Beispiel könnte Σ der ASCII-Zeichensatz sein. Außerdem wird für positive natürliche Zahlen n die Bezeichnung

$$[n] := \{1, 2, \cdots, n\}$$

benutzt.

WÖRTER, GEBILDET AUS BUCHSTABEN EINES BELIEBIGEN ALPHABETS

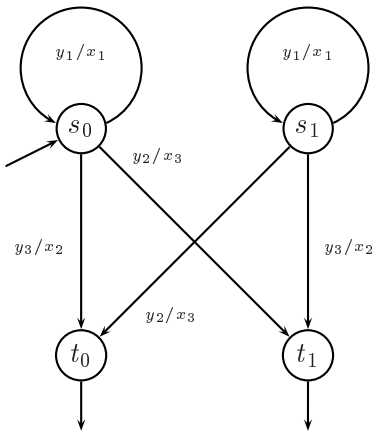


Abbildung 2.4: Ein Automat mit einem überflüssigen Zustand

Definition 2.1.3

- *Jede Abbildung* $w : [n] \to \Sigma$ *heißt Wort der Länge* n *über* Σ
- Σ^n *ist die Menge aller Wörter der Länge* n
- $\Sigma^+ := \bigcup_{n \geq 1} \Sigma_n$

Als Beispiel nehmen wir die Abbildung $w : [7] \to \Sigma$, wo Σ das lateinische Alphabet und $w(1) = A$, $w(2) = u$, $w(3) = w(7) = t$, $w(4) = o$, $w(5) = m$ und schließlich $w(6) = a$ ist. Das ist die saubere Definition eines Wortes, das üblicherweise in der Gestalt "Automat" aufgeschrieben wird. Auch in der Informatik wird diese Bezeichnungsweise benutzt und $w_i := w(i)$ sowie $w_1 w_2 \dots w_n$ geschrieben.

Wörter kann man miteinander verketten, indem man sie hintereinander schreibt. Das ist eine Operation[3]

$$\begin{array}{rcl} conc : \Sigma^+ \times \Sigma^+ & \to & \Sigma^+ \\ (v, w) = (v_1 v_2 \dots v_m, w_1 w_2 \dots w_n) & \mapsto & conc(v, w) := \mathrm{vw} = v_1 v_2 \dots v_m w_1 w_2 \dots w_n \end{array}$$

Mit dieser Operation ist Σ^+ eine Halbgruppe. Um diese algebraische Struktur aufzuwerten zu einem Monoid, wird noch ein so genanntes **leeres Wort** eingeführt, dessen Länge auf null festgelegt wird. Es wird, unabhängig von Alphabet, mit ϵ bezeichnet. Für die Menge aller Wörter über einem Alphabet Σ wird

$$\Sigma^* := \Sigma^+ \cup \{\epsilon\}$$

[3] conc=concat(engl.)=verketten

geschrieben.

Von ϵ wird verlangt, dass es sich bezüglich der Verkettung von Wörtern neutral verhält:

$$conc(w, \epsilon) = conc(\epsilon, w) = w \quad \text{für alle } w \in \Sigma^*$$

GLOBALE ANALYSE EINES AUTOMATEN

Um die Arbeit eines beliebigen Automaten zu beschreiben, betrachten wir Wörter $w = y_1 y_2 \dots y_n \in Y^*$. Ein solches Wort kann eine Folge von Ausgaben erzeugen. Dem entspricht im Zustandsgraphen ein Pfad

$$s_0 \xrightarrow{y_1/x_1} s_1 \xrightarrow{y_2/x_2} s_2 \xrightarrow{y_2/x_2} \dots \xrightarrow{y_n/x_n} s_n \tag{2.1}$$

Ein solcher Pfad existiert natürlich nur dann, wenn zwischendurch kein terminaler Zustand erreicht wird.

Die wesentlichen Informationen über alle von s_0 ausgehenden Pfade werden durch zwei Funktionen $\alpha : Y^* \to S$ und $\beta : Y^* \to X^*$ geliefert, die so definiert sind, dass für den Term 2.1 die Funktionswerte $\alpha(w) = s_n$ und $\beta(w) = x_1 x_2 \dots x_n$ erhalten werden.

Definition 2.1.4 *Sei $\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ ein Prozessor, $u \in Y^*$, $y \in Y$.*

$\alpha_{\mathfrak{A}} : Y^* \xrightarrow{\supset} S$ *definiert durch*

$$\alpha_{\mathfrak{A}}(\epsilon) := s_0$$

$$\alpha_{\mathfrak{A}}(uy) := \begin{cases} \delta(\alpha_{\mathfrak{A}}(u), y), & \text{falls } \alpha_{\mathfrak{A}}(u) \in act\ \mathfrak{A} \\ \perp \quad \text{sonst} \end{cases}$$

$\beta_{\mathfrak{A}} : Y^* \xrightarrow{\supset} X^*$ *vermöge*

$$\beta_{\mathfrak{A}}(\epsilon) := \epsilon$$

$$\beta_{\mathfrak{A}}(uy) := \begin{cases} \beta_{\mathfrak{A}}(u)\lambda(\alpha_{\mathfrak{A}}(u), y)), & \text{falls } \beta_{\mathfrak{A}}(u) \neq \perp \text{ und } \alpha_{\mathfrak{A}}(u) \in act\ \mathfrak{A} \\ \perp \quad \text{sonst} \end{cases}$$

Wir haben angedeutet, dass die Ausgaben $x \in X$ eines Automaten zu Veränderungen in Umwelten führen. Deshalb wollen wir annehmen, dass es immer eine Ausgabe "stop" gibt, die nichts bewirkt. Die Einführung einer solchen neutralen Aktion in einer Umwelt erleichtert manchen Beweis. In einem Ausgabewort $\beta(w)$ sind solcher Outputs jedoch ohne Bedeutung. Das führt uns zu

ÄQUIVALENTE AUTOMATEN

Definition 2.1.5

- *$\psi_{\mathfrak{A}} : Y^* \xrightarrow{\supset} (X \backslash \{stop\})^*$ ist die Funktion, welche aus $\beta_{\mathfrak{A}}$ dadurch hervorgeht, dass man in jedem Wort $\beta_{\mathfrak{A}} = x_1 x_2 \dots x_n$ alle x_i mit $x_i = stop$ löscht. $\psi_{\mathfrak{A}}$ nennt man* ***Blackbox-Verhalten*** *von $\mathfrak{A}$.*
- *Zwei Automaten $\mathfrak{A}$ und $\mathfrak{B}$ heißen* ***ununterscheidbar****, wenn sie gleiches Blackboxverhalten haben, d.h. wenn $\psi_{\mathfrak{A}} = \psi_{\mathfrak{B}}$ gilt.*

- *Zwei Automaten heißen* ***äquivalent****, wenn sie ununterscheidbar sind und in der Ausgabe von Endmarken übereinstimmen, d.h. wenn für alle $w \in Y^*$ gilt*

$$\alpha_{\mathfrak{A}} \in term\,\mathfrak{A} \text{ genau dann, wenn } \alpha_{\mathfrak{B}} \in term\,\mathfrak{B} \text{ und}$$
$$\mu_{\mathfrak{A}}(\alpha_{\mathfrak{A}}(w)) = \mu_{\mathfrak{B}}(\alpha_{\mathfrak{B}}(w))$$

Mit diesen Begriffen lässt sich die eingangs gestellte Frage präzisieren und beantworten. Ein Zustand $\sigma \in S$ ist nicht erreichbar, wenn er nicht zum Wertebereich von $\alpha_{\mathfrak{A}}$ gehört: Es gibt keine $w \in Y^*$ so, dass $\alpha_{\mathfrak{A}}(w) = \sigma$ ist.

Nun wird zu

$$\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

der Automat

$$\mathfrak{A}' = (S', s_0, X, Y, M, \delta', \lambda', \mu')$$

betrachtet, wobei σ ein solcher unerreichbarer Zustand ist. Es wird $S' := S \backslash \{\sigma\}$ gesetzt. δ', λ' sowie μ' sind die Einschränkungen von δ und λ auf $S' \times Y$ bzw. von μ auf S', was heißt, dass zur Bildung der Funktionswerte σ nicht mehr benutzt wird. Das bedeutet für den Zustandsgraphen, dass alle von σ ausgehenden Pfeile gelöscht werden.

Offenbar ist $\mu_{\mathfrak{A}}(\alpha_{\mathfrak{A}}(w)) = \mu_{\mathfrak{A}'}(\alpha_{\mathfrak{A}'}(w))$ fr alle $w \in Y^*$.

Es ist aber auch $\alpha_{\mathfrak{A}} = \alpha_{\mathfrak{A}'}$ sowie $\beta_{\mathfrak{A}} = \beta_{\mathfrak{A}'}$. Das lässt sich simultan durch vollständige Induktion nach der Länge von w beweisen.

Ist $w = \epsilon$, so ist per Definition $\alpha_{\mathfrak{A}}(\epsilon) = \alpha_{\mathfrak{A}'}(\epsilon) = s_0$ und $\beta_{\mathfrak{A}}(\epsilon) = \beta_{\mathfrak{A}'}(\epsilon) = \epsilon$.

Nun sei n eine beliebige natürliche Zahl, und es möge $\alpha_{\mathfrak{A}}(u) = \alpha_{\mathfrak{A}'}(u)$ und $\beta_{\mathfrak{A}}(u) = \beta_{\mathfrak{A}'}(u)$ gelten für alle Wörter $u \in Y^*$ der Länge n. Da $\alpha_{\mathfrak{A}}(u) \neq \sigma$ ist, bekommt man für ein beliebiges $y \in Y^*$:

$$\alpha_{\mathfrak{A}}(uy) = \delta(\alpha_{\mathfrak{A}}(u), y) = \delta'(\alpha_{\mathfrak{A}}(u), y) = \delta'(\alpha_{\mathfrak{A}'}(u), y) = \alpha'_{\mathfrak{A}'}(uy)$$

Damit ist $\alpha_{\mathfrak{A}}(w) = \alpha_{\mathfrak{A}'}(w)$ gezeigt für alle $w \in Y^*$. Mit diesem Resultat ergibt sich

$$\beta_{\mathfrak{A}}(uy) = \beta_{\mathfrak{A}}(u)\lambda(\alpha_{\mathfrak{A}}(u), y) = \beta_{\mathfrak{A}'}(u)\lambda'(\alpha_{\mathfrak{A}'}(u), y) = \beta_{\mathfrak{A}'}(uy)$$

Das war der Beweis für

Überflüssige Zustände

Folgerung 2.1.1 *Wenn man in einem Automaten Zustände streicht, die vom Startzustand aus nicht erreichbar sind, erhält man einen äquivalenten Automaten.*

In unterschiedlichem Zusammenhang ist es nützlich, einem Automaten Zustände hinzuzufügen, indem ein Pfad

$$s_0 \xrightarrow{y_1/x_1} \ldots \xrightarrow{y_i/x_i} s_i \xrightarrow{y_{i+1}/x_{i+1}} s_{i+1} \xrightarrow{y_{i+2}/x_{i+2}} \ldots$$

ergänzt wird zu

$$s_0 \xrightarrow{y_1/stop} \ldots \xrightarrow{y_i/x_i} s_i \xrightarrow{y_{i+1}/stop} \sigma_i \xrightarrow{y_{i+1}/x_{i+1}} s_{i+1} \xrightarrow{y_{i+2}/stop} \ldots$$

Da nach unserer Definition für Automaten die Funktionen δ und λ zu jedem aktiven Zustand und zu jedem Einblick einen Wert liefern müssen, fügen wir außerdem einen zusätzlichen terminalen Zustand τ hinzu, zu dem von jedem aktiven Zustandsknoten s_i für alle Einblicke $y \in Y$, die verschieden von y_i sind, Kanten hinlaufen:

$$\delta(\sigma_i, y) := \tau;\ \lambda(\sigma_i, y) := stop \text{ für alle } y \in Y \text{ mit } y_i \neq y$$

Dies Konstruktion soll **Überführung in die Normalform** heißen.

Folgerung 2.1.2 *Ein Automat und seine Normalform sind ununterscheidbar.*

Den Nachweis dieser Aussage überlassen wir dem Leser.

2.2 Umwelten

2.2.1 Klärung des Begriffs

ZUERST INFORMAL

Bei der Definition des Automaten sind wir davon ausgegangen, dass er Signale empfängt und Aktionen ausgibt. Er soll also mit seiner Umwelt kommunizieren. Jetzt wollen wir beschreiben, was wir uns unter einer Umwelt vorstellen.

Es kommt darauf an, ein sehr allgemeingültiges Modell zu gestalten. Denken wir zum Beispiel an einen automatischen Staubsauger, so muss er in der Lage sein, in beliebig großen Räumen bei ganz verschiedenartiger Möbelausstattung zu arbeiten.

Entsprechend flexibel muss unser Modell sein. Es liegt nahe, die zu säubernde Fläche mit einem Raster zu überziehen. Jedes Quadrat des Rasters ist ein Umweltpunkt. Der Staubsauger muss mit Sensoren ausgestattet sein, welche anzeigen, wo Hindernisse sind. Auf diese Weise werden die Einblicke vermittelt, von denen es abhängt, in welche Richtung sich der Automat bewegen kann. Eventuell muss er sich vorher drehen, ehe er in seiner Arbeit fortfahren kann. Welche Aktionen jeweils auszuführen sind, wird also von den Informationen abhängen, die aus der Umwelt empfangen werden, aber auch von der augenblicklichen Beschaffenheit des Automaten, von dem Zustand, in dem er sich befindet.

Es wäre eine voreilige Einschränkung, wenn das Modell einer Umwelt nur endlich viele Punkte hätte. Das entspräche nicht den möglichen Anwendungen. Denken wir z.B. daran, dass der Speicher eines Computers als Umwelt interpretiert wird, so entspricht es doch vollständig der rasanten technischen Entwicklung, Speicher jeder denkbaren Größe anzunehmen. Auch die Umwelt eines Raumfahrers ist unendlich. Und wenn sich jemand - wo auch immer - verirrt hat, so sind ihm die Grenzen seiner Umwelt vollständig unbekannt. Schließlich würde die Anwendung wichtiger Mittel und Methoden der Mathematik nicht möglich sein, wenn wir uns auf endliche Umwelten einschränken würden. Unser Umweltmodell wird im allgemeinen aus einer unendlichen Punktmenge bestehen, die man punktuell erkennen kann und in der man sich in Abhängigkeit von lokal getroffenen Entscheidungen bewegt. Globale, große Probleme müssen in kleinen Schritten mit endlichen Mitteln gelöst werden.

FORMALE BESCHREIBUNG

Definition 2.2.1 *Eine Umwelt $U = (X, Y, Z, d, l)$ besteht aus zwei endlichen Mengen X und Y, einer im allgemeinen unendlichen Menge Z und zwei Funktionen*

$$d : Z \times X \stackrel{\supset}{\longrightarrow} Z \qquad l : Z \longrightarrow Y$$
$$(P, x) \mapsto Q = d(P, x) \qquad P \mapsto y = l(P)$$

*Die Elemente der Mengen X, Y und Z heißen **Aktionen**, **Einblicke** bzw. **Punkte** der Umwelt.*

Die Funktion l ist total. Sie liefert zu jedem Punkt der Umwelt einen Einblick. d gibt an, wie unter dem Einfluss von Aktionen Bewegungen in der Umwelt

erfolgen können, wobei nicht jede Aktion in jedem Punkt ausführbar sein muss: d ist im allgemeinen eine partielle Funktion.

Statt $d(P,x) = Q$ werden wir häufig die kürzere Schreibweise $Q = Px$ verwenden.

DIE ZÄHLERUMWELT

Wir wollen die Definition an einem einfachen Beispiel illustrieren:

$$U = (\{0,1,-1\}, \{0,1\}, \mathbb{N}, d, l)$$

$$l(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{sonst} \end{cases} \quad ; \quad d(n,x) = \begin{cases} \perp & \text{falls } (n,x) = (0,-1) \\ n+x & \text{sonst} \end{cases}$$

$\mathbb{N}$ ist die Menge der natürlichen Zahlen. Mit anderen Worten, jede natürliche Zahl ist ein Umweltpunkt. Von einer Zahl lässt sich lediglich ermitteln, ob sie Null oder verschieden von Null ist. Und die Zahlen lassen sich nur in Schritten der Größe 1 vergrößern oder verkleinern. Die Null ist eine Aktion, welche nichts verändert.

Diese "Zählerumwelt" werden wir noch häufig benutzen. Sie ist auch interpretierbar als Modell eines einzelnen Speicherplatzes eines Computers. Jede Belegung der Speicherzelle ist ein Punkt der Umwelt. Die Wirkungsweise der Funktion d entspricht der Vorstellung, dass man den Zelleninhalt nur bitweise verändern kann. Die Funktion l drückt aus, dass man zwar erkennen kann, ob der Speicher leer oder belegt ist, mehr weiß man jedoch nicht.

Da wir diese Umwelt oft benötigen, wird sie im weiteren mit $\mathbb{N}$ abgekürzt. Diese Polymorphie in der Bezeichnung verwischt zwar etwas die Begriffe, ist aber sehr bequem und durchaus üblich.

EIN PROZESSOR RECHNET

Jetzt können wir einen Bezug zum Automaten aus dem Abschnitt 2.1.2 (vgl. Abb. 2.1) herstellen. Dieser Prozessor benutzt Aktionen und Einblicke der Zählerumwelt. Und wenn er in einem Punkt gestartet wird, so beendet er seine Arbeit, wenn Null erreicht ist und gibt als Endmarke den Rest bei Division durch 3 aus.

Die Arbeit des Prozessors hat einen Sinn bekommen, er berechnet in der Umwelt $\mathbb{N}$ die Funktion $f(n) = n \ mod \ 3$

Die Umwelt bestimmt die Semantik eines Automaten. Derselbe Prozessor wird sinnlos, wenn wir unsere Umwelt geringfügig verändern: statt d soll d' mit

$$d'(n,x) = \begin{cases} \perp & \text{für } (n,x) = (0,1) \\ n-x & \text{sonst} \end{cases}$$

benutzt werden.

Jetzt arbeitet der Automat nur noch "vernünftig", wenn er im Punkte $n = 0$ gestartet wird . Er erreicht keinen Endzustand, wenn $n > 0$ ist.

Die Beziehungen zwischen einem Automaten und seiner Umwelt werden noch weiter zu untersuchen sein. Bevor wir weitere Beispiele für Umwelten präsentieren, sollen noch drei Bemerkungen über spezielle Eigenschaften einer Umwelt eingefügt werden.

1. Die in $\mathbb{N}$ durch die Null repräsentierte **Stoppaktion** ist in vielen Zusammenhängen ein bequemes Hilfsmittel. Deshalb vereinbaren wir, dass die Aktionsmenge X immer mindestens diese Aktion enthält. Sie wird bei generellen Definitionen "*stop*" heißen und der Gleichung $d(P, stop) = P$ für alle Punkte P der Umwelt genügen.

2. Häufig gibt es einen Umweltpunkt, der eine besondere Rolle spielt. In $\mathbb{N}$ ist das die Null. Eine Umwelt U mit einem ausgezeichneten Punkt O heißt **punktierte Umwelt**. Wir benutzen die Schreibweise (U, O).

3. Will man für die partielle Wirkungsfunktion d mit endlichen Mitteln ausdrücken, für welche der unendlich vielen Punkte der Umwelt sie definiert ist, kann man die Einblicksfunktion l benutzen. Man stiftet in der Punktmenge Z eine Äquivalenzrelation $\equiv_{Ker(l)} \subseteq Z \times Z$, indem man definiert

   $$P \equiv_{Ker(l)} Q \text{ genau dann, wenn } l(P) = l(Q)$$

   Dadurch wird Z in disjunkte Klassen eingeteilt, wobei in jeder Klasse alle zueinander äquivalenten Punkte versammelt sind. Es gibt ebenso viele Klassen, wie Y Elemente hat. Schließlich ordnet man jeder Klasse diejenigen Aktionen zu, welche in allen Punkten dieser Klasse auch ausführbar sind.

   Zur sprachliche Verständigung benutzen wir den Begriff der **Valenz eines Punktes**, das ist die Menge der in diesem Punkte ausführbaren Aktionen:

   $$val(P) = \{x \in X \mid d(P, x) \neq \perp\}$$

   Damit ist der mathematische Hintergrund der nachfolgenden Definition skizziert.

   Bevor wir sie aufschreiben, weisen wir noch auf eine Bezeichnungskonvention hin. Zu einer Relation $R \subseteq Y \times X$ kann man die Funktion $\tilde{R}$ von Y in die Potenzmenge $\mathfrak{P}(X)$ der Menge X betrachten, welche durch

   $$\tilde{R}(y) := \{x \in X \mid (y, x) \in R\}$$

   definiert ist.

   Umgekehrt gibt es zu einer solchen Funktion

   $$\tilde{R} : Y \longrightarrow \mathfrak{P}(X)$$

eine wohlbestimmte Relation $R \subseteq Y \times X$, nämlich

$$R := \left\{ (y, x) \mid y \in Y \text{ und } x \in \tilde{R}(y) \right\}.$$

Es ist üblich, zwischen R und $\tilde{R}$ nicht zu unterscheiden. Das wird im folgenden praktiziert.

Definition 2.2.2 *Eine Umwelt* $U = (X, Y, Z, d, l)$ *heißt* ***R-Umwelt****, wenn* $R \subseteq Y \times X$ *eine Relation ist mit der Eigenschaft* $R(l(p)) \subseteq val(P)$ *für alle* $P \in Z$.

Mit anderen Worten: in einer R -Umwelt weiß man in jedem Punkte, welche Aktionen möglich sind, sobald man den Einblick ermittelt hat.

In $\mathbb{N}$ kann man zum Beispiel R mit

$$R(0) = \{0, 1\} \text{ und } R(1) = X = \{0, 1, -1\}$$

wählen. In diesem Falle stimmt $R(l(P))$ sogar mit $val(P)$ überein.

Mit Blick auf den Automaten, der $f(n) = n \ mod \ 3$ berechnet und die Aktion 1 gar nicht benutzt, ist $\mathbb{N}$ mit $\tilde{R}(0) = \tilde{R}(1) = \{0, -1\}$ auch eine $\tilde{\text{R}}$ -Umwelt und $\tilde{R}(l(P))$ ist eine echte Teilmenge der in P ausführbaren Aktionen, wenn P eine positive Zahl ist.

Wir wollen noch ein etwas ausführlicheres Beispiel einfügen. Ein automatischer Staubsauger (oder irgendein anderer Automat mit ähnlicher Aufgabenstellung) soll sich in einem beliebigen Raum, also auch mit irgendwie gearteten Hindernissen zurechtfinden können. Es ist praktikabel, die Fußbodenfläche mit einem quadratischen Raster zu überziehen. Die Größe der Quadrate entspricht der "Schrittweite" des Automaten. Die Quadrate sind Umweltpunkte. Ihre Position lässt sich mit Paaren ganzer Zahlen bestimmen, sobald man ein Koordinatensystem festgelegt hat. Wenn wir noch diejenigen Rasterquadrate eliminieren, die von Hindernissen (Stuhlbeinen, Möbeln u.a.) besetzt sind, erhalten wir die Punktmenge Z. $\mathbb{Z}$ sei die Menge der ganzen rationalen Zahlen. Die Menge der Umweltpunkte ist $Z := \mathbb{Z} \times \mathbb{Z} \backslash M$. Darin ist M eine endliche Menge von Zahlenpaaren, welche die Hindernisse markiert, wo die Maschine nicht hin kann.

Wir wollen annehmen, dass der Automat sich in Richtung der Achsen eines orthogonalen Koordinatensystems bewegen kann:

$$X = \{n, s, o, w\} \text{ mit } n = (0, 1),\ s = (0, -1),\ o = (1, 0),\ w = (-1, 0).$$

Die Buchstaben sollen eine Assoziation zu den vier Himmelsrichtungen herstellen.

Damit ist

$$d(P, x) = d((a, b)(x_1, x_2)) = (a + x_1, b + x_2).$$

Ein Einblick in einem Punkte P dieser Umwelt soll die Menge der Richtungen sein, in die sich der Automat bewegen kann, das heißt, wo Nachbarpunkte von P nicht in M liegen:

$$Y = \mathfrak{P}(\{n, s, o, w\}) \text{ und } l(P) = \{x \in X \mid d(P, x) \notin M\}$$

In dieser Umwelt ist per Definition $val(P) = l(P)$, und mit $R(l(P)) := l(P)$ ist das eine R -Umwelt.

Dieselbe Umwelt kann auch als Modell für ein Labyrinth benutzt werden. Was man landläufig so nennt, sind hier die Punkte "im Inneren" von M. Wenn nun die Aufgabe darin bestehen sollte, einen Ausgang aus dem Labyrinth zu finden, der nicht im nördlichen Bereich von M liegt, dann ist es sinnvoll, eine Relation $\tilde{R}$ durch $\tilde{R}(l(P)) := l(p) \backslash \{n\}$ zu definieren. Ein Suchroboter, der sich gemäß $\tilde{R}$ verhält, wird sich im Labyrinth nicht mehr nach Norden bewegen.

2.2.2 Beispiele

Die ganzen rationalen Zahlen

Die Zählerumwelt ist Vorbild

Wie üblich, bezeichnen wir die Menge dieser Zahlen mit $\mathbb{Z}$. Wie in der Umwelt der natürlichen Zahlen wird vorausgesetzt, dass man sich in Schritten der Größe 1 bewegen kann. Von einer Zahl soll erkennbar sein, ob sie positiv, negativ oder gleich Null ist. Ausgezeichneter Punkt ist die Null.

Definition 2.2.3

$$U = (X, Y, Z, d, l) \quad mit\ X = Y = \{0, 1, -1\} \quad und \quad Z = \mathbb{Z}$$

$$l(z) = \begin{cases} 1 & für\ \ z > 0 \\ 0 & für\ \ z = 0 \\ -1 & sonst \end{cases} ; \qquad d(z, x) = z + x$$

In dieser Umwelt ist d eine totale Funktion. Ebenso wie bei den natürlichen Zahlen werden wir zwischen der Umwelt und ihren Punkten keinen Unterschied machen und $\mathbb{Z}$ als Symbol für beide benutzen.

Wenn man für die ganzen Zahlen die übliche Repräsentation mit einem Zahlenstrahl benutzt, dann kann ein Automat in dieser Umwelt nach der Definition erkennen, ob er sich über der Null oder rechts bzw. links davon befindet und in Abhängigkeit von dieser Einsicht agieren.

Gelegentlich werden wir die ganzen Zahlen mit noch weniger Information ausstatten:

$$\tilde{\mathbb{Z}} := (\{0, 1, -1\}, \{*\}, \mathbb{Z}, d, \tilde{l})$$

$$\tilde{l}(z) = * \ \text{ für alle } z \in \mathbb{Z}; \qquad d(z, x) = z + x$$

In dieser Umwelt kann sich ein Automat zwar noch unbeschränkt bewegen, wo er sich auf dem Zahlenstrahl befindet ist ihm aber völlig unbekannt.

Stackumwelten

Wörter als Punkte einer Umwelt

Punkte einer Stackumwelt sind die Wörter, welche sich mit einem endlichen Alphabet Σ bilden lassen. Wegen dieser Bindung an ein Alphabet verwenden wir S_Σ , um die Umwelt zu bezeichnen. Wie bei $\mathbb{N}$ stellen wir uns einen Speicher vor, der Wörter enthalten kann. Charakteristikum eines Stack- oder Stapelspeichers ist, dass man nur die letzte Eintragung erkennt und Veränderungen der Speicherbelegung auch nur am Wortende möglich sind.

Definition 2.2.4

$S_\Sigma = (X, Y, Z, d, l)$ *mit* $Z = \Sigma^*$; $X = \Sigma \cup \{pop, stop\}$; $Y = \Sigma \cup \{\varepsilon\}$.
Ist $w = \sigma_1\sigma_2 \ldots \sigma_n \in \Sigma^n$, *so ist* $l(w) := \sigma_n$; $l(\varepsilon) := \varepsilon$.
Mit $x \in \Sigma$ *ist* $d(w, x) = wx = \sigma_1\sigma_2 \ldots \sigma_n x \in \Sigma^{n+1}$; $d(\varepsilon, x) = x$.
$d(w, pop) = \sigma_1\sigma_2 \ldots \sigma_{n-1}$, *wenn* $n > 1$ *ist.*
$d(\sigma_1, pop) = \varepsilon$; $d(\varepsilon, pop) = \perp$.

Häufig werden wir die punktierte Stackumwelt (S_Σ, ε) benutzen.

Mit $R \subseteq Y \times X$, definiert durch $R(\sigma) = X$ für alle $\sigma \in \Sigma$ und $R(\varepsilon) = \Sigma$ ist S_Σ eine R -Umwelt.

Lese- und Schreibumwelten

In einer Stackumwelt besorgen die Aktionen $x \in X$, also die Buchstaben des Alphabets das Schreiben, wogegen mit *pop* ein Wort buchstabenweise aus dem Speicher herausgelesen wird. Reduziert man folglich in einer Stackumwelt X auf die eine oder die andere Gruppe der Aktionen, erhält man eine Schreib- bzw. Leseumwelt. Allerdings werden wir meistens der Konvention folgen, dass von links her gelesen und nach rechts geschrieben wird. Die Bezeichner R_Σ und W_Σ sollen eine Assoziation zu den englischen Wörtern *read* und *write* herstellen. (R_Σ darf nicht mit der oben eingeführten Relation R in Beziehung gebracht werden.)

Leseumwelt über Σ:

$R_\Sigma = (\{pop, stop\}, \Sigma \cup \{\varepsilon\}, \Sigma^*, d, l)$;
$d(\sigma_1\sigma_2 \ldots \sigma_n, pop) = \sigma_2\sigma_3 \ldots \sigma_n$; $d(\sigma, pop) = \varepsilon$; $d(\varepsilon, pop) = \perp$;
$l(\sigma_1\sigma_2 \ldots \sigma_n) = \sigma_1$; $l(\varepsilon) = \varepsilon$.

Schreibumwelt über Σ:

$W_\Sigma = (\Sigma \cup \{stop\}, \Sigma \cup \{\varepsilon\}, \Sigma^*, d, l)$;
$d(\sigma_1\sigma_2 \ldots \sigma_n, x) = \sigma_1\sigma_2 \ldots \sigma_n x$; $d(\varepsilon, x) = x$;
$l(\sigma_1\sigma_2 \ldots \sigma_n) = \sigma_n$; $l(\varepsilon) = \varepsilon$.

Das kartesische Produkt von Umwelten

Will man mit den Möglichkeiten der Umwelt $\mathbb{N}$ zwei Zahlen addieren, kann man zwei Speicherplätze benutzen, in denen die Summanden abgelegt sind. Der Inhalt des einen Speichers wird herunter gezählt, solange der Einblick 1 ist. Im anderen Speicher wird simultan hochgezählt, wobei es auf den Einblick gar nicht ankommt.

ZWEI SPEICHER BILDEN EINEN PUNKT

Um das zu symbolisieren, wird in der nachfolgenden Skizze und im weiteren auch bei den Zustandsgraphen für Automaten (vgl. Abbildung 2.5) der Unterstrich verwendet. Bei einer ausführlichen Darstellung müsste man ja für jedem Einblick eine Kante zeichnen. Die werden auf diese Weise zu einer Kante zusammengefasst. In funktionaler oder logischer Programmierung nennt man diesen Gebrauch eines Bezeichners **anonyme** Variable.

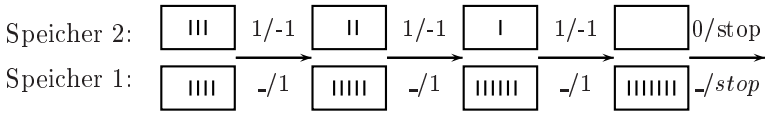


Abbildung 2.5: Zwei Speicherzellen

Ein entsprechender Automat verarbeitet Paare von Einblicken und erzeugt Paare von Aktionen. Er arbeitet in der Umwelt $U = (X, Y, Z, d, l)$ mit

$$Z = \mathbb{N} \times \mathbb{N};\ \ X = \{0, 1, -1\} \times \{0, 1, -1\};\ \ Y = \{0, 1\} \times \{0, 1\}$$

Wenn mit $\mathbb{N} = (X, Y, \mathbb{N}, \tilde{d}, \tilde{l})$ die Zählerumwelt bezeichnet wird, kann man Aktions- und Einblicksfunktion wie folgt beschreiben:

$$\begin{aligned} d(P, x) &= d((m, n), (x_1, x_2)) = (\tilde{d}(m, x_1), \tilde{d}(n, x_2)) \\ l(p) &= l(m, n) = (\tilde{l}(m), \tilde{l}(n)) \end{aligned}$$

Diese Umwelt nennen wir das kartesische Produkt von $\mathbb{N}$ mit sich selbst und bezeichnen sie mit $(\mathbb{N} \times \mathbb{N}, (0, 0))$ oder $(\mathbb{N}^2, (0, 0))$. Der Zustandsgraph des Addierautomaten ist in Abbildung 2.6 dargestellt.

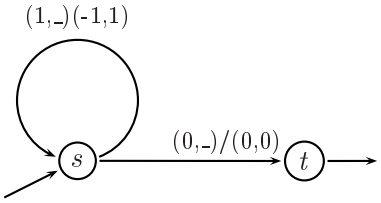


Abbildung 2.6: Addition mit der Umwelt $\mathbb{N}^2$

Diese Art und Weise der Konstruktion neuer Umwelten aus bereits vorhandenen wird verallgemeinert. In der folgenden Definition benutzen wir für das

kartesische Produkt von Mengen $M_1, M_2, \ldots, M_n$ die Schreibweise

$$M_1 \times M_2 \times \ldots \times M_n = \prod_{i=1}^{n} M_i = \{(m_1, m_2, \ldots, m_n) \mid m_i \in M_i \text{ für } 1 \leq i \leq n\}.$$

Für die Elemente der Produktmenge schreiben wir auch

$$m = (m_1, m_2, \ldots, m_n) = (m_i)$$

Eine Methode, Umwelten zu erzeugen

Definition 2.2.5 *Das **kartesische Produkt***

$$U = \prod_{i=1}^{n} U_i = (X, Y, Z, d, l)$$

der Umwelten

$$U_i = (X_i, Y_i, Z_i, d_i, l_i); \quad (1 \leq i \leq n)$$

ist definiert durch

$$X = \prod_{i=1}^{n} X_i; \quad Y = \prod_{i=1}^{n} Y_i; \quad Z = \prod_{i=1}^{n} Z_i;$$

$$l(P) = l(P_1, P_2, \ldots, P_n) := (l_1(P_1), l_2(P_2), \ldots, l_n(P_n));$$

$$d(P, x) = d((P_i), (x_i)) := \begin{cases} \perp, \text{ falls es ein } i \text{ mit } d_i(P_i, x_i) = \perp \text{ gibt,} \\ (d_1(P_1, x_1), d_2(P_2, x_2), \ldots, d_n(P_n, x_n)) \text{ sonst} \end{cases}$$

Für die Produktmenge ergeben sich zwei wichtige Eigenschaften.

Folgerung 2.2.1 *Die Menge der in einem Punkt $P = (P_i)$ möglichen Aktionen ist das kartesische Produkt der von den P_i aus möglichen Aktionen in den Teilumwelten.*

Beweis:
Die Begründung folgt aus der Definition von d und der Valenz eines Punktes:

$$val_U(P) = \{x = (x_i) \mid Px \neq \perp\}$$
$$= \{x = (x_i) \mid P_i x_i \neq \perp \text{ für alle } i\} = \prod_{i=1}^{n} val_{U_i}(P_i)$$

Folgerung 2.2.2 *Sind die Teilumwelten U_i des kartesischen Produktes R_i-Umwelten, und ist $R \subseteq Y \times X$ definiert durch*

$$R(y) = R((y_i)) := \prod_{i=1}^{n} R_i(y_i)$$

für alle $y \in Y$, so ist $U = \prod_{i=1}^{n} U_i$ eine R-Umwelt.

Beweis:

$$R(l(P)) = \prod_{i=1}^{n} R_i(l_i(P_i)) = R_1(l_1(P_1)) \times R_2(l_2(P_2)) \times \ldots \times R_n(l_n(P_n))$$

Da U_i eine R_i-Umwelt ist, folgt $R_i(l_i(P_i)) \subseteq val_{U_i}(P_i)$. Daraus ergibt sich mit Folgerung 2.2.1

$$R(l(P)) \subseteq \prod_{i=1}^{n} val_{U_i}(P_i) = val_U(P)$$

Neben der so definierten Produktumwelt werden wir bisweilen eine andere Konstruktion benutzen, die sich vom kartesischen Produkt allein in der Aktionsmenge X und einer entsprechend veränderten Funktion d unterscheidet.

Definition 2.2.6 *Zum kartesischen Produkt*

$$U = \prod_{i=1}^{n} U_i = (X, Y, Z, d, l)$$

ist das ***eingeschränkte kartesische Produkt***

$$\tilde{U} = (\tilde{X}, Y, Z, \tilde{d}, l)$$

in folgender Weise definiert:

$$\tilde{X} = \bigsqcup_{1 \le i \le n} X_i = \bigcup_{1 \le i \le n} \tilde{X}_i,$$

wobei

$$\tilde{X}_i = \left\{ x = (x_1, x_2, \ldots, x_n) \in \prod_{1 \le j \le n} X_j \mid x_j = stop \text{ für } j \neq i \text{ und } x_i \in X_i \right\}$$

ist.

Für

$$\tilde{x}_i = (stop, \ldots, stop, x_i, stop, \ldots, stop) \in \tilde{X}_i \subseteq \tilde{X}$$

wird definiert

$$\tilde{d}(P, \tilde{x}_i) = \tilde{d}((P_1, P_2, \ldots, P_n), \tilde{x}_i) = (P_1, \ldots, P_{i-1}, P_i x_i, P_{i+1}, \ldots, P_n)$$

Mit anderen Worten: in dieser Umwelt verändert eine Aktion einen Punkt immer nur in einer einzigen Koordinate.

Wichtig ist, dass man immer eine Produktumwelt durch die eingeschränkte Umwelt ersetzen kann und umgekehrt, weil einerseits jede Aktion

$$x = (x_1, x_2, \ldots, x_n) \in X$$

in $U = \prod_{i=1}^{n} U_i$ simuliert werden kann durch die n Aktionen $\tilde{x}_i \in \tilde{X}_i$ und weil andererseits jede Aktion aus $\tilde{X}$ auch aus X ist.

Praktischer Hintergrund der Konstruktion von Produktumwelten ist die Verwendung in einer beliebigen Programmiersprache. Wir werden sehen, dass ein Programm Automat und Umwelt in einem ist, wobei die Umwelt wesentlich durch die verwendeten Variablen repräsentiert wird. Für den eingangs beschriebenen Addierer könnte man z.B. in PASCAL die folgende Funktion verwenden:

```
function add (x, y: Word): Word
 begin
   while x>0 do
    begin
     dec(x);
     inc(y);
    end;
  add := y;
 end;
```

Die beiden Variablen sind mit Zahlen belegt, die hoch- bzw. herunter gezählt werden, repräsentieren mithin die beiden Speicherzellen oder Teilumwelten. Dieses Programm arbeitet in einer Produktumwelt $U \times U$, wobei die Punkte von U Zahlen des Typs Word sind. Welche Zahlenmenge das im Anwendungsfall ist, hängt vom verwendeten Computer ab. Statt $\mathbb{N}$ wird , weil es auf einem Computer gar nicht anders geht, eine endliche Umwelt benutzt. Die beiden Variablen werden nacheinander dekrementiert bzw. inkrementiert. Es werden also Aktionen der eingeschränkten Produktumwelt verwendet.

Produktumwelten werden sich als ein entscheidendes Hilfsmittel für die Untersuchung von Berechnungsprozessen erweisen.

Garbenumwelten

Die verschiedenen Speicher eines Computers - Arbeitsspeicher im Prozessor, das Random-Access-Memory (RAM), externe Speicher wie Festplatten, Disketten, Magnetbänder - kann man als Umwelten interpretieren. Wir wollen ein Umweltmodell für den RAM-Speicher konstruieren. Dazu benutzen wir den algebraischen Begriff der Garbe, weil man damit auch das Turingmodell erfassen kann. Es ist wissenschaftshistorisch interessant, dass die Struktur einer Garbe in der Mathematik in der zweiten Hälfte des 19. Jahrhunderts erdacht und untersucht wurde, ohne dass an eine Anwendung in der Informatik gedacht wurde, obwohl das Turingband längst bekannt war, ein Phänomen, das in der Geschichte der Mathematik und der Naturwissenschaften nicht selten ist.

Garben, eine Anleihe aus der Mathematik

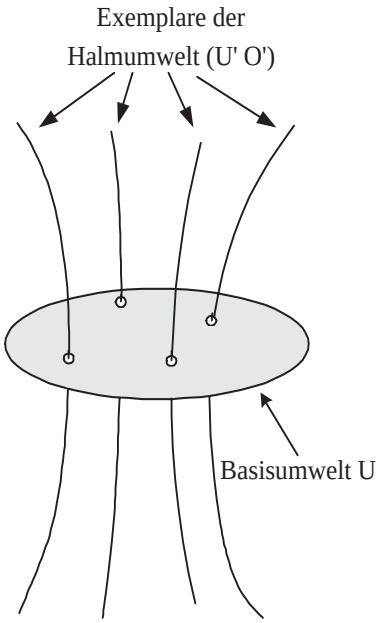


Abbildung 2.7: Garbenumwelten

Wir modifizieren den mathematischen Begriff so, dass er unseren Belangen angepasst ist. Eine Umwelt U, Basis genannt, wird mit einer zweiten punktierten Umwelt in der Weise kombiniert, dass in jedem Punkt von U die zweite Umwelt, der Halm oder die Faser (U', O') angeheftet ist (vgl. Abbildung 2.7). Die unterschiedlichen Exemplare von (U', O') stellt man sich als Halme in einer Garbe vor.

Eine Ameise kann sich in der Garbe so bewegen, dass sie in der zu einem Punkt P der Basisumwelt gehörenden Halmumwelt U' etwas tut, dann in U einen weiteren Punkt Q ermittelt und in der neuen Halmumwelt arbeitet usw.

So werden die Aktionen in der Umwelt definiert: Basis- und Halmoperationen wechseln sich ab.

Mit Blick auf das Speichermodell kann man die Punkte der Basis als Speicheradressen, die Punkte der Halme als Speicherinhalte interpretieren. Unser Speichermodell enthält also im Gegensatz zur Realität unendlich viele Speicherzellen.

DIE GARBENUMWELT

Definition 2.2.7 *Zu einer Umwelt* $U = (X, Y, Z, d, l)$ *und einer punktierten Umwelt* $(U', O') = (X', Y', Z', d', l')$ *wird die Umwelt*

$$U_{(U',O')} = (X^*, Y^*, Z^*, d^*, l^*)$$

wie folgt definiert

- $X^* := X \sqcup X'$

- $Y^* := Y \times Y'$
- $Z^* := Z \times (Z', O')^{(Z)}$, *wobei* $(Z', O')^{(Z)}$ *die Menge der Funktionen von* Z *nach* Z' *ist mit der Eigenschaft, dass supp* $f := \{P \in Z \mid f(P) \neq O'\}$ *eine endliche Menge ist.*
- $d^*((P, f), x) := (d(P, x), f)$ *für alle* $x \in X$
- *Für alle* $x' \in X'$ *ist*

$$d^*((P, f), x') := ((P, f') \text{ mit}$$
$$f'(Q) = \begin{cases} f(Q) & \text{falls } P \neq Q \\ d'(f(P), x') & \text{sonst} \end{cases}$$

- $l^*(P, f) := (l(P), l'(f(P)))$

$U_{(U',O')}$ heißt **Garbenumwelt** mit der Basis U und der **Faser** oder dem **Halm** (U', O'). Das Kürzel *supp* steht für das englische Wort *support* (=Träger). Der Träger von f umfasst diejenigen Speicherplätze, in denen effektiv etwas steht. $f(P) = O'$ wird als leere Belegung des Speichers P interpretiert. Wie in einem realen Rechner wird also immer angenommen, dass nur endlich viele Speicherplätze belegt sind.

Ein Punkt dieser Umwelt ist ein sehr komplexes Gebilde: er enthält als Information eine Speicherplatzadresse und alle Speicherinhalte. Die Sicht in einem Punkt ist dagegen beschränkt auf Informationen zu einem einzelnen Speicherplatz.

Für die in einem Punkte ausführbaren Aktionen ergibt sich

$$val_{U^*}(P, f) = val_U(P) \sqcup val_{U'}(f(P))$$

Jetzt wählen wir für U und (U', O') spezielle Umwelten.

Die RAM-Umwelt: $RAM := \mathbb{N}_{(\mathbb{N},0)}$

Ein Modell des Arbeitsspeichers eines PC

Die Abbildung 2.8 symbolisiert den Punkt $P = (5, f)$ in der RAM-Umwelt. Dabei wird f durch die Paare der Tabelle

n	0	1	2	3	4	5	6	$n > 6$
$f(n)$	7	0	1	5	0	4	3	0

definiert.

In diesem Falle ist $supp\ f = \{0, 2, 3, 5, 6\}$. Der dicke Pfeil in der Abbildung kann als Schreib-Lese-Kopf gedeutet werden, der im Beispiel den Einblick

$$l^*(P) = (1, 1)$$

liefert. Einblicke in einem Punkt dieser Umwelt bestehen aus der Information, ob der Pfeil rechts von der Null oder auf Null steht und ob im Speicher der Pfeilposition eine positive Zahl oder Null enthalten ist. Mehr weiß man nicht.

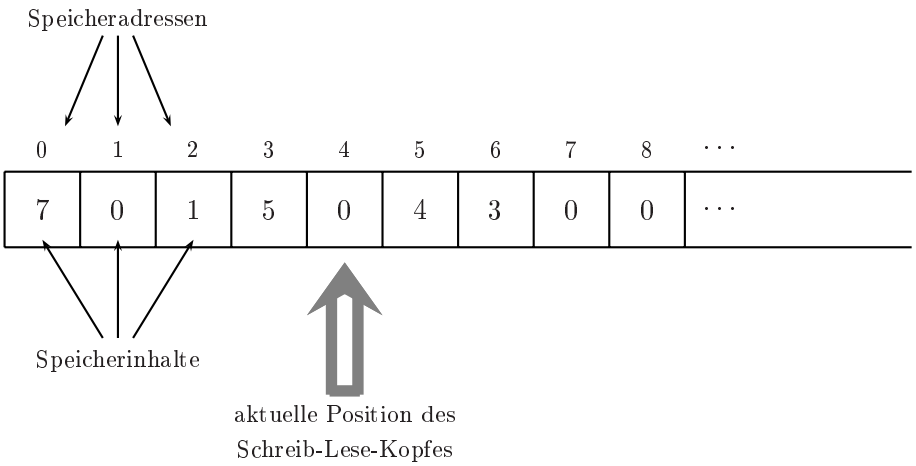


Abbildung 2.8: Ein Punkt der RAM-Umwelt

Die Turing-Umwelt: $T_\Sigma := \mathbb{N}_{(\Sigma,\flat)}$

TURINGS COMPUTERMODELL

Eine andere Spezialisierung der Garbenumwelt führt uns zu dem berühmten Umweltmodell, das Alan Mathison Turing um 1936 für seine theoretischen Arbeiten erdacht hat, ohne natürlich das Wort Umwelt zu benutzen. Abbildung 2.9 zeigt, was man unter einer Turingmaschine versteht. Mit dem Begriff Maschine befassen wir uns ausführlicher im nächsten Abschnitt. Das ist ein Automat zusammen mit einer Umwelt, in welcher er arbeitet.

Turing hatte die Vorstellung, dass der Automat das nach rechts beliebig weit reichende Band schrittweise abtasten kann und in der Lage ist, die auf dem Band abgelegten Buchstaben eines Alphabets zu lesen oder zu überschreiben.

Um die durch das Turingband gegebene Umwelt formal zu beschreiben, definieren wir zu einem beliebigen Alphabet Σ eine (endliche) **punktierte Alphabetumwelt**, die wir ebenfalls mit Σ bezeichnen. Zunächst wird gefordert, dass zum Alphabet Σ immer ein Leerzeichen $\flat$ gehört, welches in der Umwelt ausgezeichneter Punkt ist.

Damit sei

$$(\Sigma, \flat) = (X, Y, Z, d, l)$$

mit

- $X = \Sigma \cup \{stop\}$
- $Y = Z = \Sigma$
- $d(a, b) = b;\ d(a, stop) = a$ für alle $a, b \in \Sigma$
- $l(a) = a$ für alle $a \in \Sigma$

Zu einem beliebigen Alphabet Σ heißt $T_\Sigma := \mathbb{N}_{(\Sigma,\flat)}$ die Turingumwelt über dem Alphabet Σ.

Turing- und RAM-Umwelten kann man auch mit der Basis $\mathbb{Z}$ konstruieren, was bedeutet, dass die Speicherbänder nach beiden Seiten unendlich ausgedehnt sind. Dadurch lassen sich manchmal Automaten leichter konstruieren.

BEISPIEL EINER TURINGMASCHINE

Um anzudeuten, wie sich ein Automat in einer Garbenumwelt bewegt, geben wir die Lösung der folgenden Aufgabe an:

In einer Turingumwelt T_Σ mit $\Sigma = \{\alpha, \beta, \flat\}$ soll ein Automat die "Bandinschrift" um eine Stelle nach links verschieben. Dabei sei angenommen, dass $P = (i, f)$ ein Punkt von T_Σ ist mit $supp\ f = \{0, 1, 2, \dots, n\}$, d.h. auf den ersten n Plätzen des Bandes steht ein Wort $w = f(0)f(1)\dots f(n) \in \{\alpha, \beta\}^*$, und rechts davon sind alle Plätze des Bandes mit Blanks besetzt. Wenn der Automat seine Arbeit beendet hat, soll P übergegangen sein in einen Punkt $Q = (j, f')$ mit $f'(k) = f(k + 1);\ k \geq 0$.

Die Lösung ist in Abbildung 2.10 dargestellt. Der Automat rückt zuerst auf die Bandposition 0. Findet er dort das Blankzeichen, ist die Arbeit getan.

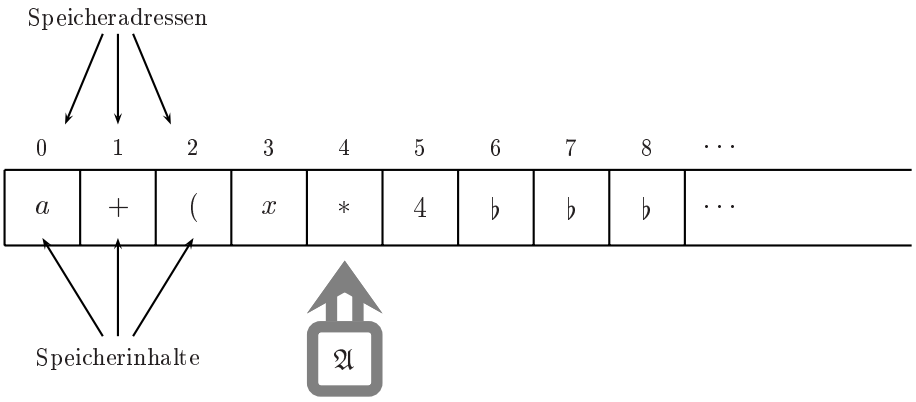


Abbildung 2.9: Turings Modell

Andernfalls rückt er auf Position 1. Je nachdem, ob er dort α, β oder $\flat$ findet, geht er in den Zustand s_α, s_β oder $s_\flat$ und rückt eine Position nach links. Dort schreibt er anschließend das entsprechende Zeichen. Hatte der Automat zuletzt ein $\flat$ gefunden, ist die Arbeit getan. In den anderen beiden Fällen rückt er zwei Plätze nach rechts und wiederholt den Vorgang.

Man erkennt an diesem Beispiel gut, wie das Gedächtnis eines endlichen Automaten arbeitet. Es wird durch die Zustände repräsentiert. Mit s_α, s_β und $s_\flat$ werden Einblicke in der Faser registriert, mit s_2, s_3, s_1 bzw. s_4, s_3, s_1 die Doppelschritte nach rechts. Die Zahl der Zustände hängt von der Zahl der Buchstaben im Alphabet ab.

In der Abbildung 2.10 wird wieder die abkürzende Bezeichnungsweise mit dem Unterstrich als anonymer Variablen benutzt.

Abschließend zeigen wir, wie sich bei beliebigen Garbenumwelten die R-Eigenschaft von Basis und Faser auf die Garbe überträgt.

Zu $R \subseteq Y \times X$ und $R' \subseteq Y' \times X'$ wird eine Relation $R^* \subseteq (Y \times X) \sqcup (Y \times X')$ definiert durch $R^*(y, y') := R(y) \sqcup R'(y')$.

Folgerung 2.2.3 *Ist U bzw. U' eine R- bzw. R'-Umwelt, so ist $U_{(U',O')}$ eine R^*-Umwelt.*

Beweis
Sei $\mathbf{P} = (P, f)$ ein Punkt der Garbe $V = U_{(U',O')}$. Daraus ergibt sich

$$\begin{aligned} R^*(l^*(\mathbf{P})) &= R^*(l^*(P,f)) = R^*(l(P), l'(f(P))) \\ &= R(l(P)) \sqcup R'(l'(f(P))) \subseteq val_U\ (P) \sqcup val_{U'}\ (f(P)) \\ &= val_{U^*}\ (\mathbf{P}) \end{aligned}$$

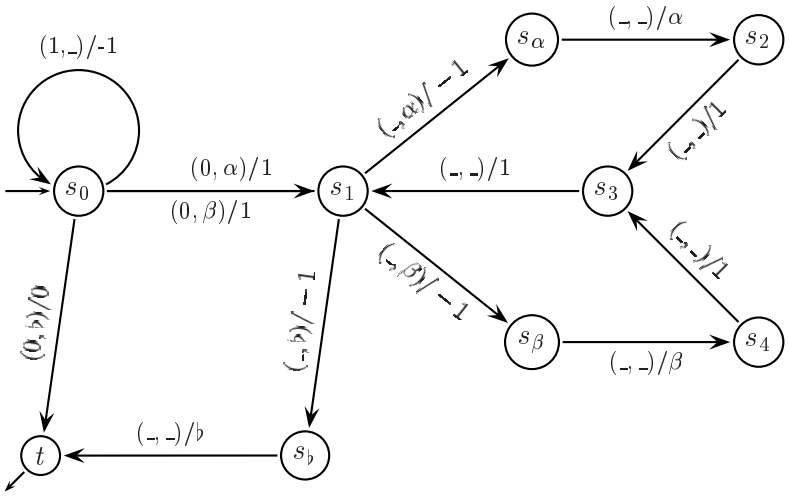


Abbildung 2.10: Eine Bandinschrift wird nach links verschoben

Registerumwelten

Mit diesem Speichermodell wird das Zusammenspiel zwischen den Arbeitsregistern im Prozessor und den Registern im RAM eines Computers simuliert. Über die Aktionen werden großzügigere Voraussetzungen getroffen als bisher: wir kümmern uns nicht mehr um die Details arithmetischer oder boolescher Operationen. Die Operationen selbst sind - in Verbindung mit den Arbeitsregistern - Aktionen. Der Zugriff auf die unendliche Menge der RAM-Register wird durch indirekte Adressierung möglich.

Zum Beispiel

Zunächst ein Beispiel: $\mathfrak{R} = \{p, q, r\}$ seien die Arbeitsregister. Die Menge der RAM-Register soll abzählbar sein. Sie werden deshalb mit natürlichen Zahlen durchnummeriert.

Wir wollen in dieser Umwelt mit natürlichen Zahlen umgehen und nehmen deshalb an, dass jedes Register jede natürliche Zahl speichern kann. Was in den Registern steht, wird durch zwei Funktionen beschrieben, wobei wir wie bei Garbenumwelten annehmen, dass fast alle RAM-Speicherplätze mit Null besetzt sind:

- Belegung der Arbeitsspeicher: $a : \mathfrak{R} \to \mathbb{N}$
- Belegung der RAM-Speicher: $f : \mathbb{N} \to \mathbb{N}$, wobei $supp\ f = \{n \in \mathbb{N} \mid n \neq 0\}$ endlich ist.

Die Mengen von Funktionen dieser Typen werden mit den Symbolen $\mathbb{N}^{\mathfrak{R}}$ bzw. $(\mathbb{N}, 0)^{(\mathbb{N})}$ bezeichnet.

Die zur Verfügung stehenden Operationen seien die Addition und die Subtraktion natürlicher Zahlen:

$$A = \{add, sub\}.$$

Ferner sei es möglich, mittels einer booleschen Funktion eq festzustellen, ob zwei natürliche Zahlen gleich sind oder nicht:

$$\begin{aligned} &B = \{eq\} \\ &eq : \mathbb{N} \times \mathbb{N} \to \mathbb{B} := \{0, 1\}; \\ &\qquad (m, n) \mapsto eq(m, n) := \begin{cases} 1, & \text{falls } m = n \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

Für den Datentransport stehen die Operationen $S = \{lod, mov\}$ zur Verfügung, wobei lod Zahlen aus einer endlichen Menge $N \subseteq \mathbb{N}$ in die Register laden kann. (lod und mov entsprechen den englischen Verben $load$ und $move$. Das ist eine Bezeichnungskonvention der Assemblerprogrammierung.) In diesem Beispiel sei $N = \{1\}$.

Schließlich sei

$$Op = A \cup B \cup S, \quad Op' = Op \setminus \{lod\} \text{ und } R = \mathfrak{R} \cup \{[p], [q], [r]\}.$$

Die Menge $\{[p], [q], [r]\}$ ist zunächst ganz formal gebildet. Ihre Elemente erhalten ihre Bedeutung durch die Definition der Wirkungsfunktion d.

Mit diesen Bezeichnungen wird die Registerumwelt

$$U_{(Op,\mathfrak{R},N)} = \{X, Y, Z, d, l\}$$

folgendermaßen definiert:

$$\begin{aligned} \text{Aktionen}: &\quad X := (Op' \times R \times R) \sqcup (\{lod\} \times R \times N) \\ \text{Einblicke}: &\quad Y := \mathbb{B} \\ \text{Punkte}: &\quad Z := \mathbb{B} \times \mathbb{N}^{\mathfrak{R}} \times (\mathbb{N}, 0)^{(\mathbb{N})} \end{aligned}$$

Ein Punkt $P \in Z$ ist demnach ein Tripel $P = (w, a, f)$, worin a und f die oben beschriebenen Belegungsfunktionen der Register sind. w speichert die mit eq möglichen Testergebnisse.

$$\text{Einblicksfunktion:} \qquad l(P) = l(w, q, f) := w$$

Die Aktionsfunktion d wird in der Tabelle 2.1 angegeben. Wenn sich die Speicherbelegungen a oder f ändern, dann wie bei Garbenumwelten immer nur in einem einzigen Funktionswert. Und nur das ist in der Tabelle vermerkt. Außerdem sind die Aktionen nicht in der üblichen Tripelschreibweise, sondern wie in Assemblersprachen notiert.

Aktion x	**Wirkung** $d(P,x)$ auf $P=(w,a,f)$	**Kommentar**
$eq\ \ u\ \ v$	$w := eq(a(u),a(v))$	$u,v \in \{p,q,r\}$
$eq\ \ [u]\ \ v$	$w := eq(f(a(u)),a(v))$	
$eq\ \ u\ \ [v]$	$w := eq(a(u),f(a(v)))$	
$eq\ \ [u]\ \ [v]$	$w := eq(f(a(u)),f(a(v)))$	
$op\ \ u\ \ v$	$a(u) := op(a(u),a(v))$	$op \in \{add,sub\}$ $u,v \in \{p,q,r\}$
$op\ \ [u]\ \ v$	$f(a(u)) := op(f(a(u)),a(v))$	
$op\ \ u\ \ [v]$	$a(u) := op(a(u),f(a(v)))$	
$op\ \ [u]\ \ [v]$	$f(a(u)) := op(f(a(u)),f(a(v)))$	
$mov\ \ u\ \ v$	$a(u) := a(v)$	$u,v \in \{p,q,r\}$
$mov\ \ [u]\ \ v$	$f(a(u)) := a(v)$	
$mov\ \ u\ \ [v]$	$a(u) := f(a(v))$	
$mov\ \ [u]\ \ [v]$	$f(a(u)) := f(a(v))$	
$lod\ \ u\ \ 1$	$a(u){:=}1$	
$lod\ \ [u]\ \ 1$	$f(a(u)){:=}1$	

Tabelle 2.1: Aktionen und Wirkungen dieses Beispiels

	Umweltpunkte						
Zustand	w	$a(p)$	$a(q)$	$a(r)$	$f(0)$	$f(1)$	$f(2)$
s_0	$\perp$	$\perp$	$\perp$	$\perp$	3	2	$\perp$
$s_1 - s_5$	$\perp$	0	1	2	3	2	0
s_6	0	0	1	2	3	2	0
s_7	0	0	1	2	3	2	0
s_8	0	0	1	2	3	2	3
s_6	0	0	1	2	3	1	3
s_7	0	0	1	2	3	1	3
s_8	0	0	1	2	3	1	6
s_6	0	0	1	2	3	0	6
s_7	1	0	1	2	3	0	6
t	1	0	1	2	3	0	6

Tabelle 2.2: Eine Konfigurationenfolge des Multiplizierautomaten

Mit dieser Definition ist auch die Bedeutung des Symbols $[u]$ festgelegt worden: ist z.B. das Arbeitsregister p mit 3 belegt, so bewirkt die Aktion *lod* $[p]$ 1, dass im RAM-Register Nummer 3 eine eins steht. $[p]$ ist das mit $a(p)$ nummerierte RAM-Register. Das ist die indirekte Adressierung der RAM-Register mit Hilfe der Arbeitsregister.

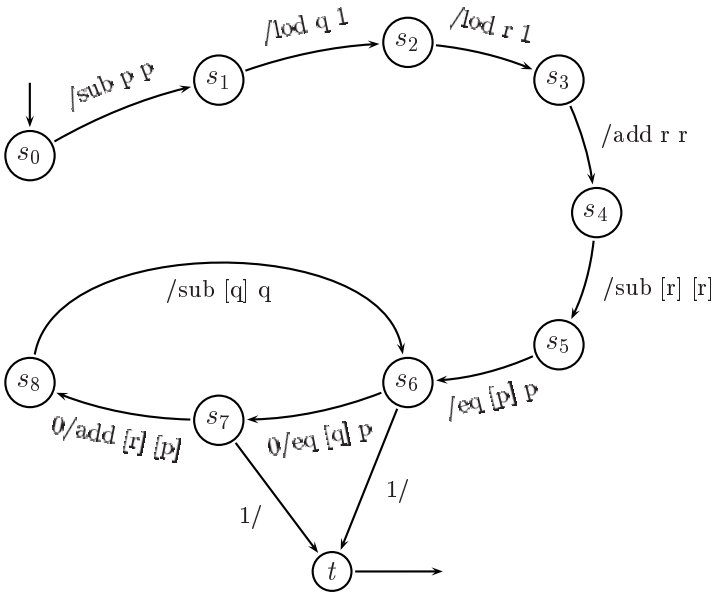


Abbildung 2.11: Ein Multiplizierer

So wird mit dieser Umwelt multipliziert

Die Abbildung 2.11 zeigt den Zustandsgraphen eines Automaten, welcher mit dieser Umwelt die Multiplikation natürlicher Zahlen ausführt. Wenn zu Beginn in den RAM-Registern 0 und 1 die Zahlen m und n stehen, so wird der Automat den terminalen Zustand t erreichen, wenn Register 1 leer ist. Im Register 2 wird dann das Produkt $m \cdot n$ stehen

Aus der Tabelle 2.2 kann man entnehmen, welche Punkte der Registerumwelt passiert werden und welche Zustände der Automat durchläuft, wenn er mit $(m, n) = (3, 2)$ gestartet wird.

Dieser Automat ist etwas komplizierter als die früheren Beispiele, so dass es angebracht ist, die Korrektheit zu beweisen, wobei wir darunter verstehen wollen, dass er bei beliebigen Startwerten für m und n das Produkt richtig ausgibt.

Eine bewährte Methodik für Korrektheitsbeweise besteht darin, Invarianten des Verfahrens zu suchen. Das sind arithmetische Ausdrücke, deren Werte sich im Prozeßverlauf nicht ändern.

Ein Blick in die Tabelle lässt vermuten, dass

$$z := f(0) \cdot f(1) + f(2)$$

diese Eigenschaft hat.

In der Tat geht bei einem Zyklus von s_6 über s_7 und s_8 zurück nach s_6 der Term z über in

$$\tilde{z} = f(0) \cdot (f(1) - 1) + f(2) + f(0) = z.$$

Sind z_a und z_e die Werte von z am Anfang und am Ende des Prozesses, ergibt sich

$$z = z_a = m \cdot n + 0 = z_e = 0 + f(2).$$

Das bestätigt unsere Erwartung.

Spätestens bei diesem Beispiel leuchtet ein, dass die grafische Darstellung von Automaten ungeeignet ist, um komplexe Sachverhalte darzustellen. Wir werden deshalb eine formale Sprache einführen. Dazu bedarf es aber noch einiger Vorbereitungen.

Nach dem Vorbild des Beispiels folgt die Definition von Registerumwelten.

REGISTER-UMWELTEN FÜR NATÜRLICHE ZAHLEN

Definition 2.2.8 *Gegeben seien die Mengen arithmetischer bzw. boolescher Operationen*

$$\begin{array}{lll} A = \{a_1, a_2, \ldots, a_p\} & \textit{mit} & a_i : \mathbb{N} \to \mathbb{N} \\ & \textit{oder} & a_i : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \textit{ und} \\ B = \{b_1, b_2, \ldots, b_q\} & \textit{mit} & b_i : \mathbb{N} \to \{0,1\} \\ & \textit{oder} & a_i : \mathbb{N} \times \mathbb{N} \to \{0,1\} \end{array}$$

sowie die Menge $\mathfrak{R} = \{r_0, r_1, \ldots, r_s\}$ der Arbeitsregister. Ferner seien

$$\begin{array}{ll} [\mathfrak{R}] := \{[r_0], [r_1], \ldots, [r_s]\}; & R := \mathfrak{R} \sqcup [\mathfrak{R}]; \\ Op' := A \sqcup B \sqcup \{mov\}; & Op := Op' \sqcup \{lod\}. \end{array}$$

N sei eine endliche Menge natürlicher Zahlen.

Damit wird die Registerumwelt $U_{(Op,\mathfrak{R},N)} = (X, Y, Z, d, l)$ in folgender Weise definiert:

- $X := (Op' \times R \times R) \sqcup (\{lod\} \times R \times N)$
- $Y := \{0,1\} = \mathbb{B}$
- $Z := \mathbb{B} \times \mathbb{N}^{\mathfrak{R}} \times (\mathbb{N}, 0)^{(\mathbb{N})}$
- $l(P) = l(w, a, f) := w$

In der Tabelle 2.3 für die Wirkungsfunktion d ist die Funktion η wie folgt definiert:

$$\eta(\alpha) = \begin{cases} a(\alpha) & \textit{für } \alpha \in \mathfrak{R} \\ f(a(p)) & \textit{für } \alpha = [p] \in [\mathfrak{R}] \end{cases}$$

Die Aktionen und die Wirkungsfuntion der Umwelt werden wie im Beispiel mit der Tabelle 2.3 simultan definiert.

Aktion x	$d(P,x)$ für $P=(w,a,f)$	Kommentar
$b\ \alpha\ \beta$	$w := b(\eta(\alpha),\eta(\beta))$	$b \in B,\ b : \mathbb{N}\times\mathbb{N}\to\mathbb{B}$
$b\ \alpha$	$w := b(\eta(\alpha))$	$b \in B,\ b : \mathbb{N}\to\mathbb{B}$
$op\ \alpha\ \beta$	$\eta(\alpha) := op(\eta(\alpha),\eta(\beta))$	$op \in A,\ op : \mathbb{N}\times\mathbb{N}\to\mathbb{N}$
$op\ \alpha$	$\eta(\alpha) := op(\eta(\alpha))$	$op \in A,\ op : \mathbb{N}\to\mathbb{N}$
$mov\ \alpha\ \beta$	$\eta(\alpha) := \eta(\beta)$	
$lod\ \alpha\ n$	$\eta(\alpha) := n$	$n \in N$

Tabelle 2.3: Aktionen und Wirkungen in einer Registerumwelt

Das Beispiel des Multiplizierautomaten macht deutlich, wie bequem es ist, genau so viele Arbeitsregister zu benutzen, wie man RAM-Register adressieren möchte. Es ist jedoch von grundsätzlicher theoretischer Bedeutung, dass man immer mit nur zwei Arbeitsregistern auskommt. Auf den ersten Blick scheint es ja so zu sein, dass man mit Registerumwelten um so mehr berechnen kann, je mehr Arbeitsregister man zur Verfügung hat. Dass dem nicht so ist, beweisen wir. Dabei wird sich allerdings herausstellen, dass die Zahl der Rechenschritte stark anwächst. Die Beschränkung auf zwei Arbeitsregister verlängert die Rechenzeit.

Satz 2.2.1 *Zu jeder Registerumwelt $U_1 = U_{(Op_1,\mathfrak{R}_1,N_1)}$ gibt es eine Registerumwelt $U_2 = U_{(Op_2,\mathfrak{R}_2,N_2)}$ mit nur zwei Arbeitsregistern, in welcher man die Aktionen von U_1 durch Automaten simulieren kann.*

Beweis: Seien $\mathfrak{R}_1 = \{r_0, r_1, \ldots, r_{s-1}\}$ und $\mathfrak{R}_2 = \{p, q\}$ die Arbeitsregister beider Umwelten. Wenn $P = (w, a, f)$ ein Punkt von U_1 ist, so entspricht dem jeder Punkt $Q = (w, \alpha, \varphi)$ der Umwelt U_2, in dessen RAM-Registern von 0 bis $s-1$ die Inhalte der Arbeitsregister von U_1 stehen. Die Inhalte der RAM-Register von U_1 befinden sich in den RAM-Registern von U_2 ab der Nummer s:

$$\varphi(i) := \begin{cases} a(r_i) & \text{falls } 0 \le i < s \\ f(i-s) & \text{wenn } i \ge s \end{cases}$$

Da über die Belegung der Arbeitsregister (R_2) keine Aussage gemacht wird, entsprechen jedem Punkt von U_1 unendlich viele Punkte von U_2.

Diese Relation zwischen den Punkten beider Umwelten nennen wir **Überdeckung** der Umwelten, sagen, dass ein Punkte P von einem Punkt Q überdeckt wird und schreiben $P \prec Q$.

Für U_2 sind noch die folgenden Festlegungen zu treffen:

$$\begin{aligned} N_2 &:= N_1 \cup \{1, 2, \ldots, s-1\} \\ Op_2 &:= Op_1 \cup \{inc\} \\ &\text{mit } inc : \mathbb{N} \to \mathbb{N} \\ &\text{inc r bewirkt } \alpha(r) := \alpha(r) + 1 \\ &\text{inc [r] bewirkt } \varphi(\alpha(r)) := \varphi(\alpha(r)) + 1 \end{aligned}$$

Aktionen	Registerbelegungen
$lod\ p\ i$	$\alpha(p) := i$
$mov\ p\ [p]$	$\alpha(p) := \varphi_1(\alpha(p)) = \varphi_1(i)$
$inc^s\ p$	$\alpha(p) := \alpha(p) + s = \varphi_1(i) + s$
$lod\ q\ j$	$\alpha(q) := j$
$mov\ q\ [q]$	$\alpha(q) := \varphi(j)$
$op\ [p]\ q$	$\varphi_2(\alpha(p)) := op(\varphi_1(\alpha(p)), \alpha(q))$ $= op(\varphi_1(i) + s, \varphi(j))$

Tabelle 2.4: Simulation der Aktion $x = op\ [r_i]\ r_j$

Soll die Inkrementierung inc s-mal ausgeführt werden, schreiben wir $inc^s\ r$.

Nun soll an einem Beispiel ausgeführt werden, wie die Simulation erreicht wird.

Die Aktion $x = op\ [r_i]\ r_j$ überführt einen Punkt $P_1 = (w, a, f_1) \in U_1$ in

$$P_2 = (w, a, f_2) \in U_1 \text{ mit } f_2(n) = \begin{cases} f_1(n) & \text{falls } n \neq a(r_i) \\ op(f_1(a(r_i)), a(r_j)) & \text{für } n = a(r_i) \end{cases}$$

P_1 wird überdeckt von $Q_1 = (w, \alpha_1, \varphi_1)$ wie oben definiert. In U_2 sind Aktionen anzugeben, welche Q_1 überführen in $Q_2 = (w, \alpha_2, \varphi_2)$ solcherart, dass $P_2 \prec Q_2$ gilt. Das bedeutet

$$\varphi_2(n) = \varphi_1(n) \text{ für } n \neq a(r_i) + s$$

und

$$\begin{aligned} \varphi_2(a(r_i) + s) &= f_2(a(r_i)) = op(\varphi_1(a(r_i) + s), \varphi_1(j)) \\ &= op(\varphi_1(\varphi_1(i) + s), \varphi_1(j)) \end{aligned}$$

Wie die Belegung der Arbeitsregister p und q vor und nach der Simulation aussehen, d.h. die Funktionen α_1 und α_2, ist ohne Belang.

Die Überführung der Punkte in der Umwelt U_2 geht aus der Tabelle 2.4 hervor.

Aufgaben

1. In Anlehnung an das nicht lösbare Problem der Parkettierung eines (nicht ganz vollständigen) Schachbrettes mit Dominosteinen ist die folgende Aufgabe zu untersuchen:

   In einem Würfel der Kantenlänge $3d$ sind 27 gleich große Kugeln mit dem Durchmesser d untergebracht. Man ermittle einen Streckenzug, welcher, ausgehend von der innersten Kugel, derart durch alle Kugeln führt, dass jede Kugel genau einmal durchlaufen wird und der Übergang von einer Kugel zur nächsten nur an den Berührungspunkten der Kugeln erfolgt! (Unter

einem Streckenzug soll eine Folge $(s_1, s_2, \ldots, s_n)$ gerichteter Strecken verstanden werden, wobei der Endpunkt der Strecke s_i gleich dem Anfangspunkt von s_{i+1} für $1 \leq i < n$ ist.)

2. (a) Es ist ein boolescher Automat anzugeben, der in $\mathbb{N}$ die Funktion

$$\begin{aligned} is\_zero : \mathbb{N} &\to \{0,1\} \\ n &\mapsto is\_zero(n) = \begin{cases} 1, \text{ wenn } n = 0 \\ 0 \text{ sonst} \end{cases} \end{aligned}$$

berechnet!

Wenn der Automat in einem Punkt $n \in \mathbb{N}$ startet, soll der Funktionswert $is\_zero(n)$ als Endmarke ausgegeben werden!

(b) Zu jeder Umwelt $U = (X, Y, Z, d, l)$ gibt es einen Automaten $look_U$, der die Werte der Funktion l als Endmarken liefert. Wie sieht dessen Zustandsdiagramm aus?

(c) Wie sieht der Zustandsgraph eines Automaten aus, welcher in $\mathbb{N}$ die Successorfunktion

$$\begin{aligned} succ : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto succ(n) = n + 1 \end{aligned}$$

berechnet? (Berechnen heißt, dass, wenn er im Punkte $n \in \mathbb{N}$ startet, er genau dann den Endzustand erreicht hat, wenn n in $n+1$ überführt ist.)

(d) Zu jeder Umwelt $U = (X, Y, Z, d, l)$ und zu jeder Aktion $x \in X$ gibt es einen Aktor $action_U(x)$, welcher bei Start in $P \in Z$ seine Arbeit beendet, sobald er im Punkt $d(P, x)$ angelangt ist. Mit anderen Worten, $action_U(x)$ berechnet die Funktion

$$\begin{aligned} d_x : Z &\to Z \\ P &\mapsto d_x(P) = d(P, x) \end{aligned}$$

3. Analog zu 2a ist ein boolescher Prozessor zu definierten, welcher mit der Umwelt $\mathbb{N} \times \mathbb{N}$ die Funktion

$$\begin{aligned} less\_then : \mathbb{N} \times \mathbb{N} &\to \{0,1\} \\ (m,n) &\mapsto less\_then(m,n) = \begin{cases} 1, \text{ wenn } m < n \\ 0 \text{ sonst} \end{cases} \end{aligned}$$

berechnet!

4. Es sind die Zustandsgraphen zweier Aktoren anzugeben, welche mit der Umwelt $\mathbb{N}^3$ das Maximum bzw. das Minimum zweier natürlicher Zahlen

ermitteln! Bei Start in einem Punkt $(m, n, 0) \in \mathbb{N}^3$ sollen die Prozessoren die Arbeit beenden, wenn $(m', n', max(m, n))$ bzw. $(m', n', min(m, n))$ erreicht ist, wobei

$$\begin{aligned} max(m,n) &:= \begin{cases} m \text{ falls } m \geq n \\ n \text{ sonst} \end{cases} \quad \text{und} \\ min(m,n) &:= m + n - max(m,n) \end{aligned}$$

bedeuten!

5. Die folgenden Funktionen sind jeweils mit einer Umwelt $\mathbb{N}^k$ bei geeignet gewähltem k in Anlehnung an Aufgabe 4 zu berechnen:

   (a) $sub : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $sub(m, n) = \begin{cases} m - n \text{ falls } m > n \\ 0 \text{ sonst} \end{cases}$

   (b) $mult : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $mult(m, n) = m \cdot n$

   (c) $div : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $div(m, n) = max\{q \in \mathbb{N} \mid q \cdot n \leq m\}$

   (d) $fak : \mathbb{N} \to \mathbb{N}$ mit $fak(n) = n! = \begin{cases} 1, \text{ falls } n = 0 \\ n \cdot fak(n-1) \text{ sonst} \end{cases}$

6. Sei R_Σ die Leseumwelt über dem Alphabet

   $$\Sigma = \{(,), a, b, \ldots, z, +, -, *, \}$$

   und $\mathbb{N}$ die Umwelt der natürlichen Zahlen. Es ist ein boolescher Prozessor anzugeben, der in der Umwelt $R_\Sigma \times \mathbb{N}$ arbeitet und ermittelt, ob bei einem mit den Buchstaben des Alphabets gebildeten Wort die Anzahlen der öffnenden und schließenden Klammern übereinstimmen!

7. Sei R_Σ ein beliebiges Alphabet und $f : \Sigma^* \to \Sigma^*$ die Funktion, welche jedem Wort

   $$w = w_1 w_2 \ldots w_n \in \Sigma^*$$

   das invertierte Wort

   $$f(w) = w_n w_{n-1} \ldots w_1$$

   zuordnet. Diese Funktion lässt sich mit der Umwelt $R_\Sigma \times S_\Sigma \times W_\Sigma$ berechnen!

8. Wie kann man mit Stackumwelten über dem Alphabet $\{0, 1\}$ natürliche Zahlen addieren?

9. Es ist ein Aktor anzugeben, welcher die Inschrift eines Turingbandes – wenn ein Alphabet mit drei Buchstaben und dem Blank benutzt wird – um eine Stelle nach rechts verschiebt!

d'	0	re	li
n	n	e	w
e	e	s	n
s	s	w	e
w	w	n	s

Tabelle 2.5: Die Wirkungsfunktion der Umwelt U' in Aufgabe 11

10. Wie sieht ein Aktor aus, welcher die Fibonacci-Zahlen mit einer Registerumwelt berechnet?

    Die Fibonacci-Zahlen sind die Werte der Funktion

$$f : \mathbb{N} \to \mathbb{N} \text{ mit } f(n) = \begin{cases} n, \text{ falls } n < 2 \\ f(n_1) + f(n_2) \text{ sonst} \end{cases}$$

11. Zwei Umwelten U und U' sind wie folgt definiert:

$$U = (X, Y, Z, d, l) \text{ mit } M \subseteq \mathbb{Z} \times \mathbb{Z} \text{ und } Z = \mathbb{Z} \times \mathbb{Z} \setminus M$$
$$X = \{(1,0)(-1,0),(0,1),(0,-1),(0,0)\}$$
$$Y = 2^X = \{A \mid A \subseteq X\}$$
$$d(P,x) = d((z_1,x_1),(z_2,x_2)) := \begin{cases} \tilde{P} = (z_1 + x_1, z_2 + x_2), \text{ falls } \tilde{P} \notin M \\ \perp \text{ sonst} \end{cases}$$
$$l(P) = \{x \in X \mid d(P,x) \neq \perp\}$$

$$(U', O') = (X', Y', Z', d', l') \text{ mit}$$
$$Z' = \{n, e, s, w\} = Y'$$
$$X' = \{re, li, 0\}$$
$$l'(P') = P'$$

    Die Funktion d' wird durch die Tabelle 2.5 definiert

    Diese Umwelten interpretieren wir in folgender Weise:

    Die Punkte von U sind diskrete Positionen in einer Ebene. Wir stellen uns vor, dass die Ebene mit einem quadratischen Gitter überzogen ist, und jede Masche, also jedes Quadrat ist ein Punkt. Die Paare $P = (z_1, z_2)$ sind die Koordinaten der Mittelpunkte der Quadrate. In dieser Ebene gibt es ein Hindernis M, eine Mauer, deren Punkte nicht zugänglich sind. Das wird durch die Funktion d und die Einblicke modelliert: Aktionen sind Bewegungen in die vier Himmelsrichtungen. Die Paare $(1,0)$, $(-1,0)$, $(0,1)$ und $(0,-1)$ entsprechen in dieser Reihenfolge Bewegungen nach Osten, Westen, Norden und Süden. $(0,0)$ ist die Stopp-Aktion. Der Einblick $l(P)$ ist die Menge aller Himmelsrichtungen, in denen das Nachbarquadrat nicht zur Mauer gehört. Wenn das Nachbarquadrat in Richtung x zu M gehört, ist $d(P,x)$ nicht definiert.

Die endliche Umwelt U' besteht nur aus den vier Himmelsrichtungen. Aktionen sind Rechts- und Linksdrehungen um 90 Grad. Die Null bezeichnet die Stoppaktion.

Die Aufgabe besteht darin, aus U und U' eine Umwelt und dazu einen Automaten zu konstruieren, der sich an der Mauer entlang tastet und sie einmal umrundet, sofern das möglich ist!

(Hinweis: Bei der Analyse der Aufgabe wird man bemerken, dass $U \times U'$ noch nicht geeignet ist, um die Aufgabe zu lösen.)

12. Folgerung 2.1.2 auf Seite 28 ist zu beweisen!

13. Der Beweis des Satzes über die Möglichkeit der Einschränkung auf zwei Arbeitsspeicher in Registerumwelten ist zu vervollständigen!

2.3 Maschinen

In diesem Kapitel ist es unser Anliegen, das Zusammenwirken eines Automaten $\mathfrak{A}$ mit einer Umwelt U systematisch zu untersuchen. Ein Paar $(U, \mathfrak{A})$ wird **Maschine** genannt.

2.3.1 Die operationale Semantik eines Automaten

Wenn ein Prozessor $\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ mit dem Einblick $y_0 = l(P_0)$ eines Punktes P_0 einer Umwelt $U = (X, Y, Z, d, l)$ gestartet wird, so entwickelt sich eine wechselseitige Beeinflussung, ein Bewegungsablauf. Der Automat erlangt in seiner Umwelt ein Eigenleben. Die Maschine wird zu einem dynamischen System, welches durch die vier Funktionen d, l, δ und λ gesteuert wird.

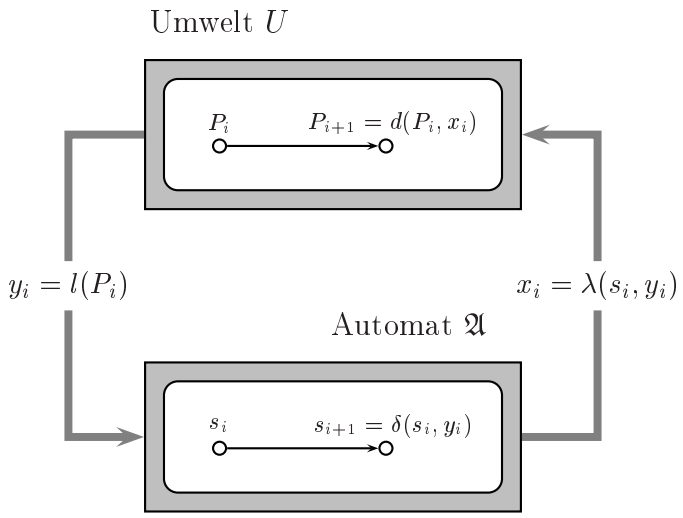


Abbildung 2.12: Eine Maschine

Das Zusammenspiel von Umwelt und Automat

Im null-ten Takt empfängt $\mathfrak{A}$ die Eingabe $y_0 = l(P_0)$, erzeugt die Aktion $x_0 = \lambda(s_0, y_0)$ und geht in den Zustand $s_1 = \delta(s_0, y_0)$ über. In der Umwelt bewirkt die Aktion x_0 den Übergang von P_0 zu $P_1 = P_0 x_0$. Der Automat empfängt den neuen Einblick $y_1 = l(P_1)$, gibt die Aktion $x_1 = \lambda(s_1, y_1)$ aus und erreicht den Zustand $s_2 = \delta(s_1, y_1)$.

Es entsteht eine Folge (v_i) von Quadrupeln $v_i = (P_i, y_i, s_i, x_i)$. Diese Folge ist endlich, wenn irgendwann ein terminaler Zustand erreicht wird oder eine Aktion x_i mit $d(P_i, x_i) = \perp$ ausgegeben wird. Andernfalls entsteht eine unendliche Folge.

Diese drei Fälle entsprechen der täglichen Erfahrung eines Programmierers: ein Programm liefert entweder ein wohlbestimmtes Ergebnis, oder der Computer "stürzt ab" und gibt eine Fehlermeldung aus, oder er hört nicht auf zu arbeiten

und muss künstlich angehalten werden, etwa, weil das Programm sich "in einer Schleife aufgehängt hat".

Die Folge (v_i) ist für eine Maschine $(U, \mathfrak{A})$ wohlbestimmt, sobald der Startpunkt P_0 vorgegeben ist. Wenn $\mathfrak{A}$ einen terminalen Zustand erreicht, ist damit auch der Umweltpunkt P_e, in dem der Automat seine Arbeit beendet, allein durch den Startpunkt P_0 festgelegt. Für diesen Punkt werden wir - in Analogie zu der Schreibweise $P_{i+1} = P_i x_i$ - die Bezeichnung $P_e = P_0\mathfrak{A}$ benutzen.

Wir stellen also fest, dass die aus einem Automaten $\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ und einer Umwelt $U = (X, Y, Z, d, l)$ bestehenden Maschine eine partielle Funktion

$$\begin{aligned} \xi_{(\mathfrak{A},U)} : Z &\xrightarrow{\supset} Z \\ P &\mapsto \xi_{(\mathfrak{A},U)}(P) = P\mathfrak{A} \end{aligned}$$

definiert.

Für die Marke, welche $\mathfrak{A}$ nach Beendigung seiner Arbeit ausgibt, schreiben wir $\mathfrak{A}(P_0)$.

Im allgemeinen wünscht man sich, dass der Automat seine Arbeit kontrolliert beendet, d.h. einen terminalen Zustand erreicht. Um nicht ausführbare Aktionen auszuschließen, wird der Begriff R-Maschine eingeführt.

Definition 2.3.1 *Eine Maschine $(U, \mathfrak{A})$ heißt **R-Maschine**, wenn U eine R-Umwelt ist und für den Automaten $\mathfrak{A}$ gilt:*

$$\lambda(s, y) \in R(y) \text{ für alle } (s, y) \in act\,\mathfrak{A} \times Y$$

*$\mathfrak{A}$ heißt dann auch **R-Automat**.*

Folgerung 2.3.1 *Bei R-Maschinen ist (v_i) entweder unendlich oder endet mit einem terminalen Zustand.*

Wie fassen zusammen und führen noch einige wichtige Begriffe ein.

ZWECKMÄSSIGE BEGRIFFE

Definition 2.3.2 *$M = (U, \mathfrak{A})$ sei eine Maschine.*

- *Die Folge (v_i) mit*

  $$\begin{aligned} v_0 &= (P_0, l(P_0), s_0, \lambda(s_0, l(P_0))) \text{ und} \\ v_i &= (P_i, y_i, s_i, x_i) := (d(P_{i-1}, x_{i-1}), l(P_i), \delta(s_{i-1}, y_{i-1}), \lambda(s_i, y_i)) \end{aligned}$$

  *heißt das **Verhalten von $\mathfrak{A}$ in U bei Start in P_0**.*

- *Die Menge aller Verhaltensfolgen heißt **operationale Semantik von $\mathfrak{A}$ in U**.*

- *Die Folge (m_i) mit $m_i = (y_i, s_i, x_i)$ als Bestandteil von v_i ist **das Modell von U in $\mathfrak{A}$**.*

- (w_i) *mit* $w_i = (P_i, x_i)$ *ist* ***der Weg von*** $\mathfrak{A}$ ***in*** U ***bei Start in*** P_0.
- (l_i) *mit* $l_i = (y_i, x_i)$ *nennen wir* ***Lauf von*** $\mathfrak{A}$ ***in*** U ***bei Start in*** P_0.
- (r_i) *mit* $r_i = (P_i, s_i)$ *heißt* ***Rechnung von*** $\mathfrak{A}$ ***in*** U ***bei Start in*** P_0.

EINBLICK MUSS MAN HABEN

Das Modell, welches ein Automat von seiner Umwelt hat, erfasst die Informationen, welche allein dem Automaten zur Verfügung stehen. Er kennt zwar die Einblicke in die Umwelt, die Punkte selbst aber nicht. In der Turingumwelt z.B. weiß der Automat, welcher Buchstabe auf der aktuellen Bandposition, sozusagen "direkt vor seiner Nase" steht, über die Belegung der Nachbarfelder weiß er nichts.

Das folgende Beispiel macht deutlich, dass es belangvoll ist, zwischen Verhalten und Modell zu unterscheiden. Wir betrachten die beiden Umwelten

$$U_i = (X, Y, Z_i, d_i, l_i) \qquad (i = 1, 2)$$
$$Z_1 = \{0, 1\} \qquad Z_2 = \mathbb{Z}$$
$$X = \{0, 1, -1\} \qquad Y = \{*\}$$
$$l_1(P) = l_2(P) = * \text{ für alle } P \text{ aus } Z_1 \text{ bzw. } Z_2$$
$$d_1(0,0) = d_1(1,1) = d_1(1,-1) = 0 \qquad d_1(0,1) = d_1(0,-1) = d_1(1,0) = 1$$
$$d_2(z,x) = z + x \text{ für alle } x \in X \text{ und alle } z \in \mathbb{Z}$$

Ein beliebiger Automat hat von beiden Umwelten das gleiche Modell. Der Grund ist, dass der Einblick in beiden Umwelten für alle Punkte nur ein einziger Wert ist. Der Automat ist praktisch blind.

Der folgende Satz verallgemeinert das Beispiel:

Satz 2.3.1 *Seien* $((U_1, O_1), \mathfrak{A})$ *und* $((U_2, O_2), \mathfrak{A})$ *R-Maschinen mit derselben Relation* $R \subseteq Y \times X$. *Die beiden folgenden Aussagen sind äquivalent, d.h. die eine gilt genau dann, wenn auch die andere wahr ist:*

1. *Der Lauf von* $\mathfrak{A}$ *in* U_1 *bei Start in* O_1 *stimmt mit dem Lauf von* $\mathfrak{A}$ *in* U_2 *bei Start in* O_2 *überein.*
2. *Das Modell von* U_1 *in* $\mathfrak{A}$ *bei Start in* O_1 *stimmt mit dem Modell von* U_2 *in* $\mathfrak{A}$ *bei Start in* O_2 *überein.*

Mit anderen Worten, wenn der Lauf eines Prozessors in beiden Umwelten der gleiche ist, kann er sie nicht mehr unterscheiden. Der Beweis wird als Aufgabe empfohlen.

2.3.2 Eine Maschine ist ein dynamisches System

Wir kommen noch einmal auf den oben intuitiv benutzten Begriff "dynamisches System" zurück. Er kommt aus der Physik und kann definiert werden als ein Tripel (X, f, x_0), wo X eine beliebige Menge ist, $x_0 \in X$ ein initiales oder Startelement und $f : X \to X$ eine beliebige Funktion oder sogar, und das wird uns im Zusammenhang mit Maschinen angehen, eine partielle Funktion $f : X \xrightarrow{\supset} X$ ist. Dei Elemente $x \in X$ werden **Konfigurationen** genannt. Die Dynamik des Systems wird beschrieben durch die Folge

$$x_0, x_1 = f(x_0), x_2 = f(f(x_0)) = f^2(x_0), \ldots, x_i = f^i(x_0), \ldots$$

Dynamische Systeme, eine Anleihe aus der Physik

Die Figur 2.13 zeigt, wie so etwas aussehen kann und welche Fälle denkbar sind. Bei Start in x_1 ergibt sich eine unendliche Folge, die aber nur endlich viele Konfigurationen enthält. Wird das System in x_1 gestartet, hält es in x_3 an, weil dort f nicht definiert ist. Schließlich ist es, wenn X eine unendliche Menge ist, auch möglich, dass sich immer neue Funktionswerte bilden lassen: die Folge besteht aus unendlich vielen Konfigurationen, von denen keine zwei überein stimmen. Bei der Verhaltensfolge einer Maschine hatten wir eine ähnliche Fallunterscheidung beschrieben.

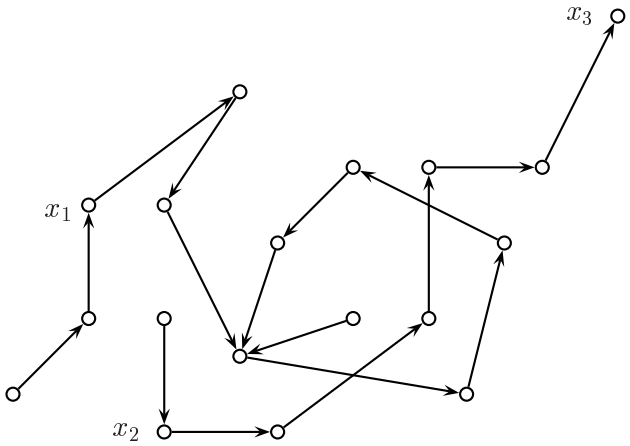


Abbildung 2.13: Punktfolgen eines dynamischen Systems

In der Mathematik(4.3 auf Seite 152) wird so etwas einsortige Algebra genannt. Und man bedient sich des Begriffs der Homomorphie, um strukturelle Verwandschaft zwischen Algebren im allgemeinen bzw. dynamischen System im besonderen zu beschreiben. Darauf kommen wir zurück. Zunächst ist aber zu klären, inwiefern eine Maschine ein partielles dynamisches System ist.

Wenn $\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ ein Prozessor ist, der in einer Umwelt $U = (X, Y, Z, d, l)$ arbeiten kann, dann kann man $Z \times S$ als die im allgemeinen unendliche Menge der Konfigurationen des dynamischen Systems ansehen, und die Funktion, welche die Dynamik erzeugt, ist

$$\begin{array}{rl} h : Z \times S & \stackrel{\supset}{\longrightarrow} Z \times S \\ (P, s) & \mapsto h(P, s) := (d(P, \lambda(s, l(P))), \delta(s, l(P))) \end{array} \tag{2.2}$$

Ist $O \in Z$ ein ausgezeichneter Punkt der Umwelt, wird (s_0, O) zur Startkonfiguration des Systems.

Die oben angedeuteten unterschiedliche Fälle von Folgen in einem dynamischen System kann man bei R-Maschinen sehr präzise eingrenzen:

Folgerung 2.3.2 *Das von einer R-Maschine gebildete dynamische System hält genau dann an, wenn der Prozessor einen terminalen Zustand erreicht:*

$$dom\ h = Z \times act\ \mathfrak{A}$$

Beweis:

Ist $(P, s) \in dom\ h$, so existiert auch $\delta(s, l(P))$. Also ist $s \in act\ \mathfrak{A}$, $(P, s) \in Z \times act\ \mathfrak{A}$ und $dom\ h \subseteq Z \times act\ \mathfrak{A}$.

Falls andererseits $(P, s) \in Z \times act\ \mathfrak{A}$ ist, existieren $\delta(s, l(P))$ und $\lambda(s, l(P))$. Da es sich um eine R-Maschine handelt, ist

$$x = \lambda(s, l(P)) \in val\ P,$$

also existiert auch $d(P, x)$, und damit ist $(P, s) \in dom\ h$. Aus der wechselseitigen Inklusion beider Mengen ergibt sich ihre Gleichheit.

Je stärker die Umwelt, umso einfacher der Automat

Mit Blick auf einen wichtigen Satz, der in diesem Abschnitt bewiesen werden soll, wollen wir an einem Beispiel Beziehungen zwischen Maschinen untersuchen.

Die Maschine $M_1 = (U_1, \mathfrak{A}_1)$ benutzt den Prozessor, welcher durch den Zustandsgraphen der Abbildung 2.1 auf Seite 21 dargestellt ist und als Umwelt die natürlichen Zahlen: $U_1 = \mathbb{N}$

Der Prozessor einer zweiten Maschine $M_1 = (U_2, \mathfrak{A}_2)$ wird durch die Abbildung 2.14 definiert. Die Umwelt ist $U_2 = \mathbb{N} \times \mathbb{N}_{mod\ 3}$, wobei $\mathbb{N}_{mod\ 3}$ die Restklassen von $\mathbb{N}$ modulo 3 sind, d.h.

$$\begin{array}{l} \mathbb{N}_{mod\ 3} = (X, Y, Z, d, l) \text{ mit } X = Y = \{0, 1\};\ Z = \{0, 1, 2\} \\ \quad l(0) = 0;\ l(1) = l(2) = 1 \\ d(n, 0) = n;\ d(0, 1) = 1;\ d(1, 1) = 2;\ d(2, 1) = 0 \end{array}$$

Beide Maschinen berechnen die Funktion $f(n) = n\ mod\ 3$. Das Ergebnis erscheint bei M_1 als Endmarke von $\mathfrak{A}_1$ und bei M_2 als Punkt in $\mathbb{N}_{mod\ 3}$, d.h. im zweiten Speicherplatz von U_2.

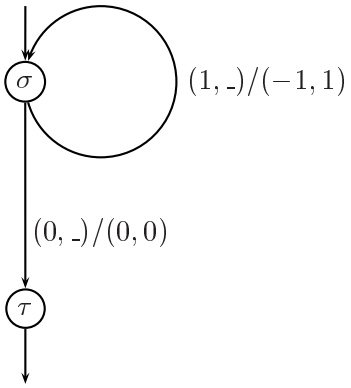


Abbildung 2.14: Dieser Automat berechnet ebenfalls $n \; mod \; 3$

Die Konfigurationenmengen beider Maschinen lassen sich einander bijektiv zuordnen, d.h. es gibt eine bijektive Funktion

$$\varphi : Z_1 \times S_1 \to Z_2 \times S_2,$$

die wie folgt definiert wird:

$$\begin{aligned} \varphi(n, s_i) &= (n, i, \sigma); \\ \varphi(n, t_i) &= (n, i, \tau); \quad (i = 0, 1, 2) \end{aligned}$$

Und nun stellt sich heraus, dass h_1 die Konfigurationen (n, s_i) analog verarbeitet wie h_2 die Konfigurationen $\varphi(n, s_i,)$. Genauer, es ist, wie man leicht nachrechnet,

$$\varphi(h_1(n, s_i)) = h_2(\varphi(n, s_i)). \tag{2.3}$$

Das ist die entscheidende Gleichung für die Homomorphie zwischen Algebren, wie sie im Kapitel über die mathematischen Grundlagen dargestellt wird.

Da φ bijektiv ist, kann man sogar vermuten, dass es sich um einen Isomorphismus handelt. Nach dem Satz 4.3.2 auf Seite 156 ist, um das zu bestätigen, nur noch nachzuprüfen, ob $h_1(n, s)$ definiert ist genau dann, wenn das auch für $h_2(n, i, \sigma)$ bzw. $h_2(n, i, \tau)$ gilt. Das ist aber richtig, wie man aus einem Vergleich der Zustandsgraphen beider Automaten erkennt. Auf den Wert von n kommt es bei beiden Automaten gar nicht an. Sie arbeiten dann und nur dann, wenn sie sich in ihren aktiven Zuständen befinden. Und die werden durch φ aufeinander abgebildet.

Das Beispiel hat allgemeingültige Bedeutung. Wir wollen es noch einmal analysieren und die Quintessenz als Satz formulieren.

VOM BEISPIEL ZUM SATZ

Einerseits haben wir einen Automaten mit nur zwei Zuständen σ und τ und die Umwelt $\mathbb{N} \times \mathbb{N}_{mod\ 3}$, wobei $\mathbb{N}_{mod\ 3}$ lediglich drei Punkte hat. Das Ergebnis findet sich am Schluss der Rechnung in dieser Umwelt.

Die dazu gleichwertige Maschine benutzt einen Prozessor mit zwei-mal-drei, also sechs Zuständen, dafür aber eine etwas ärmere Umwelt. Das Ergebnis der Rechnung wird daran erkannt, in welchem der drei terminalen Zustände der Automat endet. Der durch die endliche Umwelt $\mathbb{N}_{mod\ 3}$ repräsentierte Teil des Gedächtnisses der einen Maschine manifestiert sich in der vermehrten Anzahl von Zuständen der anderen Maschine.

Satz 2.3.2 *Sei $U = (X, Y, Z, d, l)$ eine R-Umwelt, $(\tilde{U}, \tilde{O}) = (\tilde{X}, \tilde{Y}, \tilde{Z}, \tilde{d}, \tilde{l})$ eine endliche punktierte $\tilde{R}$-Umwelt und $\mathfrak{A} = (S, s_0, X \times \tilde{X}, Y \times \tilde{Y}, M, \delta, \lambda, \mu)$ ein $R \times \tilde{R}$-Automat. Dann gibt es einen R-Automaten $\mathfrak{A}/\tilde{U} = (\Sigma, \sigma_0, X, Y, M, \delta^*, \lambda^*, \mu^*)$ so, dass die Maschinen $(\mathfrak{A}, U \times \tilde{U})$ und $(\mathfrak{A}/\tilde{U}, U)$ isomorph sind.*

Der Automat $\mathfrak{A}/\tilde{U}$ soll "$\mathfrak{A}$ **relativ zu** $\tilde{U}$" heißen.

Beweis: Die folgenden Überlegungen wirken wegen der Formeln kompliziert, sind aber in ihrer Struktur sehr einfach und nahe liegend. Die Figur 2.15 zeigt, wie der Prozessor $\mathfrak{A}/\tilde{U}$ konstruiert wird. Der Prozessor $\mathfrak{A}$ empfängt aus beiden Umwelten Signale in Gestalt von Paaren $(y, \tilde{y}) \in Y \times \tilde{Y}$ und gibt simultan Aktionen $x \in X$ und $\tilde{x} \in \tilde{X}$ an beide Umwelten zurück. Der Automat $\mathfrak{A}/\tilde{U}$ dagegen "kommuniziert" nur mit U, hat allerdings ein größeres Gedächtnis bekommen, weil er die Punkte von $\tilde{U}$ als Bestandteil seiner Zustandsmenge übernommen hat.

Zur Vereinfachung der Formeln wird konsequent die Schreibweise Px bzw. $\tilde{P}\tilde{x}$ anstelle von $d(P, x)$ bzw. $\tilde{d}(\tilde{P}, \tilde{x})$ verwendet.

Wie im Beispiel ist $\Sigma = S \times \tilde{Z}$ und $\sigma_0 = (s_0, \tilde{O})$. Mit $(P, \tilde{P}) \in Z \times \tilde{Z}$ ist $y = l(P)$ und $\tilde{y} = \tilde{l}(\tilde{P})$.

Um δ^* und λ^* zu definieren, werden u.a. δ und λ benutzt. Dabei muss man sich vergegenwärtigen, dass λ Paare $(x, \tilde{x}) \in X \times \tilde{X}$ erzeugt:

$$\lambda(s, (y, \tilde{y})) = (x, \tilde{x}) =: (\lambda_1(s, (y, \tilde{y}), \lambda_2(s, (y, \tilde{y}))$$

Mit den so eingeführten Funktionen λ_1 und λ_2 wird definiert:

$$\delta^*(\sigma, y) = \delta^*((s, \tilde{P}), y) := (\delta(s, (y, \tilde{y})), \tilde{P}\tilde{x})$$
$$\lambda^*(\sigma, y) = \lambda^*((s, \tilde{P}), y) := x$$

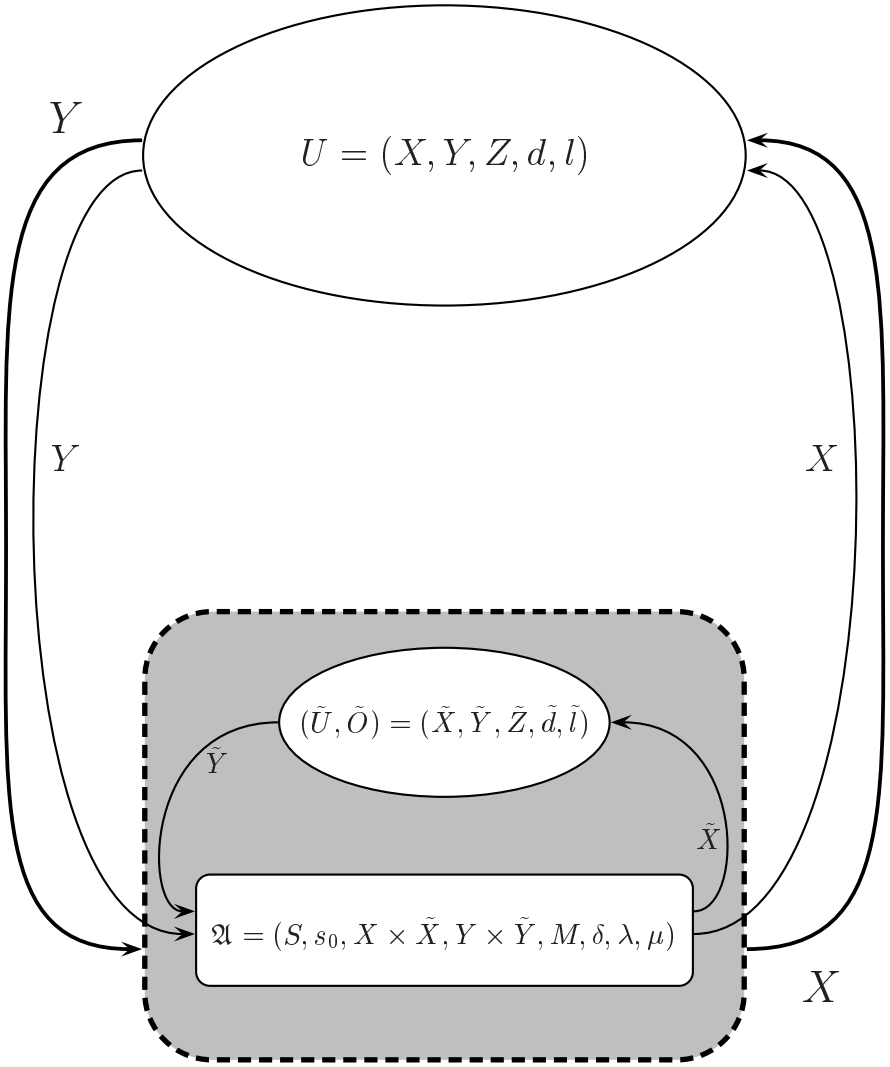
Es ist zu fragen, wie die Definitionsbereiche dieser beiden Funktionen aussehen. λ^* ist offenbar genau dann definiert, wenn $\sigma = (s, \tilde{P}) \in act\ \mathfrak{A} \times \tilde{Z}$ ist. Auf die Einblicke $y \in Y$ kommt es nicht an.

Es ist aber auch

$$dom\ \delta^* = act\ \mathfrak{A} \times \tilde{Z} \times Y$$

In der Tat, wenn $\delta^*((s, \tilde{P}), y)$ existiert, so auch $\delta(s, (y, \tilde{y}))$, also ist

$$(s, \tilde{P}) \in act\ \mathfrak{A} \times \tilde{Z}.$$



$$\mathfrak{A}/\tilde{U} = (S \times \tilde{Z}, (s_0, \tilde{O}), X, Y, M, \delta^*, \lambda^*, \mu)$$

Abbildung 2.15: Ergänzung eines Automaten durch eine endliche Umwelt

Ist umgekehrt $\sigma = (s, \tilde{P}) \in act\ \mathfrak{A} \times \tilde{Z}$, so existieren zunächst $\delta(s, (y, \tilde{y}))$ und $\lambda(s, (y, \tilde{y}))$. Da $\mathfrak{A}$ ein $R \times \tilde{R}$-Automat ist, gilt

$$\lambda(s, (y, \tilde{y})) = (\lambda_1(s, (y, \tilde{y})), \lambda_2(s, (y, \tilde{y})) \in R(y)) \times \tilde{R}(\tilde{y}) \tag{2.4}$$

Damit ist auch $\tilde{P}\tilde{x} = \tilde{P}\lambda_2(s, \tilde{l}(\tilde{P}))$ definiert und also

$$act\ \mathfrak{A} \times \tilde{Z} \times Y \subseteq dom\delta^*.$$

Als Zwischenergebnis haben wir

$$act\ \mathfrak{A}/\tilde{U} = act\ \mathfrak{A} \times \tilde{Z}$$

bekommen.

$\mathfrak{A}/\tilde{U}$ ist ein R-Automat, denn aus 2.4 folgt

$$\lambda^*((s,\tilde{P}),y) = \lambda_1(s,(y,\tilde{l}(\tilde{P}))) \in R(y).$$

Bleibt die Isomorphie beider Maschinen zu zeigen. Dazu schreiben wir zunächst einmal ganz formal auf, wie die Funktionen

$$\begin{aligned} h &: (Z \times \tilde{Z}) \times S \stackrel{\supset}{\longrightarrow} (Z \times \tilde{Z}) \times S \text{ und} \\ h^* &: Z \times \Sigma \stackrel{\supset}{\longrightarrow} Z \times \Sigma \end{aligned}$$

arbeiten. Wie oben ist wieder

$$y = l(P);\ \tilde{y} = \tilde{l}(\tilde{P});\ (x,\tilde{x}) = \lambda(s,(y,\tilde{y}))$$

Damit bekommt man

$$h((P,\tilde{P}),s) = ((P,\tilde{P})\lambda(s,(y,\tilde{y})),\delta(s,(y,\tilde{y})) = ((Px,\tilde{P}\tilde{x}),\delta(s,(y,\tilde{y}))$$

$$\begin{aligned} h^*(P,\sigma) &= h^*(P,(s,\tilde{P})) = (P\lambda^*((s,\tilde{P}),y),\delta^*((s,\tilde{P}),y)) \\ &= (Px,(\delta(s,(y,\tilde{y})),\tilde{P}\tilde{x})) \end{aligned}$$

Die beiden Gleichungen offenbaren unmittelbar, wie die Bijektion φ auszusehen hat:

$$\begin{aligned} \varphi : Z \times \tilde{Z} \times S &\to Z \times \Sigma = Z \times S \times \tilde{Z} \\ (P,\tilde{P},s) &\mapsto \varphi(P,\tilde{P},s) := (P,s,\tilde{P}) \end{aligned}$$

Damit ist jedenfalls

$$\varphi(h((P,\tilde{P}),s)) = h^*(\varphi(P,\tilde{P},s)) \text{ bzw.}$$
$$\varphi \circ h = h^* \circ \varphi$$

Nach dem Satz 4.3.2 auf Seite 156 bleibt zu zeigen, dass $h(P,\tilde{P},s)$ genau dann definiert ist, wenn das auch für $h^*(\varphi(P,\tilde{P},s)) = h^*(P,s,\tilde{P})$ der Fall ist. Dazu kann man die Folgerung 2.3.2 benutzen. $(U \times \tilde{U},\mathfrak{A})$ ist eine $R \times \tilde{R}$-Maschine. Also ist

$$dom\ h = Z \times \tilde{Z} \times act\ \mathfrak{A}.$$

Mit dem analogen Argument ist

$$dom\ h^* = Z \times act\ \mathfrak{A}/\tilde{U} = Z \times act\ \mathfrak{A} \times \tilde{Z}.$$

Und natürlich ist $(P, \tilde{P}, s)$ in der einen Menge dann und nur dann, wenn $(P, s, \tilde{P})$ in der anderen ist.

Damit ist der Beweis abgeschlossen.

Dieser Satz ist relevant im Zusammenhang mit der sprachlichen Definition von Automaten im Abschnitt 2.4. Dort werden Variable benutzt, um in flexibler Weise Algorithmen zu formulieren. Mit diesem Satz im Hintergrund ist ein Programm, in dem Variable benutzt werden, immer dann ein endlicher Automat, wenn diese Variablen nur mit endlich vielen Werten belegt werden dürfen.

2.3.3 R-Schemata

Wir greifen in diesem Abschnitt die in 2.1.3 beschriebene Verknüpfung von Automaten wieder auf, wobei aber der Zusammenhang zu einer Umwelt berücksichtigt wird. Ziel ist es, R-Automaten mit sprachlichen Mitteln zu beschreiben. Um die Gleichwertigkeit der Darstellung von Automaten mit Zustandsgraphen einerseits und mit den Mitteln einer formalen Sprach andererseits nachzuweisen, benötigen wir den Begriff des R-Schemas oder Flussdiagramms(englisch **flowchart**).

EIN OFT BENUTZTES GRAPHISCHES MODELL

Definition 2.3.3 *Das Quadrupel $G = (V, E, o, t)$ heißt **gerichteter Graph**, wenn für dessen Elemente die folgenden Voraussetzungen erfüllt sind:*

- *V und E sind endliche Mengen; die Elemente von V heißen **Knoten** (engl. vertices), jene von E nennen wir **Kanten** (engl. edges).*
- *o und t sind Funktionen von E in V:*

$$o, t : E \to V$$

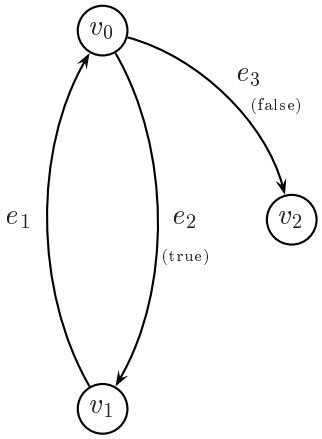
Für $e \in E$ heißt $o(e)$ der Anfangs- und $t(e)$ der Endknoten der Kante e (o und t sind Abkürzungen der englischen Worte origin und terminal).

Gerichtete Graphen lassen sich in nahe liegender Weise durch Figuren veranschaulichen. Abbildung 2.16 zeigt einen solchen. Auch die Zustandsgraphen unserer Automaten sind gerichtete Graphen.

Zur besseren sprachlichen Verständigung wollen wir noch einige gängige Begriffe definieren.

Die **Valenz** eines Knotens v ist die Menge aller Kanten, die von diesem Knoten ausgehen:

$$val\ v := \{e \in E \mid o(e) = v\}$$



$V = \{v_0, v_1, v_2\}$

$E = \{e_1, e_2, e_3\}$

	o	t
e_1	v_1	v_0
e_2	v_0	v_1
e_3	v_0	v_2

Abbildung 2.16: Beispiel eines gerichteten Graphen

In der Abbildung 2.16 ist z.B. *val* $v_0 = \{e_2, e_3\}$. Ein Knoten soll in unserem Kontext **terminal** heißen, wenn von ihm keine Kante ausgeht, also *val* $v = \emptyset$ ist. Im andere Falle, wenn *val* $v \neq \emptyset$ ist, wird v **aktiver Knoten** genannt. Diese beiden Knotenarten werden in geeigneter Weise zusammengefasst:

$$act\ G := \{v \in V \mid val\ v \neq \emptyset\} \qquad term\ G := \{v \in V \mid val\ v = \emptyset\}$$

Bei der Verwendung gerichteter Graphen im Zusammenhang mit flowcharts ist einer der Knoten immer hervorgehoben, er wird **initialer** oder **Anfangsknoten** heißen.

Nun stelle man sich vor, dass in der Abbildung 2.16 der Knoten v_0 mit einem booleschen Prozessor $\mathfrak{B}$ bestückt ist, dessen terminaler Knoten mit der Marke *true* der Kante e_2 zugeordnet ist, während zur Endmarke *false* die Kante e_3 gehört. Der Knoten v_1 soll mit einem Aktor $\mathfrak{A}$ besetzt sein, während im terminalen Knoten v_2 die Marke *fertig* abgelegt ist. Offenbar ist auf diese Weise gerade jener zusammengesetzte Prozessor entstanden, den wir im Abschnitt 2.1.3 auf Seite 22 mit der Bezeichnung "*while* $\mathfrak{B}$ *do* $\mathfrak{A}$ *end while*" beschrieben haben.

Nach diesem Prinzip werden Flussdiagramme allgemein definiert.

PROGRAMM-ABLAUFPLAN ODER *R*-SCHEMA

Definition 2.3.4 *Das Tripel* (G, A, M) *heißt R-Schema, wenn die folgenden Bedingungen erfüllt sind:*

- $G = (E, V, v_0, o, t)$ *ist ein gerichteter Graph mit dem initialen Knoten* v_0
- M *ist eine endliche Menge von Marken*

- *A ist eine Funktion, deren Definitionsbereich die Knotenmenge V von G ist. Diese Funktion wirkt in zweierlei Weise:*
  - *Wenn v aktiver Knoten ist, dann soll $A(v) =: \mathfrak{A}_v$ ein R-Prozessor sein, dessen Markenmenge M_v sich eindeutig auf val v abbilden lässt. Das heißt, es gibt zu jedem aktiven Knoten v eine Funktion*

$$\varphi_v : M_v \to val\ v.$$

  - *Falls $v \in term\ G$, dann ist $A(v)$ ein Element aus M.*

Wir wollen hervorheben, dass die in den aktiven Knoten des Graphen platzierten Prozessoren alle auf die gleiche Relation $R \subseteq Y \times X$ bezogen sind. Sie können also allesamt in ein und derselben Umwelt arbeiten.

Unser Ziel ist es, ein R-Schema als komplexen R-Automaten zu interpretieren. Die terminalen Knoten des Graphen werden die terminalen Zustände dieses Gesamtprozessors sein und M seine Markenmenge. Ist v ein aktiver Knoten, M_v die Markenmenge des Prozessors $\mathfrak{A}_v$ und $e \in val\ v$ ein Kante, die einer Marke $m \in M_v$ entspricht, so soll m Marke oder Markierung der Kante e heißen. Bei einer bildlichen Darstellung werden wir häufig e durch m ersetzen, wie das in Abbildung 2.16 angedeutet ist. Allerdings erlaubt die Definition eines R-Schemas auch, dass zu einer Graphenkante mehrere Marken gehören.

Satz 2.3.3 *Zu jedem R-Schema (G, A, M) lässt sich in kanonischer Weise ein R-Prozessor $\Gamma(G, A, M)$ konstruieren.*

EIN AUTOMATENKOMPLEX WIRD ZU EINEM KOMPLEXEN AUTOMATEN

Der Beweis wird durch die Konstruktion des Prozessors

$$\Gamma(G, A, M) = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

erbracht. Nachfolgend wird anstelle von $\Gamma(G, A, M)$ einfach Γ geschrieben. Die zu den aktiven Knoten $v \in V$ des Graphen gehörigen Automaten seien

$$\mathfrak{A}_v = (S_v, s_{0v}, X, Y, M_v, \delta_v, \lambda_v, \mu_v).$$

Die Zustandsmenge von Γ besteht aus der freien Vereinigung der aktiven Zustände aller Automaten, welche in den aktiven Knoten platziert sind, und aus den terminalen Knoten von G:

$$S := \left(\bigsqcup_{v \in act\ G} act\ \mathfrak{A}_v \right) \cup term\ G = \bigcup_{v \in act\ G} (act \mathfrak{A}_v \times \{v\}) \cup term\ G$$

Die terminalen Zustände der einzelnen Prozessoren sind also für den "Gesamtautomaten" nicht mehr von Interesse.

Der Anfangszustand von Γ ist – abgesehen von der durch die freie Vereinigung verursachten Markierung – der Anfangszustand des Prozessors, der dem Anfangsknoten v_0 des Graphen zugeordnet ist:

$$s_0 := (s_{0v_0}, v_0).$$

Die terminalen Zustände von Γ sind die terminalen Knoten des Graphen:

$$\mu : S \to M \cup \{go\} \text{ mit } \mu(s) = \begin{cases} A(v) \in M, & \text{falls } s = v \in term\ G \\ go & \text{sonst} \end{cases}$$

Überführungs- und Ausgabefunktion von Γ werden auf die entsprechenden Funktionen der Teilautomaten zurückgeführt. Für die Ausgabefunktion gilt einfach

$$\lambda((s,v),y) := \lambda_v(s,y).$$

Die Definition der Zustandsüberführungsfunktion von Γ erfordert mehr Überlegung. Sie stimmt überein mit derjenigen von $\mathfrak{A}_v$, solange dieser Teilautomat sich in der Menge seiner aktiven Zustände bewegt. Interessant wird es, wenn $\mathfrak{A}_v$ einen terminalen Zustand s erreicht. Dem ist ja in wohlbestimmter Weise eine Kante $\varphi_v(\mu_v(s))$ des Graphen zu einem neuen Knoten $v' = t(\varphi_v(\mu_v(s)))$ zugewiesen. Jetzt kommt es darauf an, ob dieser neue Knoten aktiv oder terminal ist. Im ersten Falle soll Γ in den Anfangszustand $s_{ov'}$ des Teilautomaten $\mathfrak{A}_{v'}$ übergehen, im anderen Falle in den Endzustand v'. Das wird mit den folgenden Formeln ausgedrückt:

$$\delta_v(s,v) \in act\ \mathfrak{A}_v \qquad \Rightarrow \qquad \delta((s,v),y) := (\delta_v(s,y),v)$$

$$\begin{array}{l} s' = \delta_v(s,y) \in term\ \mathfrak{A}_v \text{ und} \\ v' = t(\varphi_v(\mu_v(s'))) \in act\ G \end{array} \qquad \Rightarrow \qquad \delta((s,v),y) := (s_{0v'}, v')$$

$$\begin{array}{l} s' = \delta_v(s,y) \in term\ \mathfrak{A}_v \text{ und} \\ v' = t(\varphi_v(\mu_v(s'))) \in term\ G \end{array} \qquad \Rightarrow \qquad \delta((s,v),y) := v'$$

Damit ist Γ vollständig definiert. Dass es sich um einen R-Automaten handelt, ergibt sich unmittelbar aus der Definition von λ.

FALLUNTERSCHEIDUNG

Auf der Grundlage dieses Satzes fügen wir den Automatenverknüpfungen aus 2.1.3 ein weiteres Konstrukt hinzu, das in der Gestalt

$$\begin{array}{l} case\ of\ \mathfrak{B} \\ \quad M_1\ then\ \mathfrak{A}_1| \\ \quad M_2\ then\ \mathfrak{A}_2| \\ \qquad \cdots \\ \quad M_n\ then\ \mathfrak{A}_n \\ \qquad else\ \ \mathfrak{A}_{n+1} \end{array}$$

geschrieben wird.

Darin sind die $\mathfrak{A}_i$; $(1 \le i \le n+1)$ Aktoren und $\tilde{M} = \bigcup_{i=1}^{n} M_i$ ist die Markenmenge von $\mathfrak{B}$.

Die senkrechten Striche hinter den $\mathfrak{A}_i$ sollen ausdrücken, dass jede Merke nur einmal auszuwerten ist, und zwar dann, wenn sie zum ersten Mal in einer Menge

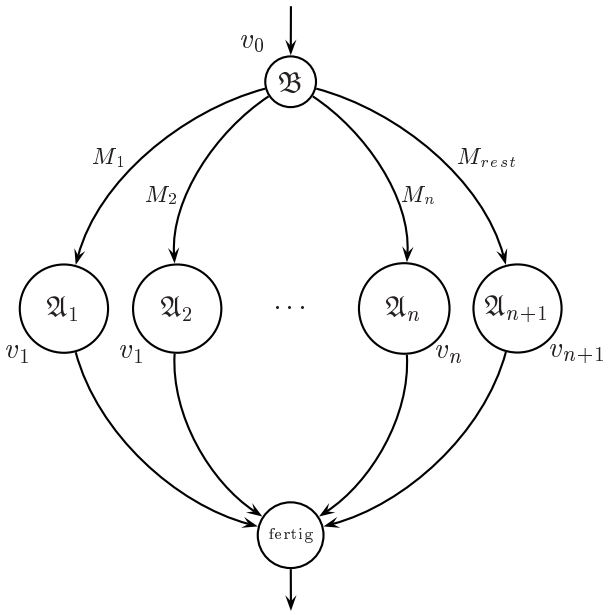


Abbildung 2.17: Das R-Schema des *case*-Automaten

M_i enthalten ist. Das heißt z.B., wenn $n \in M_1 \cup M_2$, so soll nur der Aktor $\mathfrak{A}_1$ aktiv werden.

Die Abbildung 2.17 zeigt das Flussdiagramm. Die Namen der Knoten sind neben die Kreise geschrieben, welche die Knoten darstellen. Eine Menge M_i umfasst alle jene Endmarken von $\mathfrak{B}$, welche vermittels der Funktion φ_{v_0} der Kante von v_0 nach v_i zugewiesen werden, und $M_{rest} = \tilde{M} \setminus \bigcup_{i=1}^{n} M_i$

2.4 Eine formale Sprache für Automaten

2.4.1 Ein erstes Beispiel

Wir haben jetzt alle Vorbereitungen getroffen, um Automaten darstellen zu können mit sprachlichen Mitteln, die gängigen Programmiersprachen entlehnt sind. Außerdem sind wir in der Lage, die Gleichwertigkeit dieser Darstellung mit den bisher benutzten Methoden sauber zu begründen.

Als Einstiegsbeispiel wollen wir einen Aktor beschreiben, der mit den Mitteln der Zählerumwelt die Funktion

$$\begin{aligned} &div\_3 : \mathbb{N} \to \mathbb{N} \\ &\qquad n \mapsto div\_3(n) := max\{k \in \mathbb{N} \mid k \cdot 3 \leq n\} \end{aligned}$$

Zuerst ein Beispiel

berechnet. Der Berechnungsprozeß soll in $\mathbb{N} \times \mathbb{N}$ ablaufen, wobei angenommen wird, dass die erste Koordinate eines Punktes $P = (n, m) \in \mathbb{N} \times \mathbb{N}$ die Zahl ist, deren Funktionswert zu bestimmen ist. Zunächst wird dafür gesorgt, dass die zweite Koordinate null wird. Anschließend wird die erste Koordinate solange herunter gezählt, bis null erreicht ist. Gleichzeitig wird nach je drei Zählschritten die zweite Koordinate um eins erhöht. Der Funktionswert kann nach Abschluss der Prozedur in der zweiten Koordinate abgelesen werden. Diesen Vorgang beschreiben wir in der folgenden Art:

```
actor div_3
begin
  while look_of (_,1) do action (0,-1) end while;
  while look_of (1,_) do
   action (-1,0);
   if look_of (1,0) then
     action (-1,0);
     if look_of (1,0) then
       action (-1,1)
     end if
   end if
  end while
end
```

Wie ist das gemeint?

Wie soll dieses Programm verstanden werden? Um Übereinstimmung zu bekommen mit dem Konzept der Verknüpfung von Automaten mit Hilfe von R-Schemata muss das Symbol "**action** (x_1, x_2)" einen Aktor und "**look_of** (y_1, y_2)" einen booleschen Prozessor repräsentieren. Wie sollen diese Automaten aussehen?

Beginnen wir mit dem action-Konstrukt. Ein Symbol "**action** (x)" bedeutet, dass in der Umwelt U die Aktion x ausgeführt wird, also ein Punkt P in $Q = d(P, x)$ übergeht, und zwar unabhängig vom Einblick $l(P)$(vergl. Figur 2.18)

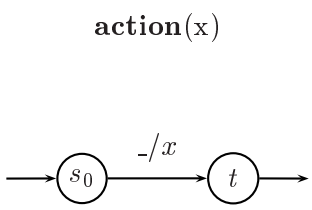


Abbildung 2.18: Der Aktor **action**(x)

Für das look-Konstrukt wollen wir einen allgemeineren Zusammenhang herstellen. Wenn $\mathfrak{A}$ ein Prozessor über einer Umwelt U mit der Markenmenge M kein Aktor ist und $M_1 \in M$ gilt, so kann man dazu einen booleschen Automaten "$\mathfrak{A}$**_of** M_1" wie folgt definieren: der Prozessor liefert bei Start im Punkte P die Endmarke $true$ genau dann, wenn $l(P) \in M_1$ ist. In allen anderen Fällen gibt er $false$ aus.

Definition 2.4.1 *Zu einem Automaten*

$$\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

mit $\#M > 1$ [4] *ist*

$$\mathfrak{A}\textbf{\_of}\, M_1 = (S', s_0, X, Y, M', \delta', \lambda', \mu')$$

definiert durch

$$S' = (act\ \mathfrak{A} \setminus term\ \mathfrak{A}) \cup \{t_1, t_2\}; \quad act(\mathfrak{A}\textbf{\_of}\, M_1) = act\ \mathfrak{A}$$

$$M' = \{true, false\}$$

$$\delta'(s,y) = \begin{cases} t_1, & wenn\ \delta(s,y) = t \in term\ \mathfrak{A}\ und\ \mu(t) \in M_1 \\ t_2, & wenn\ \delta(s,y) = t \in term\ \mathfrak{A}\ und\ \mu(t) \notin M_1 \\ \delta(s,y) & sonst \end{cases}$$

$$\mu'(s) = \begin{cases} true & für\ s = t_1 \\ false & für\ s = t_2 \\ go & sonst \end{cases}$$

[4] $\#M$ bezeichnet die Kardinalzahl einer Menge, wenn es sich wie hier um endliche Mengen handelt, die Anzahl ihrer Elemente.

Wenn im Sonderfall $M_1 = \{m\}$ nur aus einem Element besteht, wird vereinfachend die Schreibweise $\mathfrak{A}$ **_of** m benutzt.

In der Definition 2.1.2 auf Seite 23 wird zu einem booleschen Prozessor der negierte Automat beschrieben. Das kann man hier anwenden und hat mit *non* $\mathfrak{A}$*_of* M_1 einen Selektor, der alle jene Endmarken von $\mathfrak{A}$ erfasst, die nicht in M_1 liegen.

Diese für beliebige Automaten mögliche Konstruktion wird nun auf einen besonderen, **look** genannten Sensorautomaten angewendet, dessen Zustandsgraph in Abbildung 2.19 zu sehen ist. Die Markenmenge dieses Prozessors ist die Menge

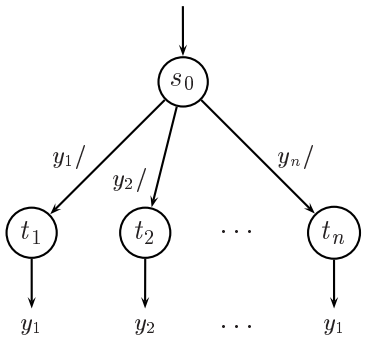


Abbildung 2.19: **look** berechnet die Einblicksfunktion

LOOK=SICH UMSCHAUEN

Y der Einblicke in der Umwelt des Automaten.

Nun ist klar, was "**look_of** (_,1)" tut. Es handelt sich um einen booleschen Automaten über der Umwelt $\mathbb{N} \times \mathbb{N}$, der solange "*true*" ausgibt, wie er in der zweiten Koordinate eines Umweltpunktes eine positive Zahl sieht, im anderen Falle erzeugt er die Marke "*false*".

Damit ist die Semantik des Programms vollständig definiert. Es beschreibt über der Umwelt $\mathbb{N} \times \mathbb{N}$ ein R-Schema, das in Abbildung 2.20 dargestellt ist.

Wenn man nach Satz 2.3.3 auf Seite 67 den zu diesem R-Schema gehörenden Prozessor konstruiert, erhält man den Zustandsgraphen der Abbildung 2.21.

An diesem Beispiel ist einmal ausführlich beschrieben worden, wie die Beziehung zwischen einem gegebenen Programm für einen Automaten und seinem Zustandsgraphen herzustellen ist. Bei zukünftigen Beispielen könnte man in jedem Falle eine entsprechende Betrachtung anstellen.

Von prinzipieller Bedeutung ist in diesem Zusammenhang, ob man die Verfahrensweise auch umkehren kann, d.h. ob es zu jedem Zustandgraphen ein Programm gibt, das einen gleichwertigen Automaten repräsentiert. Solche Graphen sind ja unter Umständen sehr komplex und unüberschaubar. Bevor wir uns die-

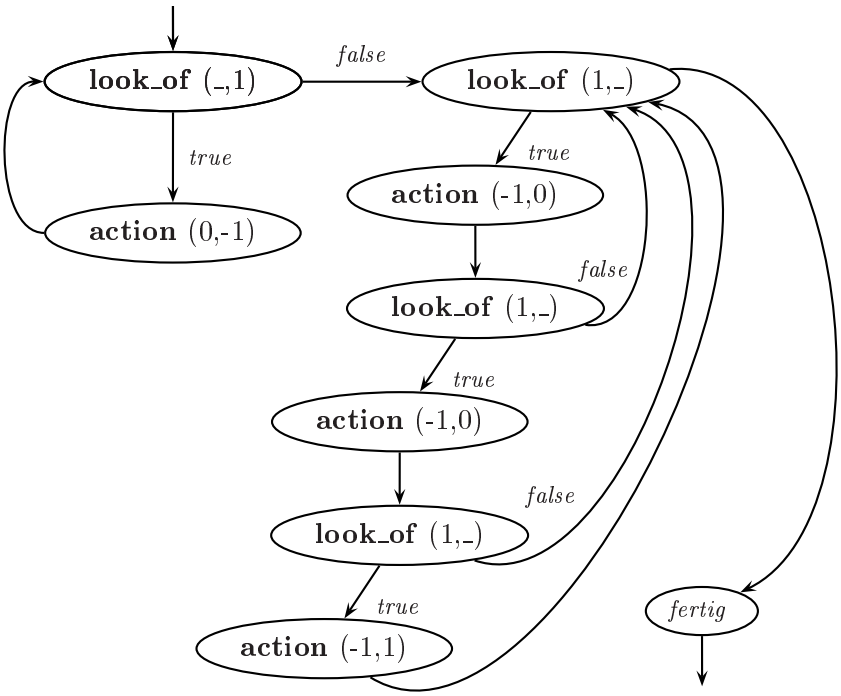


Abbildung 2.20: R-Schema zum Beispiel

sem Problem zuwenden, soll mit weiteren Beispielen die Flexibilität der sprachlichen Darstellung von Automaten demonstriert werden.

2.4.2 Programme mit Variablen

Das Beispiel des vorigen Abschnittes kann man natürlich weiterführen. Statt durch drei kann man durch jede andere positive natürliche Zahl m teilen. Man muss nur die *if-then-else-end if*-Verzweigungen in der zweiten *while*-Schleife entsprechend vervielfachen. Das ist allerdings unzumutbar umständlich. Wir werden statt dessen mit einer zusätzlichen endlichen Umwelt $[m]$ arbeiten. Der Automat *div_m* wird also insgesamt über der Umwelt $\mathbb{N}\times\mathbb{N}\times[m]$ arbeiten. Wir wissen aus dem Abschnitt über Maschinen, dass ein solcher Prozessor immer durch einen gleichwertigen Automaten über $\mathbb{N}\times\mathbb{N}$ ersetzt werden kann. Die endliche Umwelt

$$[m] = (X, Y, Z, d, l)$$

ist in unserem Beispiel folgendermaßen definiert:

- $X = Y = \{0, 1\}$
- $Z = [m] := \{0, 1, 2, \ldots, m-1\}$

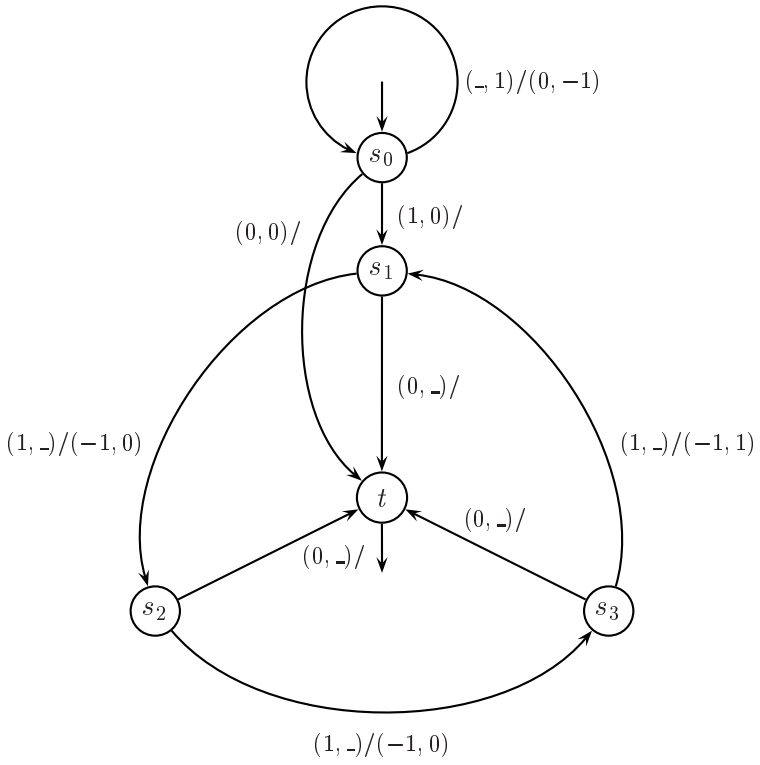


Abbildung 2.21: Der Zustandsgraph des Aktors *div_3*

- 0 ist die Stoppaktion und $d(a, 1) = a + 1 \; mod \; m$, also

$$d(a, 1) = \begin{cases} a + 1, & \text{wenn } a < m - 1 \\ 0 & \text{für } a = m - 1 \end{cases}$$

-

$$l(a) = \begin{cases} 0, & \text{falls } a = 0 \\ 1 & \text{sonst} \end{cases}$$

Die Zahlen dieser Umwelt werden benutzt, um die Arbeit des Automaten in Gruppen von m Zählschritten zu unterteilen. Und nun das Programm:

```
        actor div_m
        declare x : [m] end declare
        begin
(1)       x := 0;
(2)       while look_of (_,1) do action (0,-1) end while;
(3)       while look_of (1,_) do
(4)         action (-1,0);
```

(5) $x := x + 1$;
(6) **if** x=0 **then action** (0,1) **end if**
end while
end

Das Programm zeigt, dass die Punkte der endlichen Umwelt $[m]$ durch die Variable x repräsentiert sind. Die Aktionen in dieser Umwelt werden anders geschrieben als diejenigen, welche in der Umwelt $\mathbb{N} \times \mathbb{N}$ erfolgen: statt
action (1) steht $x := x + 1$. Damit wird deutlich gemacht, dass die Variablenwerte als Zustände des Automaten zu interpretieren sind. Und davon darf es vereinbarungsgemäß nur endlich viele geben. In der Deklarationszeile muss also jeder Variablen zwingend eine endliche Menge zugewiesen werden.

Die Anweisung $x := 0$ ist keine Aktion, sondern ein kleiner Aktor, der z.B. so geschrieben sein könnte:

repeat $x := x + 1$ **until** $x = 0$

Was bedeutet $x = 0$? Das ist der boolesche Automat, den wir sonst in der Gestalt **look_of** 0 schreiben. Wir haben ihn im vorigen Abschnitt erläutert.

Damit ist die Semantik des Umgangs mit Variablen bei diesem Programm klar. In der Tabelle 2.6 ist der Weg des Automaten erfasst, den er zurücklegt, wenn er im Punkte $(7, 4, 1) \in \mathbb{N} \times \mathbb{N} \times [3]$ gestartet wird. Die Zahlen in der vierten Spalte bezeichnen die Nummern der Programmzeilen, in welchen die entsprechenden Veränderungen ausgelöst werden.

Hier stellt sich wieder einmal die Frage, ob das Programm, das bei einem Test erwartungsgemäß funktioniert, in jedem Falle, also bei Start in einem beliebigen Punkte der Umwelt, die richtige Lösung liefert. Die Antwort wird als Lösung einer Aufgabe erwartet.

$\mathbb{N}$	$\mathbb{N}$	[3]	Zeile Nr.
7	3	1	begin
7	3	0	(1)
7	0	0	(2)
6	0	1	(4),(5)
5	0	2	(4),(5)
4	0	0	(4),(5)
4	1	0	(6)
3	1	1	(4),(5)
2	1	2	(4),(5)
1	1	0	(4),(5)
1	2	0	(6)
0	2	1	(4),(5)

Tabelle 2.6: Der Weg von *div_3*
bei Start in (7,3,1)

VARIABLEN DÜRFEN NUR MIT ENDLICH VIELEN WERTEN BELEGT WERDEN

Im allgemeinen Fall hat der Deklarationsteil eines Programms die folgende Gestalt:

$$\textbf{declare } a_1 : A_1; a_2 : A_2; \ldots ; a_p : A_p \textbf{ end declare}$$

Dabei sind die A_i endliche Mengen, die eine endliche Umwelt $\tilde{U}$ mit

$$\tilde{Z} = A_1 \times A_2 \times \cdots \times A_p$$

darstellen. Wie Aktionen, Einblicke und die Funktionen d und l in $\tilde{U}$ definiert sind, hängt vom Problem ab. Häufig wird jede Menge A_i eine Umwelt mit

$$X = Y = Z = A_i = \{c_1, c_2, \ldots , c_s\},\ l(c_i) = c_i \text{ und } d(c_m, c_n) = c_n$$

sein. Und $\tilde{U}$ ist das in 2.2.2 definierte eingeschränkte kartesische Produkt der A_i.

Die Umwelt des Automaten ist $U \times \tilde{U}$. Im Programm wird mit dem Schlüsselwort **action** Bezug genommen auf Aktionen in U, wogegen Aktionen in $\tilde{U}$ sich immer auf eine bestimmte Variable beziehen , weshalb sie in dieser endlichen Umwelt in der Regel die Gestalt von Zuweisungen der Form $a_i := c_m$ haben.

EIN ALTES BEISPIEL IN NEUEM GEWAND

Bisher haben wir lediglich Aktoren programmiert. Wie sieht ein Programm aus für einen Automaten mit mehreren Endmarken?

Als Beispiel greifen wir erneut den Automaten aus 2.1.2 (Abbildung 2.1) auf, wollen das Programm aber so verallgemeinern, dass es für jede positive natürliche Zahl einen Automaten repräsentiert, welcher $f(n) = n \ mod \ m$ berechnet. Die endliche Umwelt $[m]$ ist dieselbe wie oben.

```
automaton mod_m returns_label [m]
description{[m] ist dieselbe endliche Umwelt
            wie im vorigen Beispiel.}
declare x : [m] end declare
begin
  x := 0;
  while look_of 1 do
    action (−1);
    x := x + 1
  end while;
  return_label x
end
```

Die Bezeichnung **return_label** x drückt aus, dass im R-Schema ein terminaler Knoten erreicht ist, dessen Endmarke x heißt.

Es ist empfehlenswert, zu diesem Programm R-Schema und Zustandsgraphen für einen fixierten Wert von m zu zeichnen. Für $m = 3$ wird man zu Figur 2.1 gelangen.

Wie kann man sich in diesem Falle von der Korrektheit des Programms überzeugen?

Man kann die Beweismethode der vollständigen Induktion benutzen. Wird der Automat in einem Punkte $(n, s) \in \mathbb{N} \times [m]$ gestartet, so geht er mit $(n, 0)$ in die **while**-Schleife.

Zu jedem n gibt es eine natürliche Zahl k derart, dass

$$k \cdot m \leq n \leq (k+1) \cdot m - 1$$

gilt. Die Induktion wird nach k geführt. Ist $k = 0$, also

$$0 \leq n \leq m - 1,$$

so wird die Schleife im Punkte $(0, n)$ verlassen. Dieser Wert von n ist der verlangte Funktionswert und wird als Marke ausgegeben. Damit ist der Induktionsanfang geleistet.

Ist

$$(k+1) \cdot m \leq n \leq (k+2) \cdot m - 1,$$

wird die Schleife in $(n, 0)$ begonnen. Nachdem sie m-Mal absolviert wurde, ist der Automat im Punkte $(n - m, 0)$ angelangt, und es ist

$$k \cdot m \leq n - m \leq (k+1) \cdot m - 1.$$

Da n und $n - m$ denselben Rest bei Division durch m liefern, arbeitet der Automat für $k + 1$ korrekt, sobald das auch für k gilt.

Damit ist der Induktionsbeweis erbracht.

Wie wollen noch einmal zurück kehren zum Abschnitt 2.1.3 über die Verknüpfung von Automaten. Dort hatten wir angemerkt, dass ein Automat der Gestalt

$$\textbf{if } \mathfrak{B} \textbf{ then } \mathfrak{A}_1 \textbf{ else } \mathfrak{A}_2$$

durchaus mehrere terminale Zustände haben kann, nämlich dann, wenn z.B. $\mathfrak{A}_1$ kein Aktor ist. Dann wird jede **return**-Klausel zu einem anderen terminalen Knoten des R-Schemas bzw. terminalen Zustand des Gesamtautomaten führen.

2.4.3 Ein weiteres Hilfsmittel: "look$\rightarrow y$"

Die Beispiele des vorigen Abschnittes haben deutlich gemacht, dass die Verwendung von Variablen die Flexibilität unserer Sprache zur Darstellung von Automaten wesentlich erhöht. Jetzt werden wir einen Elementarautomaten bereitstellen, der Variablen mit Einblicken in Beziehung setzt: einer Variablen wird als Wert der Einblick in einem Punkt einer Umwelt zugewiesen. Die Konstruktion wird allgemein definiert und dann auf den Einblicksautomat **look** übertragen.

Sei also $\mathfrak{A}$ ein beliebiger Automat mit der Markenmenge M. Das Folgende ist nicht sinnvoll, wenn M außer go nur ein Element enthält. Wir setzen $\#M > 2$ voraus. Die Markenmenge wird zu einer Umwelt

$$M = (\tilde{X}, \tilde{Y}, \tilde{Z}, \tilde{d}, \tilde{l})$$

durch

- $\tilde{X} = \tilde{Y} = \tilde{Z} = M$
- $\tilde{d}(m_1, m_2) = m_2$
- $\tilde{l}(m) = m$

INFORMALE BESCHREIBUNG

Zu dem Automaten $\mathfrak{A}$, der über einer Umwelt U arbeitet, wird ein Aktor

$$\mathfrak{A} \rightarrow m$$

definiert, der über $U \times M$ arbeiten wird. m bezeichnet eine Variable, die mit Werten aus M belegt werden kann: eine Variable vom Typ M.

Die Aufgabe von "$\mathfrak{A} \rightarrow m$" besteht darin, am Ende seiner Arbeit in der zweiten Koordinate des Umweltpunktes die Endmarke abzulegen, die von $\mathfrak{A}$ erzeugt wird.

Mit anderen Worten: ist $Q = P\mathfrak{A}$ und $\alpha = \mathfrak{A}(P)$, so soll

$$(P, x)(\mathfrak{A} \rightarrow m) = (Q, \alpha)$$

sein.

GENAUE BEGRIFFSBESTIMMUNG

Definition 2.4.2 *Zu*

$$\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$$

ist

$$\mathfrak{A} \to m = (\overline{S}, s_0, \overline{X}, \overline{Y}, \overline{M}, \overline{\delta}, \overline{\lambda}, \overline{\mu})$$

bestimmt durch

$$\begin{aligned}
\overline{S} &= act\,\mathfrak{A} \sqcup \{t\} \\
term\,(\mathfrak{A} \to m) &= \{t\} \\
\overline{X} = X \times \tilde{X};&\quad \overline{Y} = X \times \tilde{X};\quad \overline{M} = \{*, go\} \\
\overline{\delta}(s,(y,h)) &= \begin{cases} t, & wenn\ \delta(s,y) \in term\,\mathfrak{A} \\ \delta(s,y) & sonst \end{cases} \\
\overline{\lambda}(s,(y,h)) &= (\lambda(s,y), \mu(\delta(s,y))) \\
\overline{\mu}(s) &= \begin{cases} * & für\ s = t \\ go & sonst \end{cases}
\end{aligned}$$

Man erkennt, dass $\overline{\lambda}$ die erste Komponente der Punkte, also den aus der Umwelt U stammenden Teil in derselben Weise wie λ verändert. Die zweite Komponente erhält solange den Wert go, wie durch δ aktive Zustände erzeugt werden. Erst falls $\delta(s,y) \in term\,\mathfrak{A}$ ist, erhält man, wenn P der Startpunkt in U ist, mit $\mu(\delta(s,y)) = \mathfrak{A}(P) \in M \setminus \{go\}$ eine Aktion in der Umwelt M, die eine "echte" Endmarke ist.

Wie in 2.4.1 wird diese für beliebige Automaten mögliche Konstruktion auf **look** angewendet. In diesem Falle ist $M = Y \cup \{go\}$ im wesentlichen die Menge der Einblicke einer Umwelt U, und der Aktor "**look**$\to\ y$" wird einen Punkt $(P,y) \in U \times M$ überführen in $(P, l(P)) \in U \times Y$. Die erste Koordinate verändert sich nicht, weil "**look**" sich nicht von der Stelle rührt.

Eine Bemerkung noch zur Schreibweise. Es wird häufig vorkommen, dass

$$Y = Y_1 \times \ldots \times Y_i \times \ldots \times Y_n$$

ein kartesisches Produkt ist, aber nur eine einzige Variable v des Typs Y_i mit dem aktuellen Einblick belegt werden soll. Das wird durch anonyme Variable in der Gestalt

$$\mathbf{look} \to (\_, \cdots, \_, v, \_, \cdots, \_)$$

ausgedrückt. Diese Schreibweise wird im folgenden Beispiel verwendet.

EIN ALTES BEISPIEL WIRD KOMPLETTIERT

In 2.2.2 wurde ein Automat beschrieben (vergl. Abbildung 2.10 auf Seite 45), welcher die Inschrift auf einem Turingband um eine Stelle nach links verschiebt. Dabei wurde der sehr einfache Fall eines Alphabets mit nur drei Buchstaben angenommen. Bei einem größeren Alphabet ist es nicht mehr möglich, einen überschaubaren Zustandsgraphen eines Automaten zu zeichnen. Man kann aber für jedes Alphabet ein sehr einfaches Programm aufschreiben, welches die Aufgabe löst. Dabei wird nach demselben Algorithmus wie in 2.2.2 verfahren. Auch die Voraussetzungen werden übernommen: es wird angenommen, dass auf dem Turingband ein unter Umständen auch leeres Wort steht, welches auf dem null-ten Platz anfängt und dessen Ende durch das Blank ♭ bezeichnet ist:

```
actor shift_left
declare v : Σ end declare
begin
  while look_of (1,_) do action (−1) end while
  if not look_of (_,♭) then
    action (1);
    repeat
      look→ (_, v);
      action (-1); action v;
      action (1); action (1)
    until look_of (_,♭);
    action (-1); action ♭
  end if
end
```

Das ist doch eine bemerkenswert elegante Darstellung eines Automaten im Vergleich zu allen anderen Möglichkeiten, die wir kennen gelernt haben. Es bleibt dem Leser überlassen, sich anhand eines Beispiels zu vergewissern, dass dieses Programm korrekt ist.

An den Beispielen sind die wesentlichen Konstrukte unserer Programmiersprache demonstriert worden. Eine Definition der Syntax findet sich im Anhang. Wir betrachten unsere Programmiersprache als einfaches und nützliches Hilfsmittel. Es existiert kein Compiler für sie. Sie wird es ermöglichen, wesentliche Zusammenhänge darzustellen, die auch mit kommerziellen Sprachen getestet werden können. Die Übersetzung in eine solche ist in jedem Falle leicht möglich. Mit dieser Metasprache haben wir jedoch den Vorzug, unabhängig von der raschen Entwicklung kommerzieller Produkte zu sein.

EIN SUCHROBOTER

Wir beenden diesen Abschnitt mit drei weiteren Beispielen.

Beispiel 1

Es ist ein Aktor anzugeben, der sich auf einer ebenen Fläche bewegt und einen Kiesel findet.

Die Fläche wird durch die Umwelt $\mathbb{N} \times \mathbb{N}$ modelliert. Geometrisch bedeutet dies, dass wir uns im ersten Quadranten eines kartesischen Koordinatensystems befinden. Die Ebene ist in Quadrate der Seitenlänge eins aufgeteilt. Punkte der Umwelt sind die Quadrate. Eine solche Rasterumwelt haben wir schon mehrfach benutzt.

In dieser Umwelt eine Marke zu finden gelingt offenbar, wenn man die Diagonalen von links oben nach rechts unten und umgekehrt, am Koordinatenursprung beginnend, systematisch absucht(vergl. Abbildung 2.22)

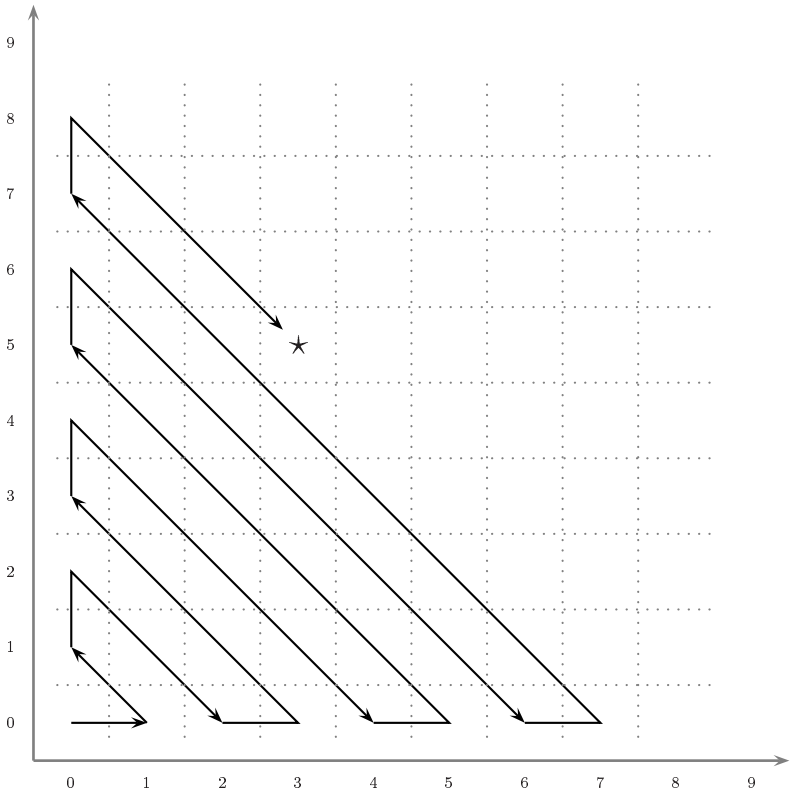


Abbildung 2.22: So findet man den Kiesel

Die Position des Kiesels muss in einem Umweltmodell markiert und erkennbar sein. Deshalb wird die Umwelt $\mathbb{N} \times \mathbb{N}$ modifiziert. Der Suchroboter wird in einer Umwelt

$$U = (X, Y, Z, d, l)$$

arbeiten, die wie folgt definiert ist:

$$
\begin{aligned}
X &= \{0, 1, -1\} \times \{0, 1, -1\} \\
Y &= \{0, 1\} \times \{0, 1\} \times \{0, \star\} \\
Z &= \mathbb{N} \times \mathbb{N} \times \{0, \star\} \\
d((m, n, u), (x_1, x_2)) &= \begin{cases} \perp & \text{für } (m, x_1) = (0, -1) \text{ oder } (n, x_2) = (0, -1) \\ (m + x_1, n + x_2) & \text{sonst} \end{cases} \\
l(m, n, u) &= (sgn\ m, sgn\ n, u)
\end{aligned}
$$

Darin ist die Funktion sgn für natürliche Zahlen so definiert:

$$
sgn\ x = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{sonst} \end{cases}
$$

Der Algorithmus als Programm

Jetzt wird der Suchautomat formuliert:

actor *finde_Marke*
description

{Zuerst geht der Automat zum Koordinatenursprung. Das ist die Startposition für den Suchvorgang. Anschließend wird der Quadrant so abgesucht, wie es Abbildung 2.22 zeigt. }

```
begin
 while look_of (1,_,_) do action (-1,0) end while
 while look_of (_,1,_) do action (0,-1) end while
 while look_of (_,_,0) do
  if look_of (_,_,0) then action (1,0) end if
  while look_of (1,_,0) do action (-1,1) end while
  if look_of (_,_,0) then action (0,1) end if
  while look_of (_,1,0) do action (1,-1) end while
 end while
end
```

Im Abschnitt 2.3.1 wurde gezeigt, dass die Menge der Einblicke, die einem Automaten zur Verfügung stehen, maßgeblich ist für dessen Leistungsfähigkeit. Das kann an diesem Modell erneut demonstriert werden. Wir ändern zunächst die Punktmenge ab. Statt des ersten Quadranten wird die ganze Ebene betrachtet:

$$Z^\star = \mathbb{Z} \times \mathbb{Z} \times \{0, \star\}.$$

Die Aktionen $X = \{0, 1, -1\} \times \{0, 1, -1\}$ bleiben unverändert. Mit der modifizierten Einblicksmenge $Y = (\{0, 1, -1\} \times \{0, 1, -1\} \times \{0, \star\}$ kann man die Aufgabe in analoger Weise wie oben lösen: der Suchweg ist eine "Spirale" um den Ursprung des Koordinatensystems.

Die Situation verändert sich vollständig, wenn man $Y^\star = \{o\} \times \{0, \star\}$ setzt. Die Einblicksfunktion muss entsprechend angepasst werden: $l^\star(m, n, u) := (o, u)$. In der Umwelt des vorangegangenen Beispiels konnte man unterscheiden, ob man sich auf einer der Koordinatenachsen, auf beiden Achsen, also im Ursprung oder woanders befindet. Diese Informationen fehlen in der abgeänderten Umwelt. Ist es denkbar, dass auch unter diesen erschwerten Bedingungen ein Automat konstruiert werden kann, der eine Marke findet? Die Antwort ist nein:

Satz 2.4.1 *Es gibt keinen Automaten, der in der Umwelt* $U = (X, Y^\star, Z^\star, d, l^\star)$ *jeden beliebigen markierten Punkt findet, oder, in anderer Formulierung, zu jedem Automaten gibt es einen Punkt* $(m, n, \star)$*, den der Automat nicht finden kann.*

WIE BEWEIST MAN, DASS ETWAS NICHT GEHT?

Beweis: Sei $\mathfrak{A} = (S, s_0, X, Y^\star, \delta, \lambda)$ ein beliebiger Automat, der nur dann in einen terminalen Zustand gelangt, wenn der Einblick $(o, \star)$ empfangen wird. $\#S = n$ ist die Zahl seiner Zustände.

Wir verfolgen die Rechnung $h : S \times Z \to S \times Z$ des Automaten in U:

$$h(s, P) = (\delta(s, l^\star(P)), P\lambda(s, l^\star(P))) = (\delta(s, (o, u)), P\lambda(s, (o, u))).$$

Ist $m > n$ und ist nach m Arbeitstakten der markierte Punkt noch nicht gefunden, so ist mindestens ein Zustand doppelt erreicht worden: in der Folge

$$(s_0, P_0), (s_1, P_1), \cdots, (s_m, P_m) \text{ mit } (s_{i+1}, P_{i+1}) = h(s_i, P_i)$$

gibt es einen Index j und ein $p > 0$ derart, dass $s_j = s_{j+p}$ ist. Daraus ergibt sich

$$s_{j+1} = \delta(s_j, (o, 0)) = \delta(s_{j+p}, (o, 0)) = s_{j+p+1}$$

und durch vollständige Induktion $s_i = s_{i+p}$ für alle $i \geq j$.

Das wiederum impliziert

$$x_i = \lambda(s_i, (o, 0)) = \delta(s_{i+p}, (o, 0)) = x_{i+p}$$

ebenfalls für alle $i \geq j$.

Mit anderen Worten, solange der Automat den markierten Punkt nicht findet, ist sein Modell von der Umwelt nach dem j-ten Arbeitstakt periodisch mit der Periodenlänge p. Ist zufällig noch $P_j = P_{j+p}$, so hat der Automat auch noch einen periodischen Weg in U. Anderenfalls sind die Punktmengen

$$\mathfrak{P}_i = \{P_{j+ip}, P_{j+ip+1}, \ldots, P_{j+(i+1)p}\}$$

für alle $i \in \mathbb{N}$ die Eckpunkte kongruenter Polygone, und folglich liegen die Punkte $P_j, P_{j+p}, P_{j+2p}, \ldots$ äquidistant auf einer Geraden, welche im wesentlichen die Richtung der Fortbewegung des Automaten ist. Jede Marke, die nicht in einem dieser Eckpunkte deponiert ist, kann von dem Automaten nicht entdeckt werden.

Das war zu zeigen.

In diesem Beispiel wird eine typischer Strategie für Unmöglichkeitsbeweise im Zusammenhang mit Automaten verwendet: es wird ausgenutzt, dass es nur endlich viele Zustände gibt und dass δ und λ wegen der verarmten Einblicksmenge zu einer Funktion nur einer Variablen werden.

Beispiel 2

Wie findet man sich im Dunkeln zurecht

Als weiteres Beispiel für einen Automaten in einer Rasterumwelt greifen wir noch einmal eine der Aufgaben aus 2.2.2 auf Seite 51 auf: ein Roboter soll eine Mauer umrunden. Dort wurde vorgeschlagen, ihn so auszustatten, dass er die vier Himmelsrichtungen ermitteln kann, also über einen Kompass verfügt. Man kann sich aber auch vorstellen, dass in völliger Dunkelheit eine Mauer abgetastet werden muss, wobei man lediglich spüren kann, ob in einer bestimmen Richtung Kontakt zur Mauer besteht oder nicht, ohne zu wissen, ob diese Richtung z.B. Norden ist(vergl. Abbildung 2.23)

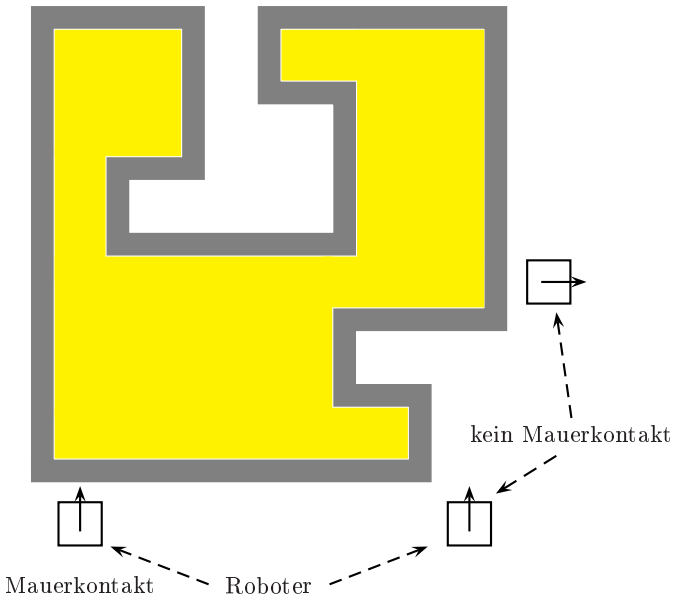


Abbildung 2.23:

Vorausgesetzt wird, dass der Automat so an die Mauer gesetzt wird, dass er Kontakt zu ihr hat. Man überlegt sich, dass der Roboter die Mauer nicht verliert, wenn er nach dem folgenden Rezept arbeitet:

1. Markiere deinen Startpunkt.
2. Wenn es Mauerkontakt gibt, so gehe zu 3., anderenfalls zu 5.
3. Führe Drehungen um 90 Grad nach rechts aus, bis es keinen Kontakt mehr zur Mauer gibt, gehe einen Schritt nach vorn und drehe dich um 90 Grad nach links.
4. Prüfe, ob die Startmarke erreicht ist. Wenn ja, so bleibe stehen. Im anderen Falle verfahre nach 2.

5. Gehe einen Schritt nach vorn, führe eine Linksdrehung um 90 Grad aus und fahre mit 2. fort.

Bei diesem Beispiel gibt es Aktionen, nämlich die Drehungen um 90 Grad, welche die "Blickrichtung" verändern. Bei unserem einfachen Modell beschränkten wir uns auf vier Richtungen. Die Punkte der Umwelt bestehen also jetzt aus Quadrupeln $\mathfrak{P} = (x, y, r, u) \in \mathbb{N} \times \mathbb{N} \times \{r_1, r_2, r_3, r_4\} \times \{0, \star\}$, wo $\star$ wie im vorigen Beispiel eine Marke ist, die in diesem Falle den Startpunkt bezeichnet. Die Drehungen werden die Werte von r beeinflussen, die Vorwärtsbewegung wird das Paar (x, y) verändern, und zwar in der Richtung, die durch r gegeben ist.

Nach diesen heuristischen Überlegungen bietet es sich an, das Problem in der komplexen Zahlenebene $\mathbb{Z} + i\mathbb{Z}$ zu behandeln. Für die Richtungen nehmen wir die Menge $R = \{1, -1, i, -i\}$. Die Mauer M ist eine Teilmenge von $\mathbb{Z} + i\mathbb{Z}$.

$Z = (\mathbb{Z} + i\mathbb{Z} \setminus M) \times R \times \{0, \star\}$ ist die Punktmenge der Umwelt. Aktionen sind $X = \{0, step, i, -i, \star\}$, und d ist definiert durch

$$\begin{aligned} d((x + iy, r, u), step) &:= (x + iy + r, r, u) \\ d((x + iy, r, u), \pm i) &:= (x + iy, \pm i \cdot r, u) \\ d((x + iy, r, u), \star) &:= (x + iy, r, \star) \end{aligned}$$

Durch die dritte Gleichung wird das Ablegen der Marke organisiert. Einblicke sind $Y = \{0, m\} \times R \times \{0, \star\}$, die Einblicksfunktion $l : Z \to Y$ ist definiert durch

$$l(P) = l(x + iy, r, u) = \begin{cases} (m, r, u) & \text{wenn } x + iy \in M \\ (0, r, u) & \text{sonst} \end{cases}$$

Nun das Programm in der Automatensprache:

actor *robot*
description

{Für die vier Richtungen wird eine Variable benutzt, wogegen die Marken 0 und $\star$ als zweipunktige Umwelt durch **look** und **action** zusammen mit $x + iy \in \mathbb{Z} \times \mathbb{Z}$ erfasst werden. In der Initialisierungsphase des Programms wird die Marke gesetzt und die "Roboternase" zur Mauer gerichtet. In der repeat-Schleife erfolgt die Umrundung der Mauer.}

declare $r : \{1, -1, i, -i\}$ **end declare**
begin
action$(0, \star)$
$r := 1$
while look_of $(0, \_)$ **do** $r := i \cdot r$ **end while**;
repeat

```
  if look_of (m, _) then
   repeat r := −i · r until look_of (0,);
   action (step)
    else action (step)
  end if
  r := i · r;
 until look_of (_, ⋆)
end
```

An diesem Beispiel dürfte klar geworden sein, welche Bedeutung einer gründlichen Vorüberlegung bei der Lösung eines Problems zukommt. Es ist immer gut, wenn man an eine Aufgabe zunächst einmal ganz naiv heran geht. Im vorliegenden Beispiel war es z.B. eine gute Idee für die endgültige Lösung, sich vorzustellen, dass man ohne jede Lichtquelle die Mauer ertasten muss. Daraus ergibt sich nach einigem Nachdenken zwangsläufig der Lösungsweg.

Beispiel 3

Um die wichtigen Registerumwelten zu wiederholen und in die sprachliche Darstellung von Automaten einzubeziehen, greifen wir den Multiplizierer auf Seite 48 wieder auf. Die Figur 2.11 ist quasi das R-Schema des Prozessors und führt uns direkt zum Programm. Wir verändern allerdings hier und in Zukunft öfter unsere Schreibweise insofern, als das Schlüsselwort **action** weg gelassen wird, weil die Bezeichnung der Aktionen selbst hinreichend ausdrucksfähig ist.

```
actor mult
begin
  sub p p;
  lod q 1;
  lod r 1;
  add r r;
  sub [r] [r];
  eq [p] p;
  if look_of 0 then;
    eq [q] p;
    while look_of 0 do
      add [r] [p];
      sub [q] q;
      eq [q] p
    end while
  end if
end
```

2.4.4 Für jeden Automaten gibt es ein Programm

Wir haben angekündigt: die Automatensprache ist universell. Jeder Automat lässt sich durch ein Programm beschreiben. Das soll jetzt bewiesen werden.

Satz 2.4.2 *Zu jeder R-Maschine* $(\mathfrak{A}, U)$ *gibt es ein isomorphes Programm.*

Mit dieser verkürzten Sprechweise soll ausgedrückte werden, dass das Programm eine Maschine darstellt, die zur ursprünglichen Maschine isomorph ist.

Beweis:

Sei $U = (X, Y, Z, d, l)$ eine R-Umwelt und $\mathfrak{A} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ ein beliebiger R-Automat über dieser Umwelt. Zu dieser Maschine wird eine endliche, punktierte Umwelt

$$(U', O') = (X', Y', Z', d', l')$$

in folgender Weise definiert:

- $X' := Y;\ Z' := S =: Y';\ O' := s_0$
- $d'(P', x') = d'(s, y) := \delta(s, y)$
- $l'(P') = l'(s) := s$

Damit U' eine R'-Umwelt wird, legen wir fest:

$$R'(s) := \begin{cases} X' = Y & \text{für } s \in act\ \mathfrak{A} \\ \emptyset & \text{für } s \in term\ \mathfrak{A} \end{cases}$$

Diese Umwelt U' ist nichts anderes als der Automat $\mathfrak{A}$, beschrieben als Umwelt. Mit diesem Trick wird es im folgenden Programm möglich, mittels der uns bekannten Elementarautomaten - man vergleiche z.B. die Anwendung der *case*-Klausel - das Verhalten der Maschine $(\mathfrak{A}, U)$ zu simulieren.

Nun wird über der Umwelt $U' \times U$ ein Programm definiert, von dem man zeigen kann, dass es der Behauptung des Satzes genügt:

JEDER MASCHINE IHR PROGRAMM

```
automaton programm_zu_𝔄 returns_label M
declare σ : term 𝔄 end declare
begin
 while look_of act 𝔄 × Y do
  case of look
    ...
   (s_i, y_j) then action (y_j, λ(s_i, y_j))|
    ...
  end case
```

end while;
look$\rightarrow (\sigma, \_)$;
return_label $\mu(\sigma)$
end

Man muss an dieser Stelle vielleicht auf einen kleinen aber wichtigen Unterschied aufmerksam machen: die *case*-Klausel benutzt den Einblicksautomaten "**look**", um in Abhängigkeit von dem gelieferten Einblick etwas zu tun. Das ist wohl zu unterscheiden von dem eine Zeile vorher eingesetzten booleschen Automaten "**look_of** $act\ \mathfrak{A} \times Y$".

Die abkürzende Schreibweise des *case*-Konstrukts bedarf ebenfalls einer Erklärung: Wenn $act\ \mathfrak{A} = \{s_0, s_1, \ldots, s_n\}$ und $Y = \{y_1, y_2, \ldots, y_m\}$ ist, so gibt es dort zu jedem Paar $(s_i, y_j) \in act\ \mathfrak{A} \times Y$ eine Zeile, insgesamt also $(n+1) \cdot m$ Fallunterscheidungen.

Wie muss der Beweis geführt werden? Man benötigt den Automaten, welcher dem Programm entspricht. Den werden wir mit $\bar{\mathfrak{A}}$ bezeichnen. Zusammen mit seiner Umwelt ist das die Maschine $(\bar{\mathfrak{A}}, U' \times U)$. Und von dieser Maschine ist nachzuweisen, dass sie zu $(\mathfrak{A}, U)$ isomorph ist.

Wie man zu einem Programm den Zustandsgraph des entsprechenden Automaten gewinnt, haben wir prinzipiell erläutert. Im vorliegenden Fall bringen wir nur mit Abbildung 2.24 das Ergebnis. In der Figur sind die $t_i;\ 1 \leq i \leq p$ die terminalen Zustände von $\mathfrak{A}$

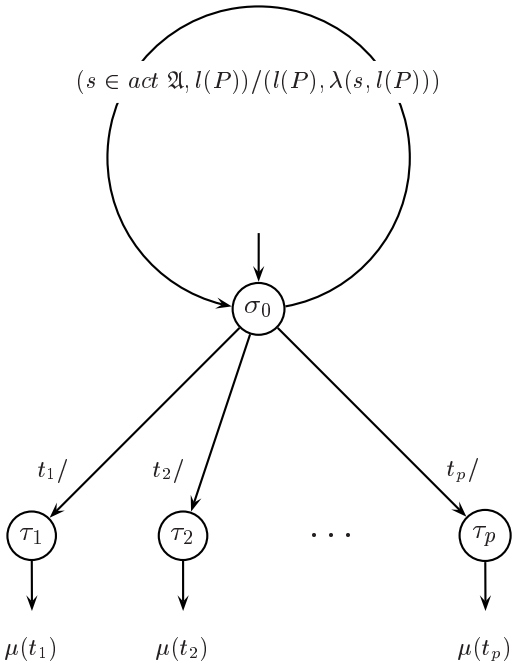


Abbildung 2.24: Zustandsgraph zum Programm

Es ist also

$$\bar{\mathfrak{A}} = (\bar{S}, \sigma_0, \bar{X}, \bar{Y}, M, \bar{\delta}, \bar{\lambda}, \bar{\mu})$$

mit

$$\begin{aligned} \bar{S} &= \{\sigma_0, \tau_1, \tau_2, \ldots, \tau_p\}; \quad act\, \bar{\mathfrak{A}} = \{\sigma_0\} \\ \bar{X} &= X' \times X = Y \times X \\ \bar{Y} &= Y' \times X = S \times Y \\ \bar{\mu}(\tau_i) &= \mu(t_i) \end{aligned}$$

$$\bar{\delta} : \bar{S} \times S \times Y \xrightarrow{\supset} \bar{S}$$

$$(\sigma_0, (l'(s), l(P))) = (\sigma_0, (s, l(P))) \mapsto \bar{\delta}(\sigma_0, (s, l(P))) = \begin{cases} \sigma_0 & \text{für } s \in act\, \mathfrak{A} \\ \tau_i, & \text{falls } t_i \in term\, \mathfrak{A} \end{cases}$$

$$\bar{\lambda} : \bar{S} \times S \times Y \xrightarrow{\supset} \bar{X} = Y \times X$$

$$(\sigma_0, (s, l(P))) \mapsto \bar{\lambda}(\sigma_0, (s, l(P))) = \begin{cases} (l(P), \lambda(s, (l(P)))) & \text{für } s \in act\, \mathfrak{A} \\ (stop, stop) & \text{sonst} \end{cases}$$

Wesentlich für die Isomorphie ist die Rechnung eines Automaten in seiner Umwelt (vergl. Formel 2.2 auf Seite 60). Jetzt sind wir in der Lage, für $(\bar{\mathfrak{A}}, U' \times U)$ die entsprechenden Gleichungen aufzuschreiben. Dabei wird für die Umwelt $U' \times U$ in Analogie zu $\bar{\mathfrak{A}}$ die Schreibweise $\bar{U} := U' \times U = (\bar{X}, \bar{Y}, \bar{Z}, \bar{d}, \bar{l})$ verwendet:

$$\bar{h} : \bar{S} \times S \times Z \xrightarrow{\supset} \bar{S} \times S \times Z$$

$$\begin{aligned} \bar{h}(\sigma_0, (s, P)) &= (\bar{\delta}(\sigma_0, \bar{l}(s, P)), \bar{d}((s, P), \bar{\lambda}(\sigma_0, \bar{l}(s, P)))) \\ &= (\bar{\delta}(\sigma_0, (s, l(P))), \bar{d}((s, P), \bar{\lambda}(\sigma_0, (s, l(P))))) \\ &= \begin{cases} (\sigma_0, (\delta(s, l(P)), P\lambda(s, l(P)))) & \text{für } s \in act\, \mathfrak{A} \\ (\tau_i, (t_i, P)), & \text{wenn } t_i \in term\, \mathfrak{A}\ (1 \leq i \leq p) \end{cases} \end{aligned}$$

Die letzte Gleichung wird mit der Rechnung von $\mathfrak{A}$ in U verglichen:

$$\begin{aligned} h : act\, \mathfrak{A} \times Z &\to S \times Z \\ (s, P) &\mapsto (\delta(s, l(P)), P\lambda(s, l(P))) \end{aligned}$$

Nun kann man erkennen, wie die beiden Maschinen in Beziehung zu setzen sind. Die bijektive Abbildung, welche die Konfigurationen beider Maschinen zuordnet, ist

$$\varphi : S \times Z \to (\{\sigma_0\} \times S \times Z) \cup (\{(\tau_i, t_i) \mid t_i \in term\, \mathfrak{A}\} \times Z)$$

$$\varphi(s, P) = \begin{cases} (\sigma_0, s, P) & \text{für } s \in act\, \mathfrak{A} \\ (\tau_i, t_i, P), & \text{wenn } s = t_i \in term\, \mathfrak{A} \end{cases}$$

In der Tat gilt damit $\varphi \circ h = \bar{h} \circ \varphi$ Das wollten wir beweisen.

2.4.5 Die universelle Turingmaschine

Wir sind mit unserer Sprache in der Lage, für viele elementare Funktionen Automaten anzugeben, welche die Berechnung der Funktionswerte routinemäßig ausführen. Aber wir brauchen zu jeder neuen Aufgabe einen neuen Automaten. Die Idee, einen Automaten zu konstruieren, der eine große Klasse von Aufgaben lösen kann, gehört zu den hervorragenden Leistungen in der Geschichte der Informatik. Schon Babbage (vgl. die Einleitung) hatte seine Maschinen im Grunde nach dem Prinzip einer universellen Maschine entworfen. Die theoretische Begründung der Funktionsfähigkeit eines solchen Gerätes stammt von Turing, weshalb man sie heute auch universelle Turingmaschine nennt. Das war in den dreißiger Jahren des vorigen Jahrhunderts. Damals war von Informatik noch keine Rede. An automatisch arbeitende Rechner hat man zwar schon gedacht, aber deren Realisierung lag noch in der Zukunft. John von Neumann hat um 1940 elektronische Computer entworfen, die nach dem Prinzip der universellen Maschine arbeiten. Seitdem spricht man von der "von-Neumann-Architektur" eines Computers.

Das Prinzip

Worin besteht diese Architektur, die uns so vertraut ist, dass deren Bedeutung heute oft gar nicht mehr erkannt wird? Ein universeller Computer muss ein Automat sein, der unterschiedliche Programme speichern und abarbeiten kann. Beliebige Automaten, durch ihre Programme gegeben, werden vom universellen Prozessor in Gang gesetzt und erledigen ihre Arbeit. Man muss nur dafür sorgen, dass das entsprechende Programm von universellen Automaten verstanden wird. Es muss in einer Sprache abgefasst sein, welche der universelle Automat übersetzen, also compilieren kann.

Wir sind jetzt in der Lage, einen solchen universellen Automaten mit unseren Mitteln zu schreiben. Es werden also die hier definierten Modelle von Automat und Umwelt benutzt. Dennoch lässt sich auf diese Weise die prinzipielle Arbeitsweise eines modernen Computers demonstrieren. Allerdings werden wird dazu keine Turingumwelt, sondern eine Registerumwelt benutzen.

Welche Arbeitsphasen sind zu absolvieren? In einem ersten Schritt muss das von der universellen Maschine zu steuernde Programm in eine lineare Gestalt gebracht werden. Jede Programmzeile enthält genau eine Anweisung. Und nach jeder Zeile wird die Anweisung der folgenden Zeile abgearbeitet.

Diese linearisierte Form des ursprüngliche Programms wird auf den ungeraden Registern der Umwelt gespeichert. Die für eine Rechnung erforderlichen Daten legen wir auf die geraden Registerplätze. Das universelle Programm ist nun in der Lage, mit dem gespeicherten Programm zu arbeiten.

Wenden wir uns also zunächst der Linearisierung eines Programms zu. Dazu erinnern wir noch einmal, welche elementaren Konstrukte wir in unseren Programmen verwenden.

LINEARISIERUNGSBEISPIELE

1. Im einfachsten Falle finden sich in einem Programm hintereinander zwei Anweisungen der Gestalt

   action x_1; **action** x_2

   Dafür steht im linearisierten Programm

   $$\begin{array}{ll} \vdots & \vdots \\ n_k & \textbf{action } x_1 \\ n_{k+1} & \textbf{action } x_2 \\ \vdots & \vdots \end{array}$$

   Dabei sind n_k und n_{k+1} Zeilennummern, die gleichzeitig als Sprungmarken benutzt werden (vergl. das folgende Konstrukt).

2. Eine Verzweigung

   if look_of 1 **then** $\mathfrak{A}$ **end if**

   erhält die Gestalt

   $$\begin{array}{ll} \vdots & \vdots \\ n_k & \textbf{jz } n_s \\ n_{k+1} & \\ \vdots & \mathfrak{A} \\ n_{s-1} & \\ n_s & \\ \vdots & \vdots \end{array}$$

   Darin ist $\mathfrak{A}$ ein Aktor und **jz** ist Kürzel für **jump zero**. Die Zeile "n_k **jz** n_s" ist in folgender Weise zu lesen: wenn die Variable w für die Wahrheitswerte der Registerumwelt mit dem Wert null belegt ist, so springe sofort zur Zeile n_s, ohne dich um die Zeilen dazwischen, also die Ausführung von $\mathfrak{A}$ zu kümmern. Wenn in der *if_then*-Zeile dagegen **look_of** 0 steht, dann muss der Sprungbefehl **jnz** heißen, was Abkürzung für **jump no zero** ist.

3. Die beiden bisher benutzten Sprungbefehle heißen in nahe liegender Weise **bedingte** Sprungbefehle. Bei der Übersetzung der *if_then_else*-Verzweigung wird zusätzlich der **unbedingte** Sprungbefehl **jmp**(=jump) benutzt:

   if look_of 0 **then** $\mathfrak{A}_1$ **else** $\mathfrak{A}_2$ **end if**

   erhält die Gestalt

$$
\begin{array}{ll}
\vdots & \vdots \\
n_p & \textbf{jnz } n_{r+1} \\
n_{p+1} & \mathfrak{A}_1 \text{ beginnt} \\
\cdots & \cdots \\
n_{r-1} & \mathfrak{A}_1 \text{ beendet} \\
n_r & \textbf{jmp } n_s \\
n_{r+1} & \mathfrak{A}_2 \text{ beginnt} \\
\cdots & \cdots \\
n_{s-1} & \mathfrak{A}_2 \text{ beendet} \\
n_s & \cdots \\
\vdots & \vdots
\end{array}
$$

Mit diesen Beispielen ist hinreichend erläutert, wie ein Programm in eine lineare Form umgesetzt werden kann. Die übrigen Strukturelemente (*while*− und *repeat*−Schleife) werden in analoger Weise behandelt.

Das folgende Beispiel mag den Vorgang der Linearisierung und die nachfolgenden Arbeitsschritte veranschaulichen. Aufgabe sei es, die Funktionswerte eines Polynoms nach dem so genannten Horner-Algorithmus zu bestimmen. Wir werden uns auf Polynome mit ganzzahligen Koeffizienten beschränken. Dieses Verfahren beruht auf der folgenden Formel:

$$
\begin{aligned}
p(x) &= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0 \\
&= ((\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots a_2)x + a_1)x + a_0
\end{aligned}
$$

Man erkennt sofort den Vorteil der Darstellung des Polynoms in der zweiten Zeile: die Berechnung eines Funktionswertes zu einem gegebenen Argument x wird auf zwei einfache und ständig wiederholte Operationen, nämlich Addition und Multiplikation reduziert.

Um dieses Verfahren mit einer Registerumwelt zu realisieren, werden die fünf Arbeitsregister p, q, r, s und t benutzt. Auf den übrigen Registerplätzen - nennen wir sie RAM-Register - liegen die Daten für die Berechnung, wobei nur die mit geraden Zahlen nummerierten Register benutzt werden. Die ungeraden Speicherplätze sollen ja für das linearisierte Programm reserviert bleiben:

$$f(0) = x,\ f(2) = n,\ f(4) = a_n,\ f(6) = a_{n-1}, \cdots$$

Dazu geben wir folgendes Programm an:

Z_nr.	w	p	q	r	s	t	0	2	4	6	8	10
Start	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	x	2	a_2	a_1	a_0	$\cdots$
(1)-(3)	1	0	2	4	0	a_2	x	2	a_2	a_1	a_0	$\cdots$
(4)	1	0	2	4	0	a_2x	x	2	a_2	a_1	a_0	$\cdots$
(5)-(6)	1	0	2	6	0	$a_2x + a_1$	x	2	a_2	a_1	a_0	$\cdots$
(7)-(8)	1	0	2	6	1	$a_2x + a_1$	x	2	a_2	a_1	a_0	$\cdots$
(4)-(8)	0	0	2	8	2	$a_2x^2 + a_1x + a_0$	x	2	a_2	a_1	a_0	$\cdots$

Tabelle 2.7: Weg des Horner-Automaten

```
      actor horner
      begin
(1)    sub p p; lod q 2; mov r q;
(2)    add r q; sub s s; mov t [r];
(3)    less s [q];
       while look do
(4)      mult t [p];
(5)      add r q;
(6)      add t [r];
(7)      inc s;
(8)       less s [q]
       end while
      end
```

Die Zeile "**while look do**" steht anstelle von "**while look_of** 1 **do**". Das ist eine in professionellen Programmiersprachen übliche Ersetzung. Die Aktion "inc s" erhöht den Wert von s um 1. Die Arbeit dieses Programmes testen wir mit dem Startpunkt, wie er in Tabelle 2.7 angegeben ist. Sie enthält auch zeilenweise Punkte der Umwelt, die der Automat absolviert.

Eine mögliche Linearisierung dieses Programms hat die folgende Gestalt:

```
01: sub p p          08: jz 14
02: lod q 2          09: mult t [p]
03: mov r q          10: add r q
04: add r q          11: add t [r]
05: sub s s          12: inc s
06: mov t [r]        13: jmp 07
07: less s [q]       14: end
```

Man kann sich überzeugen, dass mit diesem Programm ebenfalls die Zeilen der Tabelle 2.7 entstehen.

Nun ist der Zeitpunkte gekommen, das Programm des universellen Automaten zu formulieren. Er muss in der Lage sein, alle arithmetischen Operationen einer Registerumwelt zu erkennen. Zu diesem Zweck werden sie kodiert. In einem professionellen Programm ist das ein beachtliches Unterfangen. Da es uns hier

Registerplatz	Befehle/ Daten	Kode	Registerplatz	Befehle/ Daten	Kode
00	x		16		
01	sub p p	(6)	17		31
02	n		18		
03	lod q 2	(5)	19	inc s	(4)
04	a_n		20		
05	mov r q	(7)	21	mult t p	(10)
06	a_{n-1}		22		
07	add r q	(8)	23	add r q	(8)
08	a_{n-2}		24		
09	sub s s	(9)	25	add t [r]	(3)
10	a_{n-3}		26		
11	mov t r	(2)	27	jmp 13	(11)
12	a_{n-4}		28		
13	less s [q]	(1)	29		13
14			30		
15	jz 31	(12)	31	end	(14)

Tabelle 2.8: Die Belegung der RAM-Register für *horner*

nur um das Prinzip geht, beschränken wir uns auf die im Beispiel verwendeten Operationen. Außerdem ist unsere Kodierung denkbar simpel: wir ordnen den Operationen natürliche Zahlen zu:

less s [q]	$\rightarrow$	(1)	add r q	$\rightarrow$	(8)
mov t [r]	$\rightarrow$	(2)	sub s s	$\rightarrow$	(9)
add t [r]	$\rightarrow$	(3)	mult t [p]	$\rightarrow$	(10)
inc s	$\rightarrow$	(4)	jmp	$\rightarrow$	(11)
lod q 2	$\rightarrow$	(5)	jz	$\rightarrow$	(12)
sub p p	$\rightarrow$	(6)	jnz	$\rightarrow$	(13)
mov r q	$\rightarrow$	(7)	end	$\rightarrow$	(14)

Nach diesen Vorbereitungen kann man die Belegung der RAM-Register angeben. Sie ist der Tabelle 2.8 zu entnehmen.

Die Belegung der geradzahligen RAM-Register größer 2 hängt natürlich von der Zahl n im zweiten Register ab. Die Eintragung der Koeffizienten wurde bei a_4 abgebrochen. Bei den Kodierungen der Befehle auf den ungeradzahligen Registern gibt es Unregelmäßigkeiten bei den Sprungbefehlen, die folgenden Grund haben: "jz n" ist keine Aktion der Registerumwelt, weil n die Nummer jeder Zeile beliebig langer Programme sein kann. Die Zahl der Aktionen wäre beliebig groß im Gegensatz zur Definition unseres Umweltmodells. Aus diesem Grund erscheint die Sprungadresse in der übernächsten Zeile. Um sie von den Befehlen zu unterscheiden, sind die Klammern weg gelassen.

Das Programmpaket für den universellen Automaten ist auf der folgenden Seite zusammengefasst. Die Kommentare zu den einzelnen Teilen werden sicherlich nicht ausreichen, um vollständig zu erfassen, wie der Prozessor arbeitet. Das volle Verständnis wird am einfachsten dadurch erreicht, dass man versucht, das Programm schrittweise zu verfolgen und die Tabelle 2.7 mit dem universellen Prozessor zu reproduzieren. Die Spalten der Arbeitsregister müssen dann entsprechend erweitert werden.

Wem dieser "Trockentest" zu langweilig ist, kann das Ganze in eine Hochsprache übersetzen und am Rechner testen. Es gibt bereits von Studenten verfasste Java-Programme, welche dem vorgestellten Automaten genau entsprechen und korrekte Ergebnisse liefern.

Das Programm des universellen Automaten

```
actor universeller_automat
description {Aufgaben der zusätzlichen Arbeitsregister : α ist immer 1;
β ist Zählregister ; γ sichert die Werte von w; ε registriert den terminalen
Zustand; π ist Programmzeilenzähler.}
begin
 lod ε 0; mov π ε;
 inc π; mov α π; lod γ 0;
 repeat
  eq α γ;
  agiere;
  if look then
   lod γ 1 else lod γ 0
  end if
  eq ε α;
 until look
end

actor agiere
begin
 case of dekoder
  1 then less s [q]; inc π; inc π|
  2 then mov t [r]; inc π; inc π|
  3 then add t [r]; inc π; inc π|
  4 then inc s; inc π; inc π|
  5 then lod q 2; inc π; inc π|
  6 then sub p p; inc π; inc π|
  7 then mov r q; inc π; inc π|
  8 then add r q; inc π; inc π|
  9 then sub s s; inc π; inc π|
  10 then mult t [p]; inc π; inc π|
  11 then jmp|
  12 then jz|
  13 then jnz|
  14 then end
 end case
end

automaton dekoder returns_label M
description
{Im Beispiel ist M = {1, 2, . . . , 14}}.
declare m : M end declare
begin
 m:=1; lod β 0; inc β;
 if look then
  lod γ 1 else lod γ 0
 end if;
 eq [π] β;
 while look_of 0 do
  m:=m+1;
  inc β;
  eq [π] β
 end while;
 eq γ α;
 return_label m
end

actor jmp
begin
 inc π;inc π;
 mov π [π]
end

actor jz
begin
 if look then
  inc π; inc π; inc π; inc π else jmp
 end if
end

actor jnz
begin
 if look then
  jmp else inc π; inc π; inc π; inc π
 end if
end

actor end
begin lod ε α end
```

Aufgaben

1. Durch vollständige Induktion ist zu zeigen, dass die Folge, welche wir das Verhalten eines Automaten in seiner Umwelt nennen, allein vom Startpunkt in der Umwelt abhängt! (vergl. Definition 2.3.2 auf Seite 57)

2. Der Satz 2.3.1 auf Seite 58 ist zu beweisen!

3. Die Gleichung 2.3 auf Seite 61 ist anhand von Beispielen zu bestätigen!

4. Warum reicht es in der Definition **??** auf Seite **??** nicht aus, lediglich zu fordern, dass φ eine bijektive Abbildung von $S_1 \times Z_1$ auf $S_2 \times Z_2$ ist?

5. Die Isomorphie von Maschinen ist eine Äquivalenzrelation, das heißt, sie hat die drei folgenden Eigenschaften:

   (a) Reflexivität: Jede Maschine ist zu sich selbst isomorph $M \simeq M$

   (b) Symmetrie : Wenn $M_1 \simeq M_2$ so $M_2 \simeq M_1$

   (c) Transitivität: Wenn $M_1 \simeq M_2$ und $M_2 \simeq M_3$, so $M_1 \simeq M_3$

   Die Eigenschaften sind zu beweisen!

6. Die Korrektheit des Programms für den Aktor *div_3* in 2.4.1 ist zu beweisen!

7. Es ist ein Programm zu schreiben für einen Aktor, der in einer Turingumwelt eine am Anfang des Bandes stehende Inschrift um eine Stelle nach rechts verschiebt!

8. Wie findet man in $\mathbb{Z} \times \mathbb{Z}$ eine Marke? Man schreibe ein entsprechendes Programm!

9. Sei $\mathbb{Z}_{(\Sigma,\flat)}$ die Turingumwelt mit $\mathbb{Z}$ als Basisumwelt und mit der Alphabetumwelt über dem Alphabet $\Sigma = \{|, \flat\}$. In dieser Umwelt kann man ganze Zahlen durch Wörter darstellen, die aus dem einzigen Buchstaben $''|''$ gebildet sind. Die Zahl -3 kann etwa durch jeden Punkt $\mathfrak{P} = (n, f)$ dargestellt werden, wo n eine beliebige ganze Zahl ist und

$$f(z) = \begin{cases} | & \text{für } -3 \leq z \leq -1 \\ \flat & \text{sonst.} \end{cases}$$

   Man schreibe ein Programm für einen Aktor, welcher in dieser Umwelt die Subtraktion in $\mathbb{Z}$ simuliert!

10. Die Aufgaben 3 bis 6, 8 und 10 auf Seite 51 sollen in Gestalt eines Programms gelöst werden!

Kapitel 3

Berechenbare Funktionen

Zu Beginn des 20. Jahrhunderts wurde – ausgelöst durch ein von David Hilbert gestelltes Problem(vergl. die Einleitung) – die Bestimmung dessen, was man unter einem Algorithmus zu verstehen hat, zu einer weltweiten Fragestellung. Die internationale Arbeit verschiedener Mathematiker – von Informatik sprach damals noch niemand – erbrachte wichtige Resultate, von denen wir in der Folge einiges in verallgemeinerter Form darstellen werden, wobei wir uns an der von Alan Turing begründeten Vorgehensweise orientieren. Für ihn war eine Funktion berechenbar, wenn es eine Maschine des von ihm vorgeschlagenen Typs, also ein Turingband und dazu einen geeigneten Prozessor gab, welche in der Lage war, die Funktionswerte zu ermitteln. Die folgenden Überlegungen werden insofern allgemeiner sein, weil beliebige Umwelten zugelassen werden.

3.1 Der Berechenbarkeitsbegriff

Im Abschnitt 2.2.2 auf Seite 35 werden die Leseumwelt R_Σ und die Schreibumwelt W_Σ definiert, die jetzt benutzt werden.

Wir erläutern zunächst das Prinzip, die grundlegende Idee (Abbildung 3.1). Es wird angenommen, dass ein Prozessor $\mathfrak{A}$ in einer Umwelt $\mathbf{U} = R_\Sigma \times U \times W_\Sigma$ mit $(U, O) = (X, Y, Z, d, l)$ arbeitet. Die punktierte Umwelt (U, O) wollen wir die **Rechenumwelt** des Prozessors nennen. Lese- und Schreibumwelt sollen das leere Wort ε als ausgezeichneten Punkt haben.

DIE IDEE

$\mathfrak{A}$ empfängt also Signale $\mathbf{y} = (\sigma, y, \sigma') \in (\Sigma \cup \varepsilon) \times Y \times (\Sigma' \cup \varepsilon)$ und führt entsprechende Aktionen $\mathbf{x} = (x_1, x_2, x_3) \in \{pop, stop\} \times X \times (\Sigma' \cup \{stop\})$ aus. Wird $\mathfrak{A}$ in einem Punkt $\mathbf{P} = (w, O, \varepsilon)$ dieser Umwelt gestartet und beendet seine Arbeit ordnungsgemäß, so ist der finale Punkte $\tilde{\mathbf{P}} = \mathbf{P}\mathfrak{A} = (\tilde{w}, Q, u)$ wohlbestimmt und hängt nur von w ab. Die Maschine $(\mathfrak{A}, \mathbf{U})$ definiert eine partielle Funktion

$$\begin{aligned} \xi_{(\mathfrak{A},(U,O))} : R_\Sigma &\xrightarrow{\supset} W_{\Sigma'} \\ w &\mapsto u = \xi_{(\mathfrak{A},(U,O))}(w) \end{aligned}$$

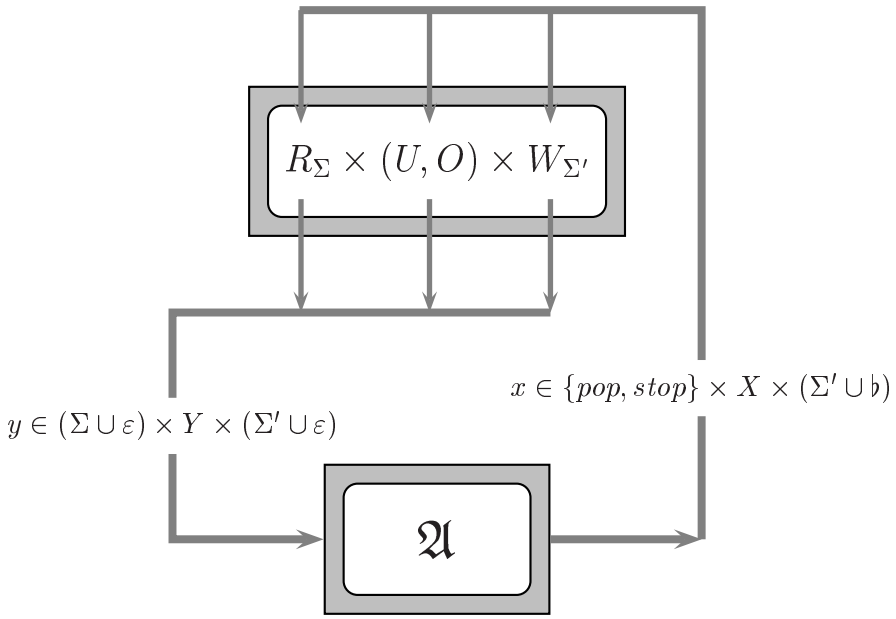


Abbildung 3.1: Modell einer Rechenanlage

Der Index $(\mathfrak{A}, (U, O))$ drückt aus, dass R_Σ und W_Σ festgehalten werden. Von wesentlichem Interesse ist der Einfluss der Rechenumwelt. Wir sagen, der Prozessor $\mathfrak{A}$ berechnet die Funktion $\xi_{(\mathfrak{A},(U,O))}$ mit der Umwelt (U, O). Die Funktion ist partiell, weil nichts über die Ausführbarkeit der Operationen vorausgesetzt wurde. In der nachfolgenden Definition wird das präzisiert. (U,O) wird eine R-Umwelt sein. In diesem Falle wissen wir (vergl. 2.3.2), dass der Automat in jedem aktiven Zustand eine Antwort abliefert, wenn er aus der Umwelt einen Einblick erhält. Aber selbst dann wird es sich um eine partielle Funktion handeln, weil der Prozessor u.U. keinen terminalen Zustand erreicht.

Für die anschließende Definition wird die Sachlage noch etwas verallgemeinert. (U_1, O_1), (U, O) und (U_2, O_2) seien punktierte R_1-, R- bzw. R_2-Umwelten. Dann ist $(\mathbf{U,O}) = (U_1 \times U \times U_2, (O_1, O, O_2))$ eine punktierte $\mathbf{R}$-Umwelt mit $\mathbf{R} = R_1 \times R \times R_2$. $\mathfrak{A}$ sei ein $\mathbf{R}$-Prozessor, agiere also über der Umwelt $U_1 \times U \times U_2$ und werde im Punkte $\mathbf{P} = (P, O, O_2)$ gestartet.

Wie oben ist dann, sofern $\mathfrak{A}$ terminiert, der Punkt $\mathbf{Q} = (\tilde{P}, \tilde{Q}, Q)$ wohlbestimmt, und es wird damit eine Funktion

$$\xi_{(\mathfrak{A},(U,O))} : U_1 \xrightarrow{\supset} U_2$$
$$P \mapsto Q = \xi_{(\mathfrak{A},(U,O))}(P)$$

definiert.

Mit diesen Vorbereitungen ist die folgende Definition motiviert:

GENAUE BEGRIFFSBESTIMMUNG

Definition 3.1.1 *Sei* $(\boldsymbol{U,O}) = (U_1 \times U \times U_2, (O_1, O, O_2))$ *eine R-Umwelt,* $\mathfrak{A}$ *ein R-Prozessor und* $\xi_{(\mathfrak{A},(U,O))}$ *die zugehörige "Rechenfunktion". Eine partielle Funktion* $f : U_1 \stackrel{\supset}{\longrightarrow} U_2$ *heißt* (U,O)***-berechenbar****, falls es einen R-Prozessor* $\mathfrak{A}$ *gibt mit* $f = \xi_{(\mathfrak{A},(U,O))}$

Mit dieser Definition ist ein sehr weit gefasster Begriff formuliert. Jeder Prozessor, das heißt jedes Programm generiert eine berechenbare Funktion. Ein und dasselbe Programm kann aber mit unterschiedlichen Umwelten U zu verschiedenen Funktionen $\xi_{(\mathfrak{A},(U,O))}$ führen. Es wird sich zeigen, dass die Menge der in diesem Sinne berechenbarer Funktionen entscheidend von U bestimmt wird.

Die Definition wird spezifiziert:

- Eine zahlentheoretische Funktion $f : \mathbb{N}^k \stackrel{\supset}{\longrightarrow} \mathbb{N}$ heißt (U,O)-berechenbar, falls f über der Umwelt $\mathbb{N}^k \times U \times \mathbb{N}$ berechenbar ist.
- Wenn $(U,O) = (T_\Sigma, (0,\omega))$ mit $\omega(n) = \flat$ für alle $n \in \mathbb{N}$ ist, so heißt f **Turing-berechenbar**
- Eine Wortfunktion $f : \Sigma^* \stackrel{\supset}{\longrightarrow} \Sigma'^*$ heißt (U,O)-berechenbar, falls sie (U,O)-berechenbar ist für $(U_1,O_1) = (R_\Sigma, \varepsilon)$ und $(U_2,O_2) = (W_{\Sigma'}, \varepsilon)$.
- Eine Funktion $f : U_1 \stackrel{\supset}{\longrightarrow} U_2$ heißt **RAM-berechenbar**, falls sie $(RAM, (0,\omega))$-berechenbar ist. Darin ist ω die Funktion, welche alle Speicherzellen mit Null besetzt: $\omega(n) = 0$ für alle $n \in \mathbb{N}$

ZUM BEISPIEL

Das folgende Beispiel illustriert die Definition: Es werden mit den Möglichkeiten der Umwelt $\mathbb{N}$ natürliche Zahlen multipliziert. Dabei wird angenommen, dass die beiden Faktoren m und n in einer Umwelt $U_1 = \mathbb{N} \times \mathbb{N}$ gegeben sind und das Ergebnis in $U_2 = \mathbb{N}$ abgelegt wird. Eine weitere Umwelt $U = \mathbb{N}$ wird als Zwischenspeicher benutzt. Insgesamt arbeitet demnach der Aktor in $\mathbf{U} = U_1 \times U \times U_2 = \mathbb{N}^4$. Bevor die schrittweise Ermittlung des Produktes beginnt, werden U und U_2 mit Null belegt. Der Automat beendet seine Arbeit sofort bzw. fast sofort, wenn einer der beiden Teile von U_1 leer ist.

```
actor mult
begin
  if look_of (_, 1, _, _) then
    while look_of (_, _, 1, _) do action (0, 0, −1, 0) end while;
    while look_of (_, _, _, 1) do action (0, 0, 0, −1) end while;
    while look_of (1, _, _, _) do
      action (−1, 0, 0, 0);
      repeat action (0, −1, 1, 1) until look_of (_, 0, _, _);
```

```
            repeat action (0, 1, −1, 0) until look_of (_, _, 0, _);
          end while
        end if
      end
```

3.2 Überdeckung von Umwelten

Eine Absichtserklärung

Im Zusammenhang mit dem Begriff der (U, O)-Berechenbarkeit hat sich die Frage nach dem Einfluss der Umwelt U gestellt. Wir haben behauptet, dass es von der "Mächtigkeit" von U abhängt, welche Funktionen berechenbar sind und welche nicht. Jetzt werden wir uns darum bemühen zu beurteilen, wann eine Umwelt mächtiger ist als eine andere. Dazu werden wir zwischen punktierten Umwelten eine Ordnungsrelation $\prec$ definieren, welche die folgenden drei Eigenschaften haben wird:

- $\prec$ ist **reflexiv**, d.h. $(U, O) \prec (U, O)$ für jede Umwelt (U, O)
- $\prec$ ist **transitiv**, d.h. wenn (U, O) von (U', O') überdeckt wird und (U', O') von (U'', O''), so überdeckt (U'', O'') auch (U, O):

  $$(U, O) \prec (U', O') \text{ und } (U', O') \prec (U'', O''), \text{ so } (U, O) \prec (U'', O'')$$

- Wenn $(U, O) \prec (U', O')$ und $f : U_1 \xrightarrow{\supseteq} U_2$ ist eine $(U, 0)$-berechenbare Funktion, so ist f auch $(U', 0')$-berechenbar.

Die dritte Eigenschaft kann man auch so formulieren: Mit einer überdeckenden Umwelt kann man mindestens so viel erreichen wie mit der überdeckten Umwelt. Überdeckung bedeutet im allgemeinen stärkere Rechenkraft.

Für die folgende Definition zur Erinnerung: $P\mathfrak{A}$ ist der Endpunkt, in dem der Prozessor $\mathfrak{A}$ nach Start in P angelangt ist, wenn ein terminaler Zustand erreicht wurde. $\mathfrak{A}(P)$ ist die Endmarke, die dann ausgegeben wird.

die Überdeckungsdefinition

Definition 3.2.1 *Seien $(U, O) = (X, Y, Z, d, l)$ und $(U', O') = (X', Y', Z', d', l')$ punktierte R- bzw. R'-Umwelten. Eine Relation $\prec \subseteq Z \times Z'$ heißt **Überdeckung von (U, O) durch (U', O')** , wenn gilt:*

1. *$(O, O') \in \prec$ bzw. $O \prec O'$*
2. *Es gibt einen R'-Prozessor*

$$Y' \longrightarrow \boxed{\mathfrak{A}} \longrightarrow X'$$
$$\downarrow$$
$$Y$$

mit der Eigenschaft, dass für alle $(P, P') \in Z \times Z'$ gilt: wenn $P \prec P'$, so existiert $P'\mathfrak{A}$ und $P \prec P'\mathfrak{A}$ sowie $\mathfrak{A}(P) = l(p)$.

3. *Für jede Aktion $x \in X$ gibt es einen R'-Aktor*

$$Y' \longrightarrow \boxed{\mathfrak{A}_x} \longrightarrow X'$$

mit der Eigenschaft: wenn $P \prec P'$, $Px = d(P,x)$ definiert ist und Px von einem Punkt aus Z' überdeckt wird, so existiert auch $P'\mathfrak{A}_x$, und es gilt $Px \prec P'\mathfrak{A}_x$.

Sind diese Bedingungen erfüllt, so schreiben wir

$$(U,O) \prec (U',O')$$

und sagen,

> *Die Automatenfamilie $\mathcal{A} = (\{\mathfrak{A}_x \mid x \in X\}, \mathfrak{A})$ realisiert die Überdeckung $(U,O) \prec (U',O')$*

EIN BILD HILFT IMMER

Es ist mnemotechnisch hilfreich und daher legitim, wenn man sich von dem Sachverhalt, der durch die Definition beschrieben wird, eine optische Vorstellung verschafft. Das soll die Abbildung 3.2 leisten. Dort wird die Überdeckung von Punkten durch senkrechte Hilfslinien markiert. Es gilt zum Beispiel $P \prec P'$ und $P \prec P''$. Die hervor gehobenen und mit $\mathfrak{A}$ bzw. $\mathfrak{A}_x$ markierten Kurven in der überdeckenden Umwelt (U',O') bezeichnen Punkte, die von den beiden Prozessoren während ihrer Arbeit passiert werden.

Start- und Endpunkt des "Einblicksautomaten" $\mathfrak{A}$ überdecken denselben Punkt der unterliegenden Umwelt U, können aber durchaus verschieden sein. Die Markenmenge dieses Prozessors ist die Menge Y der Einblicke von U.

Der zur Aktion x der Umwelt (U,O) gehörende Automat $\mathfrak{A}_x$ in (U',O') simuliert die durch x verursachten Bewegungen: dem Übergang $P \xrightarrow{x} Q$ in (U,O) entspricht in (U',O') ein Weg, in dessen Verlauf der P überdeckende Startpunkt P' in $Q'' = P'\mathfrak{A}_x$ überführt wird, und Q'' überdeckt jenen Punkt in der unterliegenden Umwelt, der aus P vermittels der Aktion x entsteht.

EINE MOTIVATION

Was ist erforderlich, um die Arbeit einer Maschine, also eines Paares (Automat,Umwelt) zu gewährleisten? Das sind die Einblicke, die von der Einblicksfunktion l geliefert und von den Automatenfunktionen δ und λ verarbeitet werden, und das sind die Aktionen, die der Automat als Funktionswerte von λ liefert und der Wirkungsfunktion d in der Umwelt zur Verfügung stellt. Und genau diese Werte kann man in der überdeckenden Umwelt zwar nicht direkt, aber immerhin durch die Arbeit von Automaten erhalten. Es ist also zu erwarten, dass es zu jedem Automaten in der Basisumwelt U einen gleichwertigen Automaten in der überdeckenden Umwelt U' gibt. Das wird im nächsten Abschnitt bewiesen.

Diese Simulation einer Umwelt in einer anderen, reicher ausgestatteten Umwelt kann man gut vergleichen mit der Emulation von Zentralprozessoren. Wenn in einer neuen Baureihe die Befehle der alten Modelle verwendbar sein sollen,

muss dafür gesorgt werden, dass die neue CPU sie versteht. Das wird durch eine

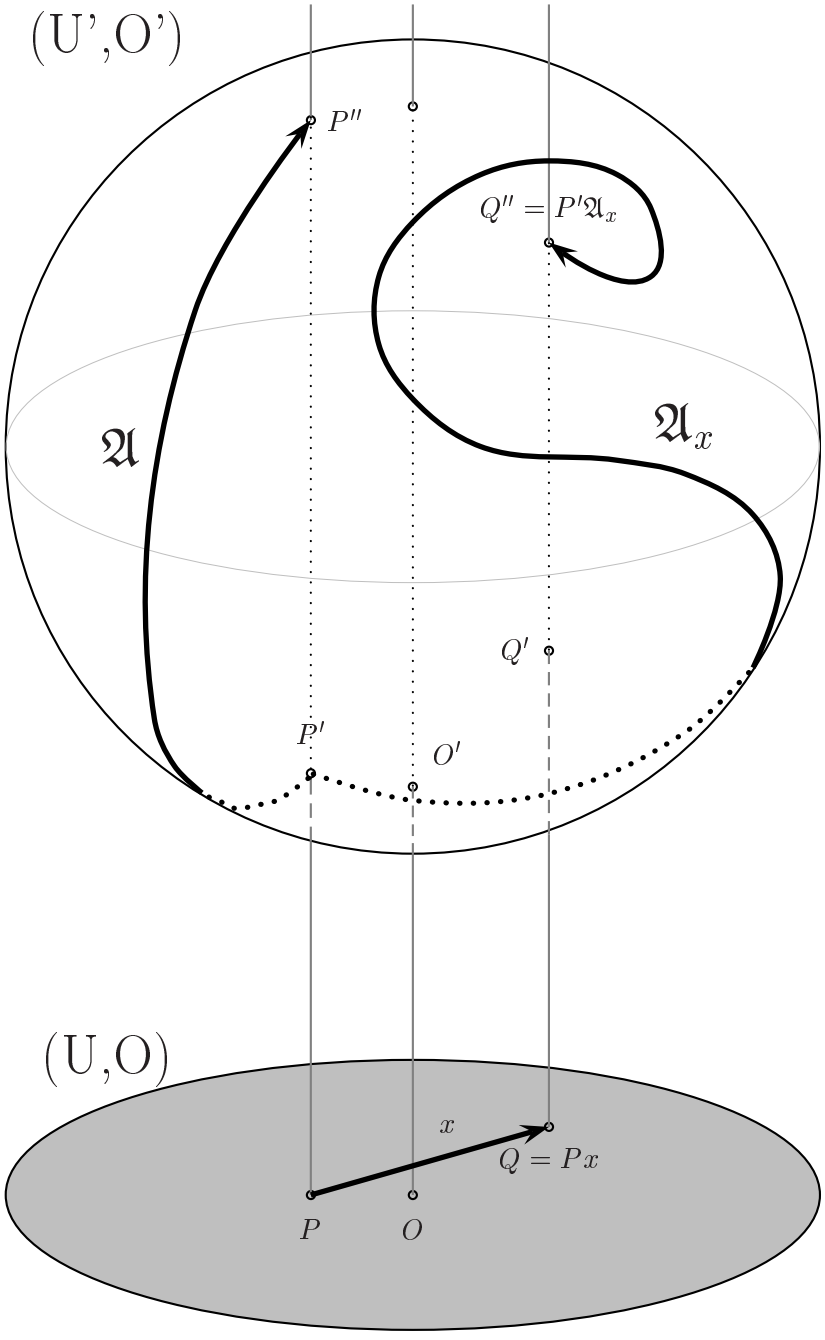


Abbildung 3.2: (U, O) wird von (U', O') überdeckt

Reihe kleiner Programm erreicht, also durch Automaten.

EIN ERSTES EINFACHES BEISPIEL

Zum Abschluss dieses Abschnittes ein einfaches Beispiel einer Überdeckung, bei dem jedoch die typischen Beweisschritte für die Verifikation der Definition sichtbar werden. Wir behaupten, dass sich die Umwelt der natürlichen Zahlen durch die Umwelt der ganzen Zahlen überdecken lässt:

$$(\mathbb{N}, 0) \prec (\mathbb{Z}, 0)$$

Die Idee der Überdeckung und auch der Konstruktion der erforderlichen Automaten ergibt sich aus der Figur 3.3. Die Darstellung der beiden Mengen auf den Zahlenstrahlen lässt vermuten, dass es sinnvoll ist, jeder natürlichen Zahl diejenigen ganzen Zahlen zuzuordnen, die denselben Betrag haben. Dann kann man jedenfalls sofort erkennen, dass die ausgezeichneten Punkte beider Umwelten zur Überdeckungsrelation gehören und dass einer ganzen Zahl mit positivem Betrag immer eine positive natürliche Zahl entspricht. Der Einblicksautomat wird also leicht zu konstruieren sein.

In der Abbildung wird insbesondere dargestellt, was der Automat $\mathfrak{A}_x$, der zur Aktion $x = 1$ in der Umwelt der natürlichen Zahlen gehört, in den beiden Punkten zu tun hat, die eine positive natürliche Zahl überdecken.

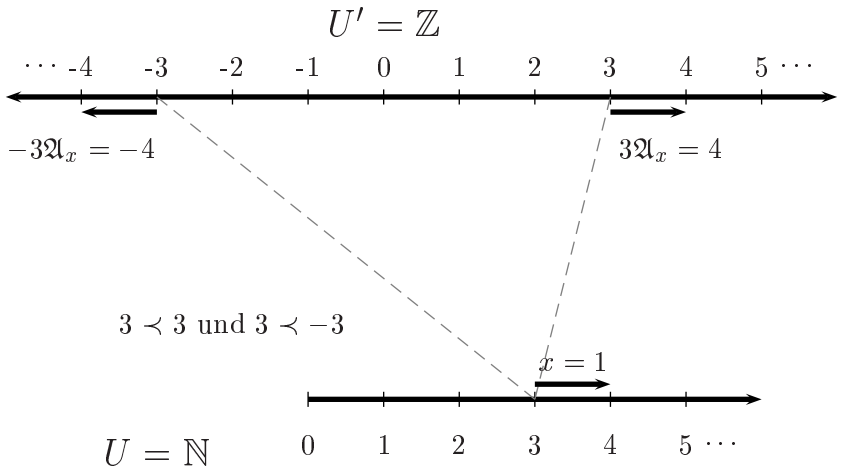


Abbildung 3.3: Die Überdeckung der natürlichen Zahlen durch die ganzen

NACH DER IDEE DEN BEWEIS

In formaler Darstellung sieht das Ganze folgendermaßen aus:

- Definition der Überdeckungsrelation:

$$n \prec z \text{ genau dann, wenn } n = |z| \text{ für alle } n \in \mathbb{N},\ z \in \mathbb{Z}$$

- Verifikation der Überdeckungsdefinition:

  1. $0 \prec 0$ ist offenbar erfüllt.
  2. Der Einblicksprozessor:

```
automaton 𝔄 returns_label {0, 1}
declare view : {0, 1} end declare
begin
  if look_of 0 then
    view := 0 else view := 1
  end if;
  return_label view
end
```

  3. Die beiden Aktoren für die Simulation der Aktionen von $(\mathbb{N}, 0)$ in $(\mathbb{Z}, 0)$:

```
actor 𝔄_1
begin
 if look_of 1 then action(1) else action(-1) end if
end
```

     Bei dem Aktor für die Simulation der Aktion $x = -1$ muss bedacht werden, dass sie im Punkte 0 nicht ausführbar ist.

```
actor 𝔄_(-1)
begin
 if not look_of 0 then
  if look_of 1 then action(-1) else action 1 end if
 end if
end
```

Dies ist ein einfaches Beispiel gewesen, demonstriert aber beispielhaft die erforderliche Vorgehensweise. In der Folge werden noch mehr Beispiele für die Überdeckung von Umwelten vorgestellt. Stets wird sich die Art und Weise der Überdeckung aus einer einfachen geometrischen oder arithmetischen Idee ergeben. Die formale Darstellung ist dann der i.a. schwieriger auszuführende technische Teil des Problems.

3.3 Wichtige allgemeine Überdeckungssätze

Der Begriff der Überdeckung von Umwelten ist sehr inhaltsreich. Deshalb lässt sich sofort eine Reihe wichtiger allgemeiner Eigenschaften beweisen.

Zunächst gehen wir der im vorigen Abschnitt ausgesprochenen Vermutung nach, dass es zu jedem Automaten in einer Umwelt (U, O) einen gleichwertigen Automaten in einer Umwelt (U', O') gibt, sofern (U, O) von (U', O') überdeckt wird.

Satz 3.3.1 *Seien (U, O) und (U', O') R- bzw. R'-Umwelten mit*

$$(U, O) \prec (U', O').$$

Es sei weiterhin $\mathfrak{B}$ ein R-Prozessor. Dann existiert ein R'-Prozessor $\mathfrak{B}'$, der folgendes leistet:

Für alle $P \in U$ und $P' \in U'$ mit $P \prec P'$ gilt

$$P\mathfrak{B} \prec P'\mathfrak{B}' \text{ und } \mathfrak{B}(P) = \mathfrak{B}'(P')$$

Bemerkung: Wir werden sagen, dass der Automat $\mathfrak{B}$ "**geliftet**" wurde, um $\mathfrak{B}'$ zu erzeugen und werden $\mathfrak{B}'$ die "**Liftung**" von $\mathfrak{B}$ nennen.

Beweis:

Der Beweis wird so geführt, dass man aus dem Zustandsgraphen von $\mathfrak{B}$ und mit den wegen der Überdeckung existierenden R'-Prozessoren $\mathfrak{A}$ und $\mathfrak{A}_x$ ein R'-Schema $(\mathbf{G}, A)$ konstruiert. $\mathfrak{B}'$ wird dann $\Gamma(\mathbf{G}, A)$ sein.

Konstruktion der Liftung eines Prozessors in der überdeckenden Umwelt

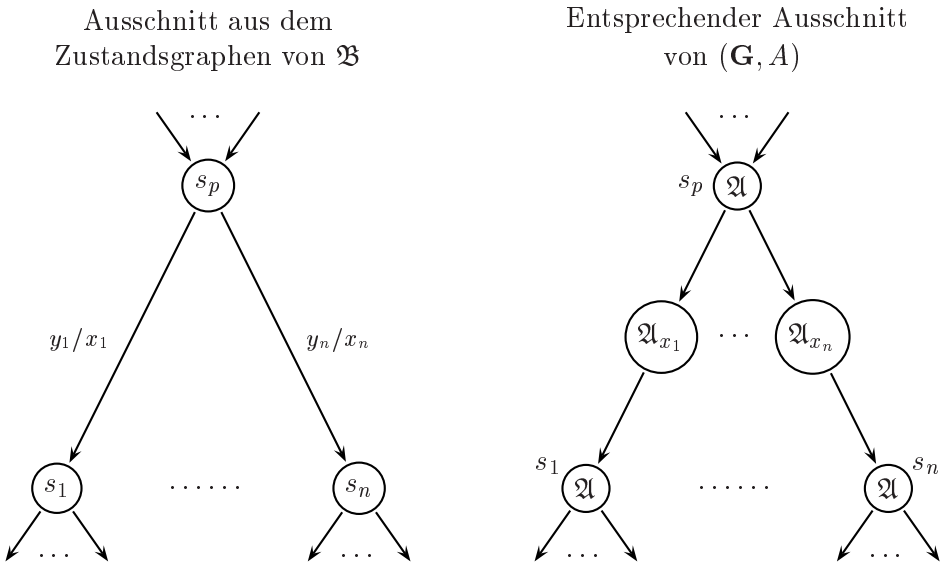


Abbildung 3.4: Vom Zustandsgraphen zum R'-Schema

Die Konstruktion des R'-Schemas ist zwangsläufig (Abbildung 3.4): wenn der Prozessor $\mathfrak{B}$ aus einem Zustand s aufgrund des Einblicks y in einen nächsten Zustand wechselt und dabei die Aktion x ausgibt, so müssen in $\mathfrak{B}'$ der Automat $\mathfrak{A}$ und anschließend der Aktor $\mathfrak{A}_x$ diese Vorgänge simulieren.

Der gerichtete Graph $\mathbf{G} = (V, E, o, t, v_0)$ entsteht folglich aus dem Zustandsgraphen des Prozessors $\mathfrak{B} = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ wie folgt: Jeder Zustand $s \in S$ wird ein Knoten in $\mathbf{G}$, und auf jeder Kante des Graphen von $\mathfrak{B}$ wird ein weiterer Knoten von $\mathbf{G}$ erzeugt. Da diese Knoten auf Kanten von einem aktiven Zustand s zu einem Zustand $s' = \delta(s, y)$ liegen, sollen sie durch Paare $(s, \delta(s, y))$ bezeichnet werden. Damit ist die Knotenmenge des Graphen

$$V = S \cup \{(s, \delta(s, y)) \mid s \in act\mathfrak{B}, y \in Y\}.$$

Insbesondere werden die terminalen Zustände von $\mathfrak{B}$ zu terminalen Knoten des Graphen und der Startzustand s_0 von $\mathfrak{B}$ zum Startknoten von $\mathbf{G}$:

$$term\mathbf{G} := term\mathfrak{B};\ v_0 := s_0.$$

Die Funktion A, die den Knoten von $\mathbf{G}$ Prozessoren bzw. Marken zuweist, wird wie folgt definiert:

- für Knoten $s \in act\mathfrak{B}$ ist $A(s) := \mathfrak{A}$
- für Knoten $s = (s, \delta(s, y))$ ist $A(s) := \mathfrak{A}_{\lambda(s,y)}$
- für Knoten $s \in term\mathfrak{B}$ ist $A(s) := \mu(s)$

Ein Vergleich mit der Definition 2.3.4 auf Seite 66 zeigt, dass damit ein R'-Schema erklärt worden ist, zu dem der R'-Prozessor $\Gamma(\mathbf{G}, A)$ gehört.

Wir werden nun zeigen, dass $\mathfrak{B}' = \Gamma(\mathbf{G}, A)$ ist. Zu diesem Zweck verfolgen wir die Rechnungen

$$(s_0, P_0 = P), (s_1, P_1), \ldots, (s_n, P_n = P\mathfrak{B}) \text{ von } \mathfrak{B} \text{ in } U \text{ und}$$

$$(s'_0, Q_0 = P'), (s'_1, Q_1), \ldots, (s'_m, Q_m = P'\mathfrak{B}') \text{ von } \mathfrak{B}' \text{ in } U'$$

Dabei ist s'_0 der initiale Zustand des in $v_0 = s_0$ platzierten Prozessors, also der initiale Zustand von $\mathfrak{A}$, P ein beliebiger Punkt von U und $P' \in U'$ mit $P \prec P'$.

Zunächst wird das folgende Zwischenresultat gewonnen: Zu der Folge (s'_j, Q_j) gibt es eine Teilfolge $(s'_{i(t)}, Q_{i(t)})$; $0 \leq t \leq n$ mit $P_t \prec Q_{i(t)}$, wobei der Übergang von $Q_{i(t)}$ nach $Q_{i(t+1)}$ durch $\mathfrak{A}$ und anschließend durch $\mathfrak{A}_x$ vollzogen wird:

$$Q_{i(t+1)} = Q_{i(t)}(\mathfrak{A}; \mathfrak{A}_x).$$

Diese Behauptung beweist man durch vollständige Induktion.

Der Induktionsanfang wird durch die Startpunkte geliefert:

$$P = P_0 \prec Q_0 = P'$$

Gilt für einen Index t mit $t < n$ die Überdeckung $P_t \prec Q_{i(t)}$, so folgt einerseits mit $\mathfrak{B}$ in U

$$P_{t+1} = P_t x \text{ mit } x = \lambda(s_t, l(P_t)).$$

Andererseits wird der komplexe Prozessor $\Gamma(\mathbf{G}, A)$ zunächst zu

$$\bar{Q} := Q_{i(t)}\mathfrak{A} \text{ mit } P_t \prec \bar{Q}$$

gelangen und im nächsten Schritt, da ja nach Definition der Überdeckung $\mathfrak{A}_x$ existiert und korrekt arbeitet,

$$Q_{i(t)}\mathfrak{A}_x \text{ mit } P_{t+1} = P_t x \prec \bar{Q}\mathfrak{A}_x = Q_{i(t+1)}$$

erreichen.

Damit ist die Zwischenbehauptung bewiesen.

Die Aussage des Satzes folgt nun aus der Definition von $\Gamma(\mathbf{G}, A)$. Wenn die Rechnung von $\mathfrak{B}$ in (s_n, P_n) endet, also s_n terminaler Zustand von $\mathfrak{B}$ ist, so ist s_n auch terminaler Knoten von $\mathbf{G}$ und damit Endzustand von $\Gamma(\mathbf{G}, A)$, der zudem genau dann von $\Gamma(\mathbf{G}, A)$ erreicht wird, wenn $\mathfrak{B}$ in s_n endet. Also ist gemäß der Definition der Funktion A

$$\mu(s_n) = \mathfrak{B}(P_0) = A(s_n) = \Gamma(\mathbf{G}, A)(Q_0)$$

und nach der eben bewiesenen Hilfsaussage auch

$$P_0\mathfrak{B} = P_n \prec Q_0(\Gamma(\mathbf{G}, A)) = Q_m.$$

Das war zu beweisen.

Die Überdeckung ist eine Relation in einer Menge von Umwelten, von der wir jetzt die Transitivität beweisen werden.

Satz 3.3.2 *Sind* (U_i, O_i); $(i = 1, 2, 3)$ *punktierte* R_i*-Umwelten und gilt*

$$(U_1, O_1) \overset{(1)}{\prec} (U_2, O_2) \text{ sowie } (U_2, O_2) \overset{(2)}{\prec} (U_3, O_3),$$

Überdeckung ist transitiv

so folgt

$$(U_1, O_1) \prec (U_3, O_3)$$

Beweis: Entscheidendes Hilfsmittel im folgenden Beweis ist die Aussage des vorigen Satzes. Zu den wegen der Überdeckung $(U_1, O_1) \overset{(1)}{\prec} (U_2, O_2)$ existierenden R_1-Automaten $\mathfrak{A}$ und $\mathfrak{A}_{x_1}$ werden die entsprechenden gelifteten R_3-Prozessoren benutzt, um die Überdeckung $(U_1, O_1) \prec (U_3, O_3)$ zu konstruieren.

Seien $U_i = (X_i, Y_i, Z_i, d_i, l_i, O_i);\ (i = 1, 2, 3)$ die drei Umwelten. Die zur Verfügung stehenden Voraussetzungen sind in den folgenden Formeln zusammengefasst:

$$(U_1, O_1) \overset{(1)}{\prec} (U_2, O_2) : \begin{cases} 1. & O_1 \overset{(1)}{\prec} O_2 \\ 2. & P_1 \overset{(1)}{\prec} P_2 \Rightarrow P_1 \overset{(1)}{\prec} P_2\mathfrak{A} \text{ und } \mathfrak{A}(P_2) = l_1(P_1) \\ 3. & P_1 \overset{(1)}{\prec} P_2 \Rightarrow P_1 x_1 \overset{(1)}{\prec} P_2\mathfrak{A}_{x_1};\ (x_1 \in X_1) \end{cases} \tag{3.1}$$

$$(U_2, O_2) \overset{(2)}{\prec} (U_3, O_3) : \begin{cases} 1. & O_2 \overset{(2)}{\prec} O_3 \\ 2. & P_2 \overset{(2)}{\prec} P_3 \Rightarrow P_2 \overset{(2)}{\prec} P_3\mathfrak{B} \text{ und } \mathfrak{B}(P_3) = l_2(P_2) \\ 3. & P_2 \overset{(2)}{\prec} P_3 \Rightarrow P_2 x_2 \overset{(2)}{\prec} P_3\mathfrak{B}_{x_2};\ (x_2 \in X_2) \end{cases} \tag{3.2}$$

Wir definieren:

$$\prec := \overset{(1)}{\prec} \circ \overset{(2)}{\prec}$$

$$= \{(P_1, P_3) \in Z_1 \times Z_3 \mid \text{ es gibt ein } P_2 \in Z_2 \text{ mit } P_1 \overset{(1)}{\prec} P_2 \text{ und } P_2 \overset{(1)}{\prec} P_3\}$$

Es ist zu zeigen, dass $\prec$ den drei Eigenschaften der Überdeckungsrelation genügt.

1. Aus der jeweils ersten Zeile von 3.1 und 3.2 folgt sofort $O_1 \prec O_3$
2. Seien $P_1 \in Z_1,\ P_3 \in Z_3$ und $P_1 \prec P_3$. Es ist ein R_3-Prozessor $\mathfrak{C}$ anzugeben, der folgendes leistet:

   $$P_1 \prec P_3\mathfrak{C} \text{ und } \mathfrak{C}(P_3) = l_1(P_1).$$

   Wegen $P_1 \prec P_3$ gibt es einen Punkt $P_2 \in Z_2$ mit $P_1 \overset{(1)}{\prec} P_2 \overset{(2)}{\prec} P_3$ und wegen $(U_1, O_1) \overset{(1)}{\prec} (U_2, O_2)$ einen R_2-Automaten $\mathfrak{A}$ mit den in Punkt 2 von 3.1 angegebenen Eigenschaften. Dieser Prozessor wird gemäß Satz 3.3.1 zu einem R_3-Prozessor $\mathfrak{A}'$ geliftet. Es ist $\mathfrak{C} = \mathfrak{A}'$, denn

   $$P_1 \overset{(1)}{\prec} P_2\mathfrak{A} \overset{(2)}{\prec} P_3\mathfrak{A}', \text{ das heißt } P_1 \prec P_3\mathfrak{A}' \text{ und } l_1(P_1) = \mathfrak{A}(P_2) = \mathfrak{A}'(P_3)$$

MIT DEN GELIFTETEN AUTOMATEN GEHT ES

3. Wir gehen wieder aus von zwei Punkten $P_1 \prec P_3$ und einer Aktion $x \in X_1$. Es ist ein R_3-Aktor $\mathfrak{C}_x$ anzugeben mit $P_1 x \prec P_3 \mathfrak{C}_x$.

   Zunächst gibt es wieder einen Punkt $P_2 \in U_2$ mit $P_1 \overset{(1)}{\prec} P_2 \overset{(2)}{\prec} P_3$. Jetzt benutzen wir Punkt 3 aus 3.1 : es gibt einen R_2-Automaten $\mathfrak{A}_x$ derart, dass $P_1 x \overset{(1)}{\prec} P_2 \mathfrak{A}_x$ gilt. Wegen Satz 3.3.1 leistet das Lifting $\mathfrak{A}'_x$ von $\mathfrak{A}_x$ das Gewünschte:

$$P_1 x \overset{(1)}{\prec} P_2 \mathfrak{A}_x \overset{(2)}{\prec} P_3 \mathfrak{A}'_x, \text{ das bedeutet } P_1 \prec P_3 \mathfrak{A}'_x$$

Damit sind die drei Überdeckungseigenschaften verifiziert. Die Behauptung ist bewiesen: die Überdeckung von Umwelten ist eine transitive Relation.

Sie ist auch reflexiv: jede Umwelt überdeckt sich selbst. Das ist leicht zu zeigen.

Es wäre schön, wenn sie auch noch antisymmetrisch wäre, das heißt, wenn aus $(U_1, O_1) \prec (U_2, O_2)$ und $(U_2, O_2) \prec (U_1, O_1)$ die Gleichheit $(U_1, O_1) = (U_2, O_2)$ folgen würde. Das stimmt aber nicht, wie das folgende Beispiel zeigt:

Mit der Menge $\mathbb{Z}$ der ganzen rationalen Zahlen bilden wir einerseits die Umwelt

$$U_1 = (X, Y, \mathbb{Z}, d, l) : \; X = \{0, 1, -1\}; \; Y = \{\star\}; \; d(z, x) = z + x; \; l(z) = \star.$$

Andererseits betrachten wir die sehr triviale Umwelt

$$U_2 = (\tilde{X}, \tilde{Y}, \tilde{Z}, \tilde{d}, \tilde{l}) : \; \tilde{X} = \tilde{Y} = \tilde{Z} = \{\star\}; \; \tilde{d}(\star, \star) = \star; \; \tilde{l}(\star) = \star.$$

Von diesen beiden Umwelten lässt sich leicht zeigen, dass

$$(U_1, 0) \prec (U_2, \star) \text{ und } (U_2, \star) \prec (U_1, 0)$$

gilt.

In einer Menge von Umwelten kann man in folgender Weise eine Äquivalenzrelation stiften:

Definition 3.3.1 *Zwei punktierte Umwelten (U_1, O_1) und (U_2, O_2) heißen äquivalent, wenn $(U_1, O_1) \prec (U_2, O_2)$ und $(U_2, O_2) \prec (U_1, O_1)$ gilt. In diesem Falle schreiben wir $(U_1, O_1) \equiv (U_2, O_2)$*

Es ist leicht zu zeigen, dass auf diese Weise tatsächlich eine Äquivalenzrelation definiert ist.

Nachdem wir etwas wissen über die algebraischen Eigenschaften der Überdeckung, wollen wir uns nun deren Bedeutung für die Rechenkraft von Umwelten zuwenden. In einem ersten Schritt wollen wir beweisen, dass die Überdeckungsrelation sich fortpflanzt, wenn man das kartesische Produkt von Umwelten bildet. Dazu müssen wir wissen, wie man aus zwei Prozessoren über den Teilumwelten in sinnvoller Weise einen Automaten über der Produktumwelt erzeugt. Das werden wir so einrichten, dass beide Teilprozessoren in den Teilumwelten ihre Arbeit nacheinander verrichten. Wenn der aus $\mathfrak{A}_1$ und $\mathfrak{A}_2$ gebildete Prozessor $\mathfrak{A} := \mathfrak{A}_1 \otimes \mathfrak{A}_2$ in einem Punkte $\mathbf{P} = (P_1, P_2)$ der Produktumwelt gestartet wird, so soll zunächst nur $\mathfrak{A}_1$ tätig werden, bis ein Endzustand erreicht ist. In dieser ersten Phase wird nur P_1 verändert. Das wird dadurch erreicht, dass die Ausgabefunktion λ von $\mathfrak{A}_1 \otimes \mathfrak{A}_2$ Aktionen der Gestalt $(x_1, stop)$ sendet. Wenn $\mathfrak{A}_1$ in einem terminalen Zustand angelangt ist, soll der andere Automat seine Arbeit beginnen und P_2 bewegen. Deshalb werden die terminalen Zustände von $\mathfrak{A}_1$ jeweils mit dem Startzustand von $\mathfrak{A}_2$ identifiziert. Wenn $\mathfrak{A}_1$ mehr als einen terminalen Zustand hat, müssen die Zustände von $\mathfrak{A}_2$ mehrfach verwendet werden. Um Verwechslungen zu vermeiden, vor allem aber um die Endmarken von $\mathfrak{A}_1$ nicht zu verlieren, wenn $\mathfrak{A}_1$ arbeitet, werden diese zur Unterscheidung benutzt.

Definition 3.3.2 *Seien $\mathfrak{A}_i = (S_i, s_{0i}, X_i, Y_i, M_i, \delta_i, \lambda_i, \mu_i)$ R_i-Prozessoren über den R_i-Umwelten $U_i = (X_i, Y_i, Z_i, d_i, l_i)$; $(i = 1, 2)$.*

AUS ZWEI MASCHINEN WIRD EINE

$$U = U_1 \times U_2 = ((X_1 \times \{stop\}) \cup (\{stop\} \times X_2), Y_1 \times Y_2, d, l)$$

ist die eingeschränkte Produktumwelt. (vergl. die Definition 2.2.6 auf Seite 38) $\mathfrak{A} = \mathfrak{A}_1 \otimes \mathfrak{A}_2 = (S, s_0, X, Y, M, \delta, \lambda, \mu)$ ist als Prozessor über U wie folgt erklärt:

- $S := act\mathfrak{A}_1 \cup (M_1 \times S_2)$; $s_0 := s_{01}$;
- *X und Y wie in U;*
- $M := (M_1 \times M_2) \cup \{go\}$;
- $term\mathfrak{A} := M_1 \times term\mathfrak{A}_2$; $act\mathfrak{A} = act\mathfrak{A}_1 \cup (M_1 \times act\mathfrak{A}_2)$;
- $$\delta(\sigma, (y_1, y_2)) := \begin{cases} \delta_1(s_1, y_1) & \text{für } \sigma = s_1 \in act\mathfrak{A}_1 \text{ und } \delta_1(s_1, y_1) \notin term\mathfrak{A}_1 \\ (\mu(\delta_1(s_1, y_1)), s_{02}) & \text{für } \sigma = s_1 \in act\mathfrak{A}_1 \text{ und } \delta_1(s_1, y_1) \in term\mathfrak{A}_1 \\ (m, \delta_2(s_2, y_2)) & \text{für } \sigma = (m, s_2) \in M_1 \times act\mathfrak{A}_2 \end{cases}$$
- $$\lambda(\sigma, (y_1, y_2)) := \begin{cases} (\lambda_1(s_1, y_1), stop) & \text{für } \sigma = s_1 \in act\mathfrak{A}_1 \\ (stop, \lambda_2(s_2, y_2)) & \text{für } \sigma = (m, s_2) \in M_1 \times act\mathfrak{A}_2 \end{cases}$$

- $$\mu(\sigma) := \begin{cases} (m_1, \mu_2(s_2)) & \text{für } \sigma = (m_1, s_2) \in M_1 \times term\mathfrak{A}_2 \\ go\ sonst \end{cases}$$

Erwartungsgemäß ergibt sich die

Folgerung 3.3.1

$$\boldsymbol{P}\mathfrak{A} = (P_1, P_2)\mathfrak{A} = (P_1\mathfrak{A}_1, P_2\mathfrak{A}_2)$$
$$\mathfrak{A}(\boldsymbol{P}) = \mathfrak{A}(P_1, P_2) = (\mathfrak{A}_1(P_1), \mathfrak{A}_2(P_2))$$

Zum Beweis der Folgerung reicht es aus, die Rechnungen der drei Automaten $\mathfrak{A}_1$, $\mathfrak{A}_2$ und $\mathfrak{A}$ aufzuschreiben:

$$\mathfrak{A}_1 : \begin{cases} P_0 \xrightarrow{l_1(P_0)/x_{01}} P_1 = P_0 x_{01} \xrightarrow{l_1(P_1)/x_{11}} \dots \xrightarrow{l_1(P_{m-1})/x_{(m-1)1}} P_m = P_{m-1} x_{(m-1)1}; \\ s_{01} \xrightarrow{y_{01}=l_1(P_0)} s_{11} \xrightarrow{y_{11}=l_1(P_1)} \dots \xrightarrow{y_{(m-1)1}=l_1(P_{m-1})} s_{m1} = t_1 \in term\mathfrak{A}_1; \\ x_{i1} = \lambda_1(s_{i1}, y_{i1}); \;\; \mu_1(t_1) = m_1 \end{cases}$$

$$\mathfrak{A}_1 : \begin{cases} Q_0 \xrightarrow{l_2(Q_0)/x_{02}} Q_1 = Q_0 x_{02} \xrightarrow{l_2(Q_1)/x_{12}} \dots \xrightarrow{l_2(Q_{n-1})/x_{(n-1)2}} Q_n = Q_{n-1} x_{(n-1)2}; \\ s_{02} \xrightarrow{y_{02}=l_2(Q_0)} s_{12} \xrightarrow{y_{12}=l_2(Q_1)} \dots \xrightarrow{y_{(n-1)2}=l_2(Q_{n-1})} s_{n2} = t_2 \in term\mathfrak{A}_2; \\ x_{i1} = \lambda_1(s_{i1}, y_{i1}); \;\; \mu_1(t_1) = m_1 \end{cases}$$

Daraus wird mit dem neuen Prozessor

$$\mathfrak{A} : \begin{cases} \mathbf{P}_0 = (P_0, Q_0) \xrightarrow{(y_{01}, y_{02})/(x_1, stop)} (P_1, Q_0) \xrightarrow{(y_{11}, y_{02})/(x_{11}, stop)} \dots \\ \qquad \dots (P_{m-1}, Q_0) \xrightarrow{(y_{(m-1)1}, y_{02})/(x_{(m-1)1}, stop)} (P_m, Q_0) \xrightarrow{(y_m, y_{02})/(stop, x_{02})} \dots \\ \qquad \dots \; (P_m, Q_(n-1)) \xrightarrow{(y_{m1}, y_{(n-1)2})/(stop, x_{(n-1)2})} (P_m, Q_n); \\ \\ s_{01} \xrightarrow{(y_{01}, y_{02})} s_{11} \xrightarrow{(y_{11}, y_{02})} \dots \; s_{(m-1)1} \xrightarrow{(y_{(m-1)1}, y_{02})} \\ \qquad (m_1, s_{02}) \xrightarrow{(y_{m1}, y_{02})} \xrightarrow{(y_{m1}, y_{02})} (m_1, s_{12}) \; \dots \; (m_1, s_{(n-1)2}) \xrightarrow{(y_{m1}, y_{(n-1)2})} (m_1, t_2); \\ \\ \mu(m_1, t_2) = (m_1, \mu_2(t_2)) = (m_1, m_2). \end{cases}$$

Diese Konstruktion kann man selbstverständlich auf mehr als zwei Prozessoren ausdehnen. Dabei nimmt lediglich die Schreibarbeit zu.

Jetzt kann der angekündigte Satz bewiesen werden.

DAS KARTESISCHE PRODUKT BEWAHRT DIE ÜBER-DECKUNGS-EIGENSCHAFT

Satz 3.3.3 *Aus*

$$(U_1, O_1) \overset{(1)}{\prec} (U_1', O_1') \text{ und } (U_2, O_2) \overset{(2)}{\prec} (U_2', O_2')$$

folgt

$$(U_1 \times U_2, (O_1, O_2)) \prec (U_1' \times U_2', (O_1', O_2'))$$

Beweis: Für alle $\mathbf{P} = (P_1, P_2) \in U_1 \times U_2$ und $\mathbf{Q} = (Q_1, Q_2) \in U_1' \times U_2'$ wird festgelegt:

$$\mathbf{P} = (P_1, P_2) \prec \mathbf{Q} = (Q_1, Q_2) \text{ genau dann, wenn} P_1 \overset{(1)}{\prec} Q_1 \text{ und } Q_1 \overset{(2)}{\prec} Q_2$$

1. Das ergibt sofort $(O_1, O_2) \prec (O_1', O_2')$
2. Wenn $\mathbf{P} = (P_1, P_2) \prec \mathbf{Q} = (Q_1, Q_2)$, dann gibt es einen R_1'-Prozessor $\mathfrak{A}_1$ und einen R_2'-Prozessor $\mathfrak{A}_2$ mit

   $$P_i \overset{(i)}{\prec} Q_i\mathfrak{A}_i \text{ und } l_i(P_i) = \mathfrak{A}_i Q_i;\ (i = 1, 2).$$

   Für den Prozessor $\mathfrak{A} = \mathfrak{A}_1 \otimes \mathfrak{A}_2$ ergeben sich die entsprechenden Eigenschaften:

   $$\mathbf{P} = (P_1, P_2) \prec (P_1\mathfrak{A}_1, P_2\mathfrak{A}_2) = (P_1, P_2)\mathfrak{A} = \mathbf{P}\mathfrak{A}$$

   und

   $$l(\mathbf{P}) = (l_1(P_1), l_2(P_2)) = (\mathfrak{A}_1 P_1, \mathfrak{A}_2 P_2) = \mathfrak{A}P$$

3. Ist $\mathbf{P} = (P_1, P_2) \prec \mathbf{Q} = (Q_1, Q_2)$ und ist $P_i x_i;\ (i = 1, 2)$ definiert, so gibt es R_1- bzw. R_2-Prozessoren $\mathfrak{A}_{x_1}$ und $\mathfrak{A}_{x_2}$ derart, dass $P_i x_i \overset{(i)}{\prec} Q_i\mathfrak{A}_{x_i}$ gilt. Daraus folgt aber

   $$\mathbf{P}\mathbf{x} = (P_1 x_1, P_2, x_2) \prec (Q_1\mathfrak{A}_{x_1}, Q_2\mathfrak{A}_{x_2}) = \mathbf{Q}(\mathfrak{A}_{x_1} \otimes \mathfrak{A}_{x_2})$$

   $\mathfrak{A}_x := \mathfrak{A}_{x_1} \otimes \mathfrak{A}_{x_2}$ ist der Prozessor in $U_1' \times U_2'$, welcher die Aktion $\mathbf{x} = (x_1, x_2)$ der Umwelt $U_1 \times U_2$ simuliert.

Wenn man eine Umwelt (U_1, O_1) mit einer zweiten Umwelt zu einem kartesischen Produkt $(U_1 \times U_2, (O_1, O_2))$ anreichert, erhält man eine Überdeckung.

Satz 3.3.4

$$(U_1, O_1) \prec (U_1 \times U_2, (O_1, O_2)) \text{ und } (U_2, O_2) \prec (U_1 \times U_2, (O_1, O_2))$$

Beweis: Es wird die triviale Umwelt $\star = (X, Y, Z, d, l)$, definiert durch

$$X = Y = Z = \{\star\};\ d(\star, \star) = l(\star) = \star,$$

zu Hilfe genommen. Man überlegt sich leicht, dass für jede Umwelt (U, O) die Überdeckungen $(U, O) \prec (U \times \star, (O, \star))$ und $(\star, \star) \prec (U, O)$ gelten. Mit Hilfe des vorigen Satzes ergibt sich

$$(U_1, O_1) \prec (U_1 \times \star, (O_1, \star)) \prec (U_1 \times U_2, (O_1, O_2)).$$

Wegen der Transitivität von Überdeckungen folgt die Aussage des Satzes bezüglich der Umwelt (U_1, O_1). Die Überlegungen für die andere Komponente des kartesischen Produktes verlaufen analog.

EINE PRODUKTUMWELT ÜBERDECKT JEDE IHRER KOMPONENTENUMWELTEN.

Nun werden wir zeigen können, dass Überdeckung stärkere Rechenkraft impliziert.

Satz 3.3.5 *Wenn (U, O) durch (U', O') überdeckt wird, ist jede (U, O)-berechenbare Funktion auch (U', O')-berechenbar.*

Beweis: Wir haben uns an die Definition 3.1.1 auf Seite 101 zu erinnern: wenn eine Funktion $f : U_1 \xrightarrow{\supset} U_2$ mit (U, O) berechenbar ist, dann gibt es einen $R_1 \times R \times R_2$-Aktor $\mathfrak{A}$ über der Umwelt $(U_1 \times U \times U_2, (O_1, O, O_2))$ mit

$$(P_1, O, O_2)\mathfrak{A} = (Q_1, Q, Q_2) \text{ und } f(P_1) = Q_2$$

für jeden Punkt $P_1 \in dom\ f$.

Da die Überdeckung eine reflexive Relation ist und wegen Satz 3.3.3 gilt

$$(U_1 \times U \times U_2, (O_1, O, O_2)) \prec (U_1 \times U' \times U_2, (O_1, O', O_2))$$

Der Aktor $\mathfrak{A}$ wird gemäß der Konstruktion des Satzes 3.3.1 (Seite 107) zum Aktor $\mathfrak{A}'$ geliftet.

Damit erhält man wegen $(P_1, O, P_2) \prec (P_1, O', P_2)$

$$(Q_1, Q, Q_2) = (P_1, O, O_2)\mathfrak{A} \prec (P_1, O', O_2)\mathfrak{A}' = (\tilde{Q_1}, \tilde{Q}, \tilde{Q_2}).$$

Daraus folgt $Q_1 \prec \tilde{Q_1}$. Da die beiden Punkte zur selben Umwelt U_1 gehören, ist Überdeckung deren Gleichheit: $Q_1 = \tilde{Q_1}$. Ebenso ist $Q_2 = \tilde{Q_2}$.
Ergebnis:

$$(P_1, O', O_2)\mathfrak{A}' = (Q_1, \tilde{Q}, Q_2)$$

Und das bedeutet

$$\xi_{(\mathfrak{A}', (U', O'))}(P_1) = Q_2 = f(P_1).$$

f lässt sich mit Hilfe von $\mathfrak{A}'$ über $(U_1 \times U' \times U_2, (O_1, O', O_2))$ berechnen. Das war zu zeigen.

Die Definition der Berechenbarkeit mit einer Umwelt $(U, O) = (X, Y, Z, d, l)$ benutzt wesentlich die Existenz eines ausgezeichneten Punktes und aller jener Punkte von Z, die im Verlauf eines Rechenprozesses von O aus erreicht werden. Das gibt Anlass, diesen Teil von U gesondert zu betrachten.

Definition 3.3.3 *Sei $(U, O) = (X, Y, Z, d, l)$ eine punktierte Umwelt. Die Menge Z_O der von O aus erreichbaren Punkte wird induktiv in folgender Weise definiert:*

1. *$O \in Z_O$*
2. *Wenn $P \in Z \cap Z_O$ und $Q = Px = d(P, x)$ existiert für eine Aktion $x \in X$, so ist auch $Px \in Z_O$*
3. *Das sind alle Punkte, die zu Z_O gehören.*

Zu d und l seien $d' = d|(Z_O \times X)$ und $l' = l|Z_O$ die Einschränkungen auf U_O. $(U_O, O) = (X, Y, Z_O, d', l')$ ist ebenfalls eine punktierte Umwelt. Wenn (U, O) eine R-Umwelt ist, so auch (U_O, O).

Nun kann man zeigen, dass (U, O) und (U_O, O) gleiche Rechenkraft besitzen.

Satz 3.3.6 *Für alle punktierten Umwelten (U, O) ist*

$$(U, O) \equiv (U_O, O)$$

NUR ERREICHBARE PUNKTE SIND WICHTIG

Beweis: Es werden die binären Relationen

$$\overset{(1)}{\prec} = \{(P, P) \mid P \in Z_O\} \subseteq Z \times Z_O$$

und

$$\overset{(2)}{\prec} = \mathbb{1}_{Z_O} = \{(P, P) \mid P \in Z_O\} \subseteq Z_O \times Z$$

benutzt. Die Überdeckungen

$$(U,O) \overset{(1)}{\prec} (U_O,O) \text{ und } (U_O,O) \overset{(2)}{\prec} (U,O)$$

werden beide durch dieselbe Automatenfamilie

$$(\{\mathbf{action(x)} \mid x \in X\}, \mathbf{look})$$

(vergl. den Abschnitt 2.4.1 auf Seite 70) realisiert.

Damit beenden wir allgemeine Überlegungen zu Überdeckungen und wenden uns konkreten Umwelten zu.

3.4 Umwelten im Vergleich

In diesem Abschnitt wird eine Reihe von Überdeckungssätzen bewiesen, welche die wichtigsten Umwelten bezüglich ihrer Rechenkraft in Beziehung setzen. Der folgende erste dieser Sätze ist nahe liegend und einfach.

Überdeckungssatz 1 $(\mathbb{N},0)^2 \prec (RAM,(0,\omega))$

Beweis:
Zunächst eine organisatorische Bemerkung. In einer RAM-Umwelt gibt es bei den Aktionen einen Bezeichnungskonflikt, weil Halm- und Basisumwelt übereinstimmen. Aus diesem Grunde sollen in der Basisumwelt die Aktionen, die bekanntlich die Bewegungen des Schreib-Lese-Kopfes bewirken, mit re für die Rechts- und li für die Linksbewegung bezeichnet werden.

Die Überdeckung wird so organisiert, dass die in der Umwelt $\mathbb{N}^2$ möglichen Aktivitäten auf die ersten beiden Plätze des RAM-Bandes verlegt werden. Den Paaren $(m,n) \in \mathbb{N}^2$ entsprechen in der RAM-Umwelt Belegungen des Bandes durch Funktionen $f_{(m,n)}$, die wie folgt definiert sind:

Platz für zwei Zahlen auf dem RAM? Kein Problem!

$$f_{(m,n)}(i) = \begin{cases} m & \text{für } i = 0 \\ n & \text{für } i = 1 \\ 0 & \text{sonst} \end{cases}$$

Punkte der RAM-Umwelt enthalten außer der Belegungsvorschrift des Bandes noch die aktuelle Position auf dem Band, die bei unserer Definition der Überdeckung auf Null gesetzt wird:

$$(m,n) \prec (0, f_{(m,n)})$$

Damit ist offenbar die Überdeckung der ausgezeichneten Punkte gesichert:

$$(0,0) \prec (0,\omega)$$

Die Definition der Automatenfamilie zu dieser Überdeckung ist ebenfalls nicht schwierig. Der Einblicksautomat kann wie folgt angegeben werden:

automaton $\mathfrak{A}$ **returns_label** $\{0,1\}^2$
declare $label1, label2 : \{0,1\}$ **end declare**
begin
  look$\rightarrow$ $(\_, label1)$;
  action (re);
  look$\rightarrow$ $(\_, label2)$;
  action (li);
  return_label$(label1, label2)$
end

Bei der Definition der Aktionsprozessoren kann man sich ohne Beschränkung der Allgemeinheit auf solche Aktionen in $(\mathbb{N}^2, (0,0))$ beschränken, wo eine Komponente die Stopp-Aktion ist. Von den vier möglichen Aktoren schreiben wir nur zwei auf.

actor $\mathfrak{A}_{(0,1)}$
begin
  action(re);
  action(1);
  action(li)
end

actor $\mathfrak{A}_{(-1,0)}$
begin
  if look_of $(\_, 1)$
    then action(-1)
  end if
end

Was meinte Minsky?

Im folgenden Satz wird ein erstaunliches Ergebnis formuliert: die Rechenkraft einer beliebigen endlichen Anzahl von Zählerumwelten lässt sich mit lediglich zwei Zählerumwelten realisieren. Dieses Resultat wurde zuerst von Marvin Minsky in einer anderen Fassung ausgesprochen: jedes Programm mit endlich vielen Variablen kann man so abändern, dass man mit zwei Variablen auskommt. Dabei hat er an Variablen gedacht, die mit beliebig vielen Zahlen belegt werden können. Das sind also nicht Programme, welche Prozessoren darstellen. Jede Variable in einem Automatenprogramm repräsentiert ja immer einen Punkt in einer endlichen Umwelt. Minsky hat also irgendwelche Computerprogramme gemeint. Nehmen wir ein Beispiel. Die folgende kleine, in Java geschriebene Methode multipliziert natürliche Zahlen, wobei lediglich die Rechenoperationen zu Hilfe genommen werden, die in der Umwelt der natürlichen Zahlen möglich sind:

```
int mult(int faktor1, int faktor2){
  int x1 = faktor1, x2 = faktor2, zs = 0, resultat = 0;
  if (x1 > 0){
    while (x2 > 0){
      x2 − −;
      while (x1 > 0){
        x1 − −;
        zs + +;
        resultat + +;
      }
      while (zs > 0){
        zs − −;
        x1 + +;
      }
    }
  }
  return resultat;
}
```

Die zu multiplizierenden Faktoren werden den Variablen $x1$ und $x2$ zugewiesen. Der Wert von $x1$ wird $x2$-Mal zur Variablen *resultat* aufaddiert. Dabei wird zs als "Zwischenspeicher" benutzt, damit der ursprüngliche Wert von $x1$ nicht verloren geht. Diesem Computerprogramm entspricht das folgende Programm in unserer Automatensprache:

```
actor mult
begin
 if look of (1, _, _, _)
  while look of (_, 1, _, _) do
   action (0, −1, 0, 0)
   while look of (1, _, _) do
    action (−1, 0, 1, 1)
   end while
   while look of (_, _, 1, _) do
    action (1, 0, −1, 0)
   end while
 end if
end
```

Dieser Aktor arbeitet ebenso wie das Java-Programm in der Umwelt $\mathbb{N}^4$. Wenn er im Punkte $(x1, x2, 0, 0)$ gestartet wird, endet er in $(x1, 0, 0, x1 \cdot x2)$. Der nachfolgende Satz behauptet nun, dass man dasselbe mit der Umwelt $\mathbb{N}^2$ bewerkstelligen kann, was für ein entsprechendes Computerprogramm heißt, dass man statt der vier Variablen $x1, x2, zs$ und *resultat* nur zwei benötigt.

Dies ist ein wichtiges theoretisches Ergebnis, welches allerdings keine praktische Bedeutung hat. Der Beweis zeigt, dass die Einsparung an Speicher mit hohem Rechenaufwand erkauft werden muss.

EINE STARKE BEHAUPTUNG...

Überdeckungssatz 2 *Für alle positiven natürlichen Zahlen k gilt*

$$(\mathbb{N},0)^k \prec (\mathbb{N}^2,(0,1))$$

Beweis: Für $k = 2$ ist die Aussage trivial, und für $k = 1$ haben wir mit Satz 3.3.4 auf Seite 115 bereits ein allgemeineres Ergebnis. Im weiteren wird also $k > 2$ angenommen.

Zur Definition der Überdeckung von Punkten wählt man k paarweise verschiedene Primzahlen p_i $(1 \leq i \leq k)$ und legt fest:

..UND EINE RAFFINIERTE KODIERUNG.

$$(n_1, n_1, \ldots, n_k) \prec (0, p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_k^{n_k})$$

Damit ist zunächst die Überdeckung der ausgezeichneten Punkte abgesichert:

$$(0,0,\ldots,0) \prec (0,1)$$

Der lokale Einblick von einem Punkt $(n_1, n_2, \ldots, n_k)$ der Umwelt $\mathbb{N}^k$ aus ist

$$l(n_1, n_2, \ldots, n_k) = (sgn(n_1), sgn(n_2), \ldots, sgn(n_k)).$$

Da $sgn(n_i) = 1$ ist genau dann, wenn $z = p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_k^{n_k}$ durch p_i teilbar ist, muss $\mathfrak{A}$ ein Prozessor sein, der diese Teilbarkeit für alle beteiligten Primzahlen prüft. Dazu wird ganz ähnlich wie im Beweis des vorhergehenden Satzes in der Umwelt $\mathbb{N}^2$ die erste Koordinate der Punkte gebraucht. Überlegen wir zunächst, wie ein Automat $\mathfrak{A}_{p_i}$ aussieht, der die Teilbarkeit durch p_i prüft. Dabei können wir das Beispiel aus dem Abschnitt 2.4.2 auf Seite 73 als Vorlage benutzen.

```
actor 𝔄_{p_i}
declare label_i : {0,1}, counter_i : [p_i] end declare
begin
 counter_i := 0;
 while look_of (_,1) do
  action (1,−1);
  counter_i := counter_i + 1
 end while;
 if counter_i = 0 then
  label_i := 0 else label_i := 1
 end if;
 while look_of (1,_) do action (−1,1) end while;
end
```

Dieser Aktor belegt die Variable *label_i* mit dem Wert 0, wenn $z = p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_k^{n_k}$ teilbar ist durch p_i, anderenfalls ist $label\_i = 1$.

Nun lässt sich der Einblicksautomat aus solchen Aktoren wie folgt zusammensetzen:

automaton $\mathfrak{A}$ **returns_label** $\{0,1\}^k$
begin
 $\mathfrak{A}_{p_1}$;
 $\mathfrak{A}_{p_2}$;
  $\ldots$;
 $\mathfrak{A}_{p_k}$;
 return_label $(label\_1, label\_2, \ldots, label\_k)$
end

Bei den Aktoren zur Simulation der Aktionen

$$\mathbf{x} = (x_1, x_2, \ldots, x_k) \in \{1, 0, -1\}^k$$

kann man sich beschränken auf k-Tupel der Gestalt

$$\mathbf{x\_i} = (0, \ldots, 0, x_i, 0, \ldots, 0) \text{ mit } x_i \in \{1, -1\}.$$

Jede andere Aktion kann man aus solchen elementaren Aktionen kombinieren. Und wenn man die Aktoren für die **x_i** hat, kann man in analoger Weise die Aktoren für alle Aktionen **x** zusammensetzen. Zum Zwecke der Unterscheidung in den folgenden Programmen sei **x_i**_*plus* die Aktion mit $x_i = 1$. Ist $x_i = -1$, so soll die Aktion mit **x_i**_*minus* bezeichnet werden.

Wenn auf einen Punkt

$$\mathbf{n} = (n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$$

die Aktion **x_i**_*plus* angewendet wird, muss der Aktor $\mathfrak{A}_{\mathbf{x\_i\_}plus}$ den überdeckenden Punkt $(0, z)$ mit

$$z = p_1^{n_1} \cdot \ldots \cdot p_{i-1}^{n_{i-1}} \cdot p_i^{n_i} \cdot p_{i+1}^{n_{i+1}} \cdot \ldots \cdot p_k^{n_k}$$

überführen in $(0, z)\mathfrak{A}_{\mathbf{x\_i\_}plus} = (0, \tilde{z})$ mit

$$\tilde{z} = p_1^{n_1} \cdot \ldots \cdot p_{i-1}^{n_{i-1}} \cdot p_i^{n_i+1} \cdot p_{i+1}^{n_{i+1}} \cdot \ldots \cdot p_k^{n_k},$$

also z mit p_i multiplizieren.

Das leistet das folgende Programm:

actor $\mathfrak{A}_{\mathbf{x\_i\_}plus}$
declare $counter\_i : [p_i]$ **end declare**
begin
 $counter\_i := 0$;

```
 while look_of (_, 1) do
  repeat
   action (1, 0);
   conter_i := conter_i + 1
  until counter_i = 0;
  action (0, −1)
 end while
 while look_of (1, _) do action (−1, 1) end while
end
```

Man erkennt, dass die Zahl p_i durch diesen Aktor z-mal auf die erste Koordinate addiert wird. Zum Schluss muss, um der Überdeckungsdefinition Genüge zu tun, der Inhalt er ersten Koordinate wieder auf die zweite "umgeschichtet" werden.

In entsprechender Weise muss ein Aktor arbeiten, der eine Aktion **x_i_***minus* simuliert: er muss z durch p_i dividieren, allerdings vorher prüfen, ob z überhaupt eine solche Division erlaubt. Dazu wird wieder der oben definiert Aktor $\mathfrak{A}_{p_i}$ benutzt.

```
actor 𝔄x_i_minus
begin
 𝔄p_i;
 if label_i=1 then
  counter_i := 0;
  while look_of (0, 1) do
   repeat
    action (0, −1);
    conter_i := conter_i + 1
   until counter_i = 0;
   action (1, 0)
  end while;
  while look_of (1, _) do action (−1, 1) end while
 end if
end
```

Bei diesem Satz fällt der ausgezeichnete Punkt $(0, 1) \in \mathbb{N}^2$ etwas aus der Reihe. Für den Vergleich mehrerer Umwelten ist es erforderlich, ergänzend die Überdeckung $(\mathbb{N}, (0, 1)) \prec (\mathbb{N}, (0, 0))$ zu zeigen. Nach Satz 3.3.3 auf Seite 114 reicht es aus,

Lemma 1 $(\mathbb{N}, 1) \prec (\mathbb{N}, 0)$

nachzuweisen.

Beweis: Bei der Definition der Überdeckung wird die Null der Umwelt $(\mathbb{N}, 1)$ außer acht gelassen:

$$m \prec n \text{ genau dann, wenn } m > 0 \text{ und } m - 1 = n$$

Auf die Formulierung der Programme für die Automatenfamilie, welche die Überdeckung simuliert, verzichten wir, weil sie sehr einfach sind. Der Einblicksprozessor $\mathfrak{A}$ braucht lediglich eine Eins zu senden. Der Inkrementaktor $\mathfrak{A}_1$ produziert stets die Aktion 1. Der dekrementierende Aktor $\mathfrak{A}_{-1}$ liefert fast immer nichts anderes als die Aktion -1. Im Punkte Null tut er nichts: Im überdeckten Punkt 1 wäre die Aktion -1 zwar ausführbar. Aber $d(1, -1) = 0$ gehört nicht zum Definitionsbereich der Überdeckungsrelation.

Jetzt werden wir zeigen, dass $\mathbb{N}^2$ mindestens die gleiche Rechenkraft hat wie eine Stackumwelt.

Überdeckungssatz 3 *Eine Stack-Umwelt wird durch zwei Zählerumwelten überdeckt:*

$$(S_\Sigma, \varepsilon) \prec (\mathbb{N}, 0)^2 = (\mathbb{N} \times \mathbb{N}, (0, 0)) \tag{3.3}$$

Beweis: Sei $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{n-1}\}$ das Alphabet, mit dessen Buchstaben die Wörter gebildet werden, welche die Punkte der Stack-Umwelt sind. Den Buchstaben a_i werden durch die Funktion

$$\begin{aligned} a : \Sigma &\to \mathbb{N} \\ a_i &\mapsto a(\sigma_i) = i \end{aligned}$$

in kanonischer Weise natürliche Zahlen zugewiesen. Diese Abbildung kann man auf Wörter $w = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_m}$ fortsetzen durch eine so genannte n-äre Kodierung:

KODIERUNG VON WÖRTERN DURCH ZAHLEN

$$\begin{aligned} a(w) = a(\sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_m}) &:= i_1 \cdot n^{m-1} + i_2 \cdot n^{m-2} + \ldots + i_{m-1} \cdot n + i_m \\ a(\varepsilon) &:= 0 \end{aligned}$$

Wie muss ein Prozessor $\mathfrak{A}$ in der Umwelt der natürlichen Zahlen beschaffen sein, der $l(w) = a_i$ liefert? Er muss den Rest $a(w)\ mod\ n = i_m$ ermitteln und dazu aus dem Alphabet den zugehörigen Buchstaben suchen. Diese Rechnung ist für den Fall $n = 3$ schon beschrieben und in $\mathbb{N}$ allein durchführbar. Es erweist sich aber, dass man dennoch in $\mathbb{N}\times\mathbb{N}$ arbeiten muss, damit der Startpunkt nicht verloren geht.

Deshalb wird definiert:

$$w = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_m} \prec (a(w), 0)$$

Damit kann man die Überdeckungsdefinition verifizieren. Jedenfalls gilt offenbar, dass sich die ausgezeichneten Punkte überdecken:

$$\varepsilon \prec (a(\varepsilon), 0) = (0, 0)$$

Für die Definition des Einblicksautomaten $\mathfrak{A}$ wird eine Variable *counter* so verwendet, wie es im Abschnitt 2.4.2 auf Seite 73 beschrieben ist.

Eine zweite Variable y repräsentiert die ebenfalls endliche Umwelt der Einblicke von S_Σ, also die Elemente der Menge $\Sigma \cup \{\varepsilon\}$. In diesem Falle ist (vergl. ebenfalls 2.4.2) $l(\sigma) = \sigma$ und $d(\sigma, \tau) = \tau$

```
    automaton 𝔄 returns_label Σ ∪ {ε}
    declare counter : [n]; y : Σ ∪ {ε} end declare
    begin
(1)   if look_of (0, _) then y := ε
      else
(2)    counter := 0;
(3)    repeat
         action (−1, 1); counter := counter + 1
       until look_of (0, _);
(4)    repeat action (1, −1) until look_of (_, 0);
(5)    case of
(6)      counter = 1 then y := σ_1 |
(7)      counter = 2 then y := σ_2 |
                 ...
(8)      counter = n − 1 then y := σ_{n−1}
       end case
      end if;
      return_label y
    end
```

Nur scheinbar aufwändiger ist es, die Aktionen der Stackumwelt durch natürliche Zahlen zu simulieren. Wenn an ein Wort $w = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_m}$ ein Buchstabe σ_j angehängt wird, also $w' = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_m}\sigma_j$ entsteht, bedeutet dies, dass aus $a(w)$ die Zahl $a(w') = a(w) \cdot n + j$ gebildet werden muss. Umgekehrt, wenn der letzte Buchstabe σ_m eines Wortes w gestrichen wird, so muss m von $a(w)$ subtrahiert werden und das Ergebnis anschließend noch durch n dividiert werden.

Es folgt der Aktor $\mathfrak{A}_x$ in $\mathbb{N} \times \mathbb{N}$, wenn $x = a_j$ ein Buchstabe ist, der an ein Wort angehängt wird oder, wie man traditionell sagt, "gepusht" wird. In diesem Programm wird an zwei Stellen ein anderer Aktor "**actor**_$(0, k)$" aufgerufen, der in $\mathbb{N} \times \mathbb{N}$ lediglich k-mal die Aktion $(0, 1)$ ausführen soll.

```
actor 𝔄_{a_j}
begin
```

Programmzeile	$\mathbb{N}$	$\mathbb{N}$	*counter*	y
start	126	0	$\perp$	$\perp$
(2)	126	0	0	$\perp$
(3)	125	1	1	$\perp$
(3)	124	2	2	$\perp$
(3)	123	3	3	$\perp$
(3)	122	4	0	$\perp$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
(3)	0	126	2	$\perp$
(4)	126	0	2	$\perp$
(7)	126	0	2	β

Tabelle 3.1: Weg des Einblicksautomaten

repeat
actor_$(0, n)$;
action (-1,0)
until look_of (0,_);
actor_$(0, j)$;
repeat action (1,-1) **until look_of** (_,0)
end

Der Aktor $\mathfrak{A}_{pop}$ wird als Übungsaufgabe empfohlen.

MIT EINEM BEISPIEL WIRD ES KLARER

In diesen und allen anderen Programmen stellt sich immer die Frage nach der Korrektheit: löst das Programm die gestellte Aufgabe. An früherer Stelle, bei Registerumwelten, wurde einmal ein Korrektheitsbeweis geführt. Das wird hier vermieden. Wenn es schwer fällt, ein Programm zu verstehen, so hilft immer ein Beispiel. Im vorliegenden Fall nehmen wir einmal an, dass

$$\Sigma = \{\sigma_1 = \alpha, \sigma_2 = \beta, \sigma_3 = \gamma\}$$

ist, demnach $n = 4$.

Für $w = \alpha\gamma\gamma\beta$ bekommt man

$$a(w) = 1 \cdot 4^3 + 3 \cdot 4^2 + 3 \cdot 4 + 2 = 126$$

Startet man $\mathfrak{A}$ im Punkte $(126, 0) \in \mathbb{N} \times \mathbb{N}$, erhält man den in der Tabelle 3.1 aufgelisteten Ablauf von Werten, die den Weg des Automaten in der Umwelt $\mathbb{N} \times \mathbb{N} \times [3] \times (\Sigma \cup \{\varepsilon\})$ darstellen. Hingewiesen sei insbesondere noch auf die Programmzeile (4). Dort wird sicher gestellt, dass die in der Definition geforderte Überdeckung erreicht wird, wenn der Prozessor seine Arbeit beendet hat.

Mit den folgenden Sätzen werden die Beziehungen zwischen Turing-, Stack- und *RAM*-Umwelt geklärt.

Überdeckungssatz 4

$$(RAM, (0, \omega)) \prec (S_{\Sigma_0}, \varepsilon)^2 \quad mit\ \Sigma_0 = \{|, \flat\}$$

Wie immer zuerst die Idee

Beweis: Wie fast immer bei diesen Aussagen ist die Beweisidee einfach zu beschreiben. Den auf dem Band der RAM-Umwelt abgelegten natürlichen Zahlen werden in simpler Weise Wörter zugeordnet, die lediglich aus zwei Buchstaben gebildet sind: der Zahl $n > 0$ entspricht das Wort $\flat|^n$, der Null der Buchstabe $\flat$. Auf diese Weise wird die Belegung des Bandes in ein Wort aus $\Sigma_0^\star$ umgewandelt. Von diesem Wort kommt alles, was links von der aktuellen Position des Schreib-Lese-Kopfes steht, in den einen Stack, der andere Teil wird in umgedrehter Reihenfolge im zweiten Stack abgelegt. Dabei wird der Teil des RAM, der "rechts vom Träger liegt", nicht berücksichtigt oder nur bis zur Position des Schreib-Lese-Kopfes. Die Figur 3.5 zeigt die Verfahrensweise an einem Beispiel.

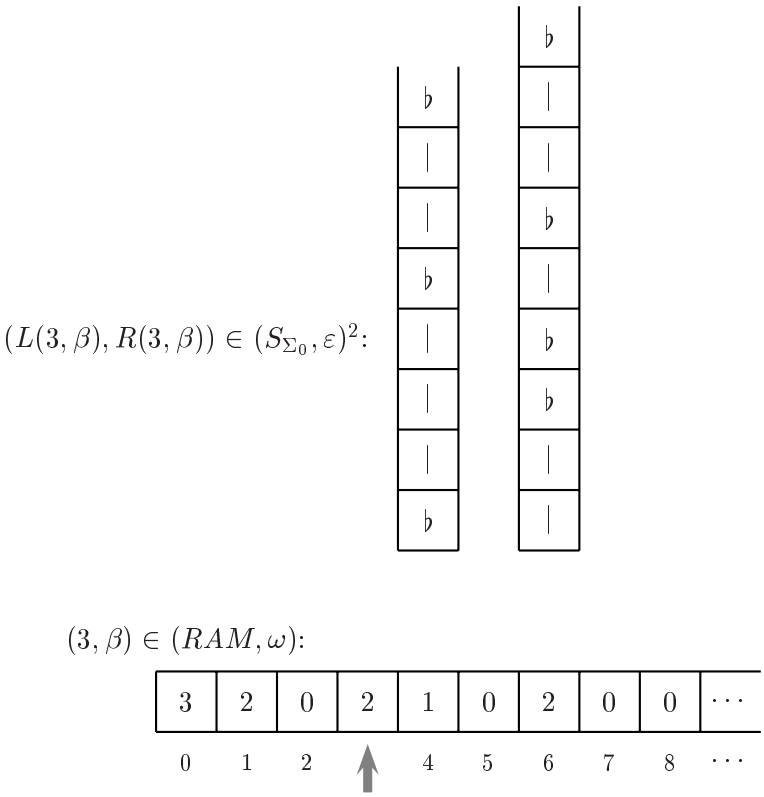


Abbildung 3.5: Die Überdeckung eines RAM-Punktes durch ein Paar von Stack-Punkten

Man erkennt leicht, wie sich aus den beiden Stackpunkten die Einblicke der RAM-Umwelt ergeben und wie Aktionen im RAM in den beiden Stacks simuliert werden müssen.

Diese Idee muss nun formalisiert werden, um die Details zu erfassen.

Sei (n, β) mit $\beta : \mathbb{N} \to \mathbb{N}$ ein Punkt des RAM und $\kappa - 1$ die größte der Zahlen $i \in \mathbb{N}$ mit $\beta(i) > 0$. Dann ist $\beta(i) = 0$ für alle $i \geq \kappa$ und $\beta = \omega$, wenn $\kappa = 0$ ist. Wir fassen natürliche Zahlen als Buchstaben des unendlichen Alphabets $\mathbb{N}$ auf und definieren eine Funktion $\varphi : \mathbb{N}^\star \to \Sigma_0^\star$ wie folgt:

$$\begin{aligned} \varphi(0) &= \flat \\ \varphi(p) &= \flat|^p \text{ für } p \in \mathbb{N},\ p > 0 \\ \varphi(p_1 p_2 \dots p_k) &= \varphi(p_1)\varphi(p_2)\dots\varphi(p_k). \end{aligned}$$

Die allgemeine, mathematische Darstellung

Nun wird festgelegt, wie die Codierung von β in einen mit $L(n, \beta)$ bezeichneten linken und einen rechten Teil $R(n, \beta)$ zerlegt wird:

$$L(n,\beta) = \begin{cases} \varepsilon & \text{falls } n = 0 \\ \varphi(\beta(0)\beta(1)\dots\beta(n-1)) & \text{sonst} \end{cases}$$

$$R(n,\beta) = \begin{cases} \varphi(\beta(\kappa-1)\beta(\kappa-2)\dots\beta(n+1)\beta(n)) & \text{falls } n < \kappa \\ \varepsilon & \text{sonst} \end{cases}$$

Damit kann nun die Überdeckung definiert werden:

$$(n, \beta) \prec (L(n,\beta), R(n,\beta))$$

Offenbar ist die erste Forderung der Überdeckungsdefinition erfüllt. Der ausgezeichnete Punkt der RAM-Umwelt wird korrekt überdeckt:

$$(0, \omega) \prec (\varepsilon, \varepsilon)$$

Wie gewinnt man aus den beiden Stacks den Einblick im überdeckten Punkt des RAM?

Ob der Schreib-Lese-Kopf im RAM auf der Null steht oder rechts davon, erkennt man daran, ob $R(n, \beta)$ das leere Wort ist oder nicht. Ebenso einfach ist es festzustellen, ob auf der aktuellen Position im RAM eine positive Zahl steht oder die Null: wenn $R(n, \beta) \neq \varepsilon$ ist und an zweiter Stelle ein Strich, so ist $\beta(n) > 0$, anderenfalls gleich Null. Daraus ergibt sich das folgende Programm.

```
automaton 𝔄 returns_label {0,1}²
declare label1, label2 : {0,1} end declare
begin
  if look_of (ε, _) then
    label1 := 0 else label1 := 1
  end if;
  if look_of (_, ♭)
```

```
    then
    action (stop, pop);
    if look_of (_, |) then label1 := 1 end if;
    action (stop, ♭)
    else
      label2 := 0
  end if;
  return_label (label1, label2)
end
```

Von den Aktoren zur Simulation der Aktionen werden wir nur zwei angeben. Die anderen lassen sich in analoger Weise konstruieren. Beginnen wir mit der Aktion re, das heißt wenn der Schreib-Lese-Kopf im RAM einen Platz nach rechts rückt. Wenn $R(n, \beta) = \varepsilon$ ist, muss zu $L(n, \beta)$ ein $\flat$ "gepusht" werden, im anderen Fall muss der Inhalt des rechten Stacks bis zum nächsten $\flat$ oder bis er leer ist auf den linken Stack umgestapelt werden.

```
actor 𝔄_re
begin
  if look_of (_, ε) then action (♭, stop)
    else
      action (♭, pop);
      while look_of (_, |) do
        action (|, pop)
      end while
  end if
end
```

Will man die Aktion -1, d.h. die Verminderung der Zahl an der aktuellen Position um 1, durch einen Aktor in den beiden Stackumwelten modellieren, muss man sich davon überzeugen, dass sie in der RAM-Umwelt auch ausführbar ist: es muss $\beta(n) > 0$ sein. Das ist genau dann der Fall, wenn $R(n, \beta) = \flat|w$ mit $w \in \Sigma_0^\star$ ist. Wenn dieser Fall vorliegt ist, muss aus $R(n, \beta)$ das neue Wort $L(n, \beta') = \flat w$ werden. Das tut der folgende Aktor.

```
actor 𝔄_(−1)
begin
  action(stop, pop);
  if look_of (_, |) then action (stop, pop) end if;
  action (stop, ♭)
end
```

Damit beenden wir diesen Beweis. Die restlichen beiden Aktoren werden als Aufgaben empfohlen.

Überdeckungssatz 5

$$(S^2_\Sigma, (\varepsilon, \varepsilon)) \prec (T_{\Sigma \sqcup \{\flat\}}, (0, \omega)) \prec (S^2_{\Sigma \sqcup \{\flat\}}, (\varepsilon, \varepsilon))$$

Beweis: Es wird erneut die Strategie des Beweises zum vorhergehenden Satz angewendet (vergl. die Abbildung 3.6).

Zwei Stacks haben die selbe Rechenkraft wie eine Turingumwelt

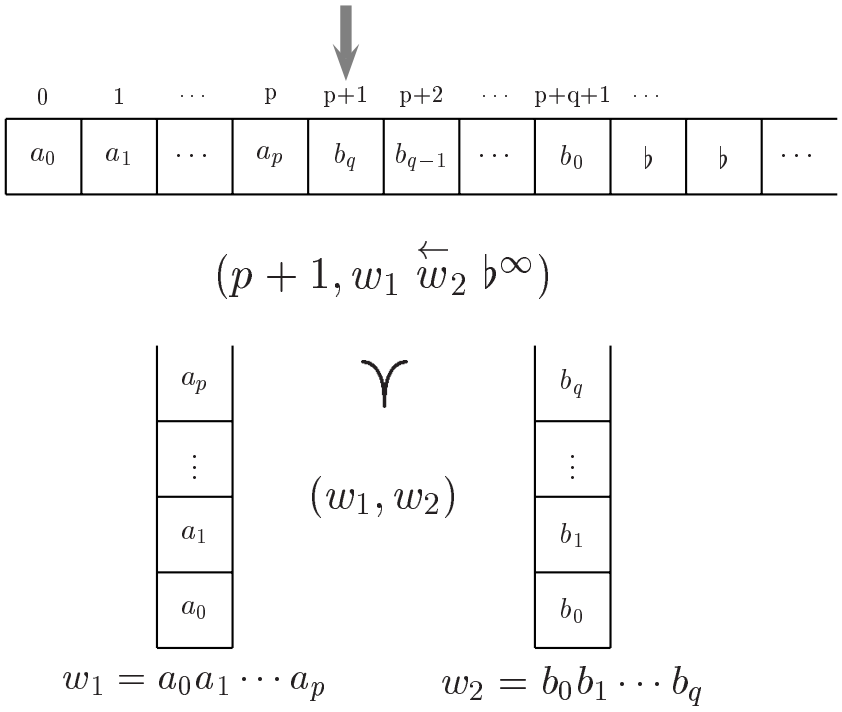


Abbildung 3.6: So wird ein Paar von Stackpunkten durch einen Turingpunkt überdeckt

Um die Überdeckung

$$(S^2_\Sigma, \varepsilon) \prec (T_{\Sigma \sqcup \{\flat\}}, (0, \omega))$$

zu definieren, wird einem Punkt $(w_1, w_2) \in S_\Sigma \times S_\Sigma$ mit

$$w_1 = a_0 a_1 \cdots a_p \in \Sigma^\star \text{ und}$$
$$w_2 = b_0 b_1 \cdots b_q \in \Sigma^\star$$

die Funktion

$$\beta_{w_1 w_2} : \mathbb{N} \to \Sigma \sqcup \{\flat\}$$
$$n \mapsto \beta_{w_1 w_2}(n) = \begin{cases} a_n & \text{für } 0 \leq n \leq p \\ b_{p+q+1-n} & \text{für } p+1 \leq n \leq p+q+1 \\ \flat & \text{sonst} \end{cases}$$

zugewiesen. Es ist übersichtlicher, wenn man für diese Funktion eine andere Notation wählt. Wir werden sie als unendlich langes Wort $\beta = \beta(0)\beta(1)\cdots\beta(i)\cdots$ aufschreiben, dabei aber den Rest, der für $n \geq p+q+2$ nur aus $\flat's$ besteht, in der Gestalt $\flat^\infty$ notieren. Schließlich sei

$$\overleftarrow{w}_2 := b_q b_{q-1} \cdots b_0$$

und

$$|w| = |a_0 a_1 \cdots a_p| := p+1;\ |\varepsilon| = 0$$

die Länge eines Wortes. Mit diesen Vereinbarungen ist

$$\beta_{w_1 w_2} = a_0 a_1 \cdots a_p b_q b_{q-1} \cdots b_0 \flat^\infty = w_1\ \overleftarrow{w}_2\ \flat^\infty$$

Die Überdeckung von Punkten beider Umwelten wird nun definiert durch

$$(w_1, w_2) \prec (|w_1|, \beta_{w_1 w_2})$$

Offenbar überdecken sich die ausgezeichneten Punkte:

$$(\varepsilon, \varepsilon) \prec (|\varepsilon|, \flat^\infty) = (0, \omega)$$

Die Einblicke in die Stacks ermittelt der folgende Prozessor:

```
automaton 𝔄 returns_label (Σ ⊔ {♭}) × (Σ ⊔ {♭})
declare label1, label2 : Σ ⊔ {♭} end declare
begin
  look → (_, label2);
  if  label2 = ♭ then label2 = ε end if
  if look_of (0, _) then label1 = ε
    else
      action (li);
      look → (_, label1);
      action (re)
    end if;
    return_label (label1, label2)
  end
```

Auch in diesem und den folgenden Programmen bezeichnen die Aktionen re bzw. li wieder die Rechts- bzw. Linksbewegung des Schreib-Lese-Kopfes, diesmal in der Turingumwelt.

Zur Simulation der Aktionen in den Stackumwelten müssen auf dem Turingband einigermaßen aufwändige Verschiebungen vorgenommen werden, von deren Ausführbarkeit wir uns schon mehrfach überzeugt haben, weshalb hier nur noch ein Fall exemplarisch dargestellt wird, nämlich wenn aus $(w_1, w_2) \in S_\Sigma \times S_\Sigma$ vermittels der Aktion $x = (\sigma, stop)$; $\sigma \in \Sigma$ der neue Punkt $(w_1\sigma, w_2)$ wird. Auf dem Turingband muss in diesem Falle zwischen w_1 und $\overleftarrow{w_2}$ der Buchstabe σ eingefügt werden:

```
actor 𝔄_(σ,stop)
declare letter1, letter2 : Σ ⊔ {♭} end declare
begin
  look → (_, letter1);
  action (♭); action (re);
  while not look_of (_, ♭) do
    look → (_, letter2);
    action (letter1); action (re);
    letter1 := letter2
  end while;
  action (letter1); action (li);
  while not look_of (_, ♭) do
    action (li)
  end while;
  action (σ)
end
```

Um diesen ersten Teil des Beweises vollständig zu machen, sind die Aktoren für die Aktionen $(pop, stop)$, $(stop, \sigma)$ und $(stop, pop)$ nach zu tragen. Das wird als nützliche Übung empfohlen.

Die Vorgehensweise für den zweiten Teil des Beweises ist offensichtlich. Um die Überdeckung zweier Stackpunkte durch einen Turingpunkt zu definieren, wurden sie zu einer Beschriftung des Bandes zusammengesetzt, wobei das zweite Wort umgedreht und durch eine unendliche Folge von ♭'s ergänzt wurde. Jetzt wird die Beschriftung des Bandes vor der Position des Schreib-Lese-Kopfes zerschnitten, der unwichtig Bestandteil $♭^\infty$ weg gelassen und das zweite Wort umgedreht.

Formal sieht das folgendermaßen aus: Sei $(p, \beta) \in T_{\Sigma\sqcup\{♭\}}$ und $\kappa_\beta - 1$ die größte der Zahlen j mit $\beta(j) \neq ♭$. (Offenbar ist dann $\kappa_\omega = 0$.) Mit dieser Vereinbarung

wird definiert:

$$\begin{aligned} w_1 &:= \beta(0)\beta(1)\cdots\beta(p-1) \\ u &:= \beta(p)\beta(p+1)\cdots\beta(k-1) \\ w_2 &:= \overleftarrow{u} \\ (p,\beta) &\prec (w_1, w_2) \end{aligned}$$

Das ist, abgesehen davon, dass wir hier ein anderes Alphabet haben, die gleiche Vorgehensweise wie im Beweis des vorigen Satzes, so dass mit diesem Hinweis der Beweis beendet werden kann.

Die Sätze dieses Abschnittes werden durch die anschließende Folgerung zusammengefasst: Alle hier betrachteten Umwelten haben gleiche Rechenkraft. Damit wird das im Satz 2 auf Seite 120 von Minsky gefundene Resultat vertieft.

Folgerung 3.4.1 *Sei $p \geq 2$ und Σ bestehe aus mindestens zwei Buchstaben. Dann gilt:*

$$(\mathbb{N},0)^2 \equiv (\mathbb{N},0)^p \equiv (RAM,(0,\omega)) \equiv (T_{\Sigma\sqcup\{\flat\}},(0,\omega)) \equiv (S_\Sigma,\varepsilon)^2$$

DAS GROSSE RESUME

Beweis: Im folgenden ist $\overset{i}{\prec}$ eine mit dem Überdeckungssatz i bewiesene Überdeckung. Ferner sei wieder $\Sigma_0 = \{\flat, |\}$. Dann haben wir:

$$\begin{aligned} (\mathbb{N},0)^p &\overset{2}{\prec} (\mathbb{N},0)^2 \overset{1}{\prec} (RAM,(0,\omega)) \overset{4}{\prec} (S_{\Sigma_0},\varepsilon)^2 \prec (S_\Sigma,\varepsilon)^2 \overset{5}{\prec} (T_{\Sigma\sqcup\{\flat\}},(0,\omega) \\ &\overset{5}{\prec} (S_{\Sigma\sqcup\{\flat\}},\varepsilon)^2 \overset{3}{\prec} (\mathbb{N},0)^4 \overset{2}{\prec} (\mathbb{N},0)^2 \prec (\mathbb{N},0)^p \end{aligned}$$

Die Überdeckung $(\mathbb{N},0)^2 \prec (\mathbb{N},0)^p$ folgt aus dem Satz 3.3.4 auf Seite 115. Die Beziehung $(S_{\Sigma_0},\varepsilon)^2 \prec (S_\Sigma,\varepsilon)^2$ gilt, weil man einen Stack mit einem anderen Stack überdecken kann, der ein umfangreicheres Alphabet hat.

Die Behauptung der Folgerung ergibt sich aus der Transitivität der Überdeckungsrelation.

Unter der Hand hat sich noch ein hübsches Resultat ergeben:

Korollar 3.4.1 *Bei einer Turingumwelt kann man sich ohne Einbuße an Rechenkraft auf ein Alphabet aus drei Buchstaben beschränken.*

Beweis: Anstelle von $\Sigma_0 = \{\flat, |\}$ wird jetzt $\Sigma_1 = \{\sharp, |\}$ benutzt. Σ ist wieder ein Alphabet mit mindestens zwei Buchstaben. Es gilt

$$(T_{\Sigma_1\sqcup\{\flat\}},(0,\omega)) \prec (T_{\Sigma\sqcup\{\flat\}},(0,\omega)) \equiv (RAM,(0,\omega)) \prec (S_{\Sigma_1},\varepsilon)^2 \prec (T_{\Sigma_1\sqcup\{\flat\}},(0,\omega))$$

	P_0	P_1	P_2	P_3	
Z	♭	† ♭ ♭ ♭ ♭	† ♭ ♭ ♭ ♭	† ♭ ♭ ♭ ♭	† ♭ ♭ ♭ ♭	...
Y	♭	0 ♭ ♭ ♭ ♭	1 ♭ ♭ ♭ ♭	2 ♭ ♭ ♭ ♭	0 ♭ ♭ ♭ ♭	...
$x_1 = 0$	♭	† † ♭ ♭ ♭	† † † † ♭	† ♭ ♭ ♭ ♭	† † ♭ ♭ ♭	...
$x_2 = 1$	♭	† † † † ♭	† † † ♭ ♭	† † † † ♭	♭ ♭ ♭ ♭ ♭	...
$Zeiger1$	♭	♭ ♭ ♭ ♭ ♭	♭ ♭ ♭ ♭ ♭	† ♭ ♭ ♭ ♭	♭ ♭ ♭ ♭ ♭	...
$Zeiger2$	†	♭ ♭ ♭ ♭ ♭	♭ ♭ ♭ ♭ ♭	♭ ♭ ♭ ♭ ♭	♭ ♭ ♭ ♭ ♭	...

Tabelle 3.2: Die Beschriftung des Turingbandes, wenn P_2 überdeckt wird.

Die beiden Turingumwelten in dieser Kette von Relationen sind äquivalent.

Dieser Abschnitt soll beendet werden mit einem Satz über endliche Umwelten, der im nächsten Kapitel benötigt wird. Sie sind bezüglich ihrer Rechenkraft die schwächsten. Man kann zeigen, dass sich jede endliche Umwelt (U, P_0) durch die Zählerumwelt $(\mathbb{N}, 0)$ überdecken lässt. Dazu werden den Punkten

$$Z = \{P_0, P_1, \ldots P_n\}$$

die Zahlen 0 bis n in dieser Reihenfolge zugeordnet:

$$P_i \prec i, \quad (0 \leq i \leq n).$$

Wenn man das ausführt, stellt man fest, dass die Familie der Prozessoren, welche die Überdeckung realisieren, für jede Umwelt (U, O) eine andere ist. Diese Automaten hängen nicht nur von den Einblicken und den Aktionen der endlichen Umwelt ab, sie werden auch von der Einblicks- und Wirkungsfunktion der jeweiligen Umwelt determiniert. Uns kommt es aber darauf an, eine Überdeckung zu konstruieren, die von der endlichen Umwelt weitgehend unabhängig ist. Das wird mit einer Turingumwelt gelingen.

Um das Verständnis für die Konstruktion im allgemeinen Fall zu erleichtern, beginnen wir mit einem Beispiel. Die Figur **??** zeigt die endliche Umwelt, die benutzt werden soll. Es ist $X = \{0, 1\}$, $Y = \{0, 1, 2\}$ und $Z = \{P_0, P_1, P_2, P_3\}$. Die Kanten des Graphen markieren die Wirkungsfunktion. Zum Beispiel ist $d(P_3, 0) = P_1$ und $d(P_3, 1) = \perp$

Zu dieser Umwelt wird das Alphabet

$$\Sigma = \{\dagger, \flat\} \times (Y \cup \{\flat\}) \times \{\dagger, \flat\}^4$$

gewählt, aus dessen Buchstaben eine zweckmäßige Beschriftung des Turingbandes gebildet wird. Wie das geschieht, zeigt die Tabelle 3.2

Der Inhalt jeder Spalte ist jeweils ein Buchstabe des Alphabets Σ. Zu jedem Punkt $P_i \in Z$ gehört ein Wort aus fünf Buchstaben. Die Trennlinien zwischen den Buchstaben dieser Wörter sind der Übersichtlichkeit wegen nicht eingezeichnet. In den Spalten hinter P_3 stehen nur noch $\flat's$.

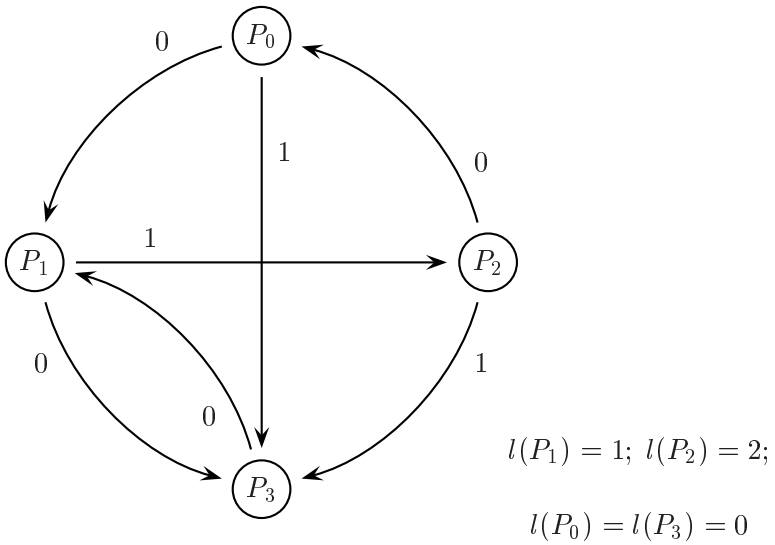


Abbildung 3.7: Eine endliche Umwelt mit vier Punkten

Die zweite Zeile enthält ersichtlich die Einblicke. In der dritten und vierten Zeile ist die Wirkungsfunktion d abgelegt. Dabei wird die Funktion

$$\begin{aligned} \iota : Z = \{P_0, P_1, P_2, P_3\} &\to \{\dagger, \flat\}^\star \\ P_i &\mapsto \dagger^{i+1}\flat^{4-i} \end{aligned}$$

benutzt. Offenbar steht in den beiden Zeilen, die vorne durch $x = 0$ bzw. $x = 1$ bezeichnet sind, unter P_i jeweils $\iota(d(P_i, x))$.

In den letzten beiden Zeilen wird das Zeichen † als Zeiger benutzt. Die in der Tabelle 3.2 vorgestellte Beschriftung des Bandes ist vermittels der Position von "Zeiger1" dem Punkte P_2 zugeordnet. Wenn die Punktmenge Z sowie die Einblicksfunktion $l : Z \to Y$ und die Wirkungsfunktion $d; Z \times X \to Z$ der endlichen Umwelt gegeben sind, dann ist die Beschriftung des Turingbandes bis auf die vorletzte Zeile definiert. Die endgültige Beschriftung hängt nur noch davon ab, welchen Punkt der Umwelt man hervorheben will. Sei α_{P_i} die dem Punkte P_i entsprechende Beschriftung. Die Überdeckung

$$(U, P_0) \prec (T_\Sigma, (0, \alpha_{P_0}))$$

wird durch $P_i \prec (0, \alpha_{P_i})$ definiert.

Wie gewinnt man den Einblick aus dieser Beschriftung des Turingbandes? Der Schreib-Lese-Kopf muss die Stelle suchen, wo in der Zeile $Zeiger1$ ein † steht und den Wert in der Y-Zeile ablesen:

automaton $\mathfrak{A}$ **returns_label** Y
declare $einblick : Y$ **end declare**
begin
  repeat action (re) **until look_of** $(\_, (\_, \dots, \_, \dagger, \_))$;
  look $\rightarrow (\_, (\_, einblick, \_, \dots, \_))$;
  repeat action (li) **until look_of** $(0, (\_, \dots, \_))$;
  returns_label $einblick$
end

Dieses Programm ist so angelegt, dass es auch gültig ist, wenn die Zahl der Umweltpunkte beliebig ist. Dasselbe wird für die Aktoren zur Modellierung der Aktionen gelten. Das erfordert aber mehr Aufwand.

Zunächst eine Erinnerung. Eine Turingumwelt $T_{(\Sigma,O)}$ benutzt als Faser die endliche Alphabetumwelt (Σ, O) mit

$$Z = \Sigma = Y,\ X = \Sigma \cup \{stop\},\ l(\sigma) = \sigma \text{ und } d(\sigma_1, \sigma_2) = \sigma_2 \text{ bzw. } d(\sigma, stop) = \sigma$$

Im einfachsten Falle ist

$$\Sigma = \{O = \sigma_0 = \flat, \sigma_1, \cdots, \sigma_n\}.$$

Im vorliegenden Falle ist die Sachlage komplex:

$$\Sigma = \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_p,\ O = (\flat, \flat, \dots, \flat)$$

Aktionen sind ebenfalls p-Tupel, wobei allerdings in unserem Falle immer nur eine Komponente aktiv sein wird, etwa in der Gestalt

$$\textbf{action } (\sigma, stop, \dots, stop),$$

was der Gleichung

$$d((\sigma_1, \sigma_2, \dots, \sigma_p), (\sigma, stop, \dots, stop) = (\sigma, \sigma_2, \dots, \sigma_p)$$

entspricht.

Nun zu den Aktoren für die Aktionen in der endlichen Umwelt. Zunächst werden nach dem Prinzip "teile und herrsche" einige elementare Aktoren zusammengestellt.

Der folgende Prozessor führt lediglich den Schreib-Lese-Kopf in die Nullposition zurück:

actor $zurNullposition$
begin
  while look of $(1, (\_, \dots, \_))$ **do action** (li) **end while**
end

Die Bewegung der Zeiger erfordert mehr Aufwand. Wir beschränken uns auf den Fall, dass $Zeiger1$ um eine Stelle nach rechts bewegt wird:

```
actor Zeiger1_Re
begin
  zurNullposition;
  while look_of (_, (_, ... , _, ♭, _)) do action (re) end while;
  action (stop, ... , stop, ♭, stop);
  action (re);
  action (stop, ... , stop, †, stop)
end
```

Es wird nötig sein, den $Zeiger1$ auf die Position von $Zeiger1$ zu verschieben. Das wird mit dem folgenden Programm vorbereitet. Dieser Aktor bewegt $Zeiger1$ von einer beliebigen Position ganz nach rechts.

```
actor Zeiger1zurNullposition
begin
  zurNullposition;
  while look_of (_, (_, ... , _, ♭, _)) do action (re) end while;
  while look_of (1, (_, ... , _)) do
    action (stop, ... , stop, ♭, stop);
    action (li);
    action (stop, ... , stop, †, stop);
  end while;
end
```

Jetzt die Bewegung von $Zeiger1$ zur Position von $Zeiger2$:

```
actor Zeiger1zuZeiger2
begin
  Zeiger1zurNullposition;
  while look_of (_, (_, ... , _, ♭)) do Zeiger1_Re end while
end
```

Jetzt sind alle Vorbereitungen getroffen, um einen Aktor für die Emulation einer Aktion zu formulieren. Wir beschränken uns auf die Aktion $x_1 = 0$.

actor $\mathfrak{A}_{x_1}$
begin
  while look_of $(\_, (\_, \ldots, \_, \flat, \_))$ **do action** (re) **end while**;
  while look_of $(\_, (\_, \_, \dagger, \_, \ldots, \_))$ **do**
    $Zeiger1\_Re$;
    repeat $Zeiger2\_Re$ **until look_of** $(\_, (\dagger, \_, \ldots, \_))$
  end while;
  $Zeiger1 zu Zeiger2$;
  $Zeiger2 zur Nullposition$;
  $zur Nullposition$
end

Das Beispiel hat folgendes gezeigt: Zu der endlichen Umwelt $(U, O) = (X, Y, Z, d, l)$ gibt es ein Alphabet Σ, mit dem zu jedem Punkt $P \in Z$ eine Beschriftung α_P des Turingbandes $T_{(\Sigma,\alpha_O)}$ hergestellt werden kann derart, dass

$$(U, O) \prec (T_\Sigma, (0, \alpha_O)), \text{ wenn } P \prec (0, \alpha_P) \text{ gesetzt wird.}$$

Dabei ist die Automatenfamilie, welche die Überdeckung realisiert, zwar abhängig von den endlichen Mengen X, Y und Z sowie von dem Alphabet Σ, welches daraus entsteht. Die Prozessoren sind aber unabhängig von der jeweiligen Beschriftungen des Bandes, die von der Zahl der Umweltpunkte, den Funktionen d und l sowie dem Punkte $P \in Z$ abhängt, der gerade überdeckt wird. Und darauf kommt es uns an.

Das Beispiel lässt sich unmittelbar zu dem folgenden Satz verallgemeinern.

Überdeckungssatz 6 *Seien X und Y zwei endliche Mengen. Es gibt ein endliches Alphabet Σ und eine Automatenfamilie*

$$(\{\mathfrak{A}_x \mid x \in X\}, \mathfrak{A}),$$

$$Y' \longrightarrow \boxed{\mathfrak{A}_x} \longrightarrow X' \qquad\qquad Y' \longrightarrow \boxed{\mathfrak{A}} \longrightarrow X' \quad (\downarrow Y)$$

mit

$$X' = \{re, li\} \sqcup \Sigma$$
$$Y' = \{0, 1\} \times \Sigma$$

und folgender Eigenschaft:

Wenn (U,O) mit $U = (X, Y, Z, d, l)$ eine beliebige endliche Umwelt ist, dann gibt es eine Überdeckung

$$(U, O) \prec (T_\Sigma, (0, \alpha_O)),$$

welche durch die Automatenfamilie $(\{\mathfrak{A}_x \mid x \in X\}, \mathfrak{A})$ realisiert wird.

		P_0		P_i		P_n	
$Z:$	$\flat$	$\dagger\flat^{n+1}$	$\cdots$	$\dagger\flat^{n+1}$	$\cdots$	$\dagger\flat^{n+1}$	$\cdots$
$Y:$	$\flat$	$l(P_0)\flat^{n+1}$	$\cdots$	$l(P_i)\flat^{n+1}$	$\cdots$	$l(P_n)\flat^{n+1}$	$\cdots$
$x_1:$	$\flat$	$\iota(d(P_0,x_1))$	$\cdots$	$\iota(d(P_i,x_1))$	$\cdots$	$\iota(d(P_n,x_1))$	$\cdots$
$x_2:$	$\flat$	$\iota(d(P_0,x_2))$	$\cdots$	$\iota(d(P_i,x_2))$	$\cdots$	$\iota(d(P_n,x_2))$	$\cdots$
$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$	
$x_q:$	$\flat$	$\iota(d(P_0,x_q))$	$\cdots$	$\iota(d(P_i,x_q))$	$\cdots$	$\iota(d(P_n,x_q))$	$\cdots$
$Zeiger1:$	$\flat$	$\flat^{n+2}$	$\cdots$	$\dagger\flat^{n+1}$	$\cdots$	$\flat^{n+2}$	$\cdots$
$Zeiger2:$	$\dagger$	$\flat^{n+2}$	$\cdots$	$\flat^{n+2}$	$\cdots$	$\flat^{n+2}$	$\cdots$

Tabelle 3.3: Beschriftung des Turingbandes zur Überdeckung einer beliebigen endlichen Umwelt. Sie entspricht dem Punkt $P_i \in Z$

Beweis:

Es ist nur erforderlich, das Beispiel für beliebige endliche Umwelten zu verallgemeinern.

Sei $Z = \{P_0, P_1, \ldots, P_n\}$, $X = \{x_1, x_2, \ldots, x_q\}$ und $Y^\flat := Y \sqcup \{\flat\}$. $(Y^\flat, \flat)$ und $(\{\dagger, \flat\}, \flat)$ seien Alphabetumwelten, aus denen die komplexe Umwelt

$$(\Sigma, (\flat, \ldots, \flat)) := (\{\dagger, \flat\}, \flat) \times (Y^\flat, \flat) \times (\{\dagger, \flat\}, \flat)^{q+2}$$

gebildet wird. Aus deren Buchstaben wird die Beschriftung des Turingbandes gebildet, die wir uns als eine Matrix mit unendlich vielen Spalten und $q+4$ Zeilen vorstellen können.

Nun wird die Beschriftung des Bandes in vollständiger Analogie zum Beispiel definiert. Sei

$$\iota : Z = \{P_0, P_1, \ldots, P_n\} \to \{\dagger, flat\}$$

definiert durch

$$\iota(P_i) := \dagger^{i+1}\flat^{n+1-i}.$$

Da mit dieser Funktion Worte definiert werden sollen, die Werten der Wirkungsfunktion d entsprechen, muss außerdem ein Funktionswert für ι festgelegt werden, wenn d nicht definiert ist. Wie im Beispiel setzen wir

$$\iota(\perp) := \flat^{n+2}.$$

Nun kann die Beschriftung α_{P_i} des Turingbandes angegeben werden, welche dem Punkte $P_i \in Z$ entspricht. Sie ist der Tabelle 3.3 zu entnehmen.

Die Überdeckung

$$(U, P_0) \prec (T_\Sigma, (0, \alpha_{P_0}))$$

wird durch $P_i \prec (0, \alpha_i)$ definiert.

Die Familie der Prozessoren, welche die Überdeckung realisieren, wurden mit dem Beispiel oben geliefert. Der Beweis ist abgeschlossen.

Kapitel 4

Mathematische Grundlagen

Dieser Anhang erläutert in knapper Darstellung die verwendeten, mathematischen Inhalte.

4.1 Mengen

Mengen werden von uns in der naiven Weise benutzt, wie sie von **Georg Cantor** vor etwa 120 Jahren eingeführt worden sind. Eine Menge ist charakterisiert durch die Elemente, die in ihr enthalten sind, wobei angenommen wird, dass man von zwei Elementen immer feststellen kann, ob sie sich unterscheiden oder gleich sind. Sind das endlich viele, dann kann man sie immer einfach aufschreiben. Zum Beispiel ist

$$M = \{a, b, 3, 7, katze, 3, 8\}$$

eine Menge. Die geschweiften Klammern sind ein auf der ganzen Welt üblicher Bezeichnungsstandard. Dass ein Element, wie im Beispiel die '3', mehrfach auftaucht, ist erlaubt. Wenn man dieses Element einmal streicht, ist es immer noch dieselbe Menge. Auch unendliche Mengen beschreibt man durch ihre Elemente, wenn sie allgemein bekannte Serien sind. Jedermann weiß zum Beispiel, dass

$$\mathbb{N} = \{0, 1, 2, \dots\}$$

die Menge der natürlichen Zahlen ist.

Will man ausdrücken, dass '3' eine natürliche Zahl ist, schreibt man $3 \in \mathbb{N}$. Dass 3.4 nicht zu $\mathbb{N}$ gehört, schreibt man so: $3.4 \notin \mathbb{N}$

Wenn man eine Menge, sozusagen eine Grundmenge zur Verfügung hat, dann ist es möglich, Teilmengen davon durch Eigenschaften von Elementen, durch so genannte Prädikate zu definieren. Die Menge der geraden natürlichen Zahlen kann man so schreiben:

$$\mathbb{N}_g = \{n \in \mathbb{N} \mid \text{ es gibt ein } k \in \mathbb{N} \text{ mit } n = 2k\}.$$

Lesart: Die geraden natürlichen Zahlen sind diejenigen Elemente n aus $\mathbb{N}$, für welche es eine natürliche Zahl k derart gibt, dass $n = 2k$ ist.

Eine besondere Bedeutung hat die Menge, welche keine Elemente enthält, die **leere Menge**, die mit dem Symbol $\emptyset$ bezeichnet wird.

Mit Mengen kann man in ganz elementarer Weise rechnen.

4.1.1 Mengeninklusion, Mengengleichheit

- Eine Menge A ist Teilmenge einer anderen Menge B, wenn jedes Element von A auch zu B gehört. Schreibweise: $A \subseteq B$. Stets ist $\emptyset \subseteq A$
- Zwei Mengen A und B sind gleich, wenn $A \subseteq B$ und $B \subseteq A$
- $A \subseteq B$ schließt $A = B$ nicht aus. Wenn man ausdrücken will, dass $A \subseteq B$ und $A \neq B$ ist, wird $A \subset B$ verwendet. Man sagt dann auch, dass A **echte Teilmenge** von B sei.

4.1.2 Mengenoperationen

Das im folgenden benutzte Symbol ":=" soll als "ist per Definition gleich" gelesen werden. Diese Bezeichnung wollen wir aber auch so verwenden, wie es in Programmiersprachen üblich ist, nämlich als Zuweisungsvorschrift für die Belegung von Variablen. Z.B. bedeutet $x := 3x + 5$, dass aus einer Belegung der Variablen x mit 4 die neue Belegung 17 geworden ist.

Jetzt wird angenommen, dass die Mengen A und B Teilmengen einer Menge X sind.

- Der **Durchschnitt** der Mengen A und B besteht aus allen Elementen (von X), die sowohl zu A als auch zu B gehören:

  $$A \cap B := \{x \in X \mid x \in A \textbf{ und } x \in B\}$$

  Die Mengen A und B heißen **disjunkt**, wenn $A \cap B = \emptyset$ ist.

- Die **Vereinigung** der beiden Mengen wird gebildet von den Elementen, die in A oder in B enthalten sind. Die Konjunktion "oder" muss dabei unterschieden werden von "entweder oder". Zum Beispiel ist ein Element $a \in A$ in der Vereinigungsmenge enthalten, auch wenn es nicht zu B gehört.

  $$A \cup B := \{x \in X \mid x \in A \textbf{ oder } x \in B\}$$

- Es ist insbesondere in der Informatik häufig erforderlich, Mengen A und B mit nicht leerem Durchschnitt zu vereinigen, jedoch so, dass in der Vereinigungsmenge ersichtlich ist, welche Elemente aus A oder B sind. Zum

Beispiel können 0 und 1 aus A Zahlen sein, wogegen 0 und 1 aus B Einblicke in einer Umwelt bezeichnen. Um diese gleich bezeichneten, jedoch mit unterschiedlichen Bedeutungen verknüpften Elemente in der Vereinigungsmenge auseinander halten zu können, muss man sie entsprechend markieren. Zum Beispiel kann man zu A die Menge $\tilde{A} := \{(\alpha, a) \mid a \in A\}$ und in entsprechender Weise zu B die Menge $\tilde{B} := \{(\beta, b) \mid b \in B\}$ bilden. Dann ist

$$A \sqcup B := \tilde{A} \cup \tilde{B}.$$

Diese Art der Verknüpfung von Mengen wird **freie Vereinigung** genannt.

- Bei der Bildung der **Differenz** zweier Mengen werden die Elemente einer Menge aus der anderen eliminiert:

$$A \setminus B := \{x \in X \mid x \in A \textbf{ und } x \notin B\}$$

- Das **Komplement** einer Menge A besteht aus allen Element der Grundmenge X, die nicht zu A gehören:

$$\mathbb{C}A := \{x \in X \mid x \notin A\} = X \setminus A$$

Aus diesen Definitionen ergeben sich unter anderem die folgenden Gleichungen:

$$A \cap B = B \cap A \qquad A \cup B = B \cup A \tag{4.1}$$
$$A \cap \emptyset = \emptyset \qquad A \cup \emptyset = A \tag{4.2}$$
$$(A \cap B) \cap C = A \cap (B \cap C) \qquad (A \cup B) \cup C = A \cup (B \cup C) \tag{4.3}$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \qquad A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \tag{4.4}$$
$$\mathbb{C}(A \cap B) = \mathbb{C}A \cup \mathbb{C}B \qquad \mathbb{C}(A \cup B) = \mathbb{C}A \cap \mathbb{C}B \tag{4.5}$$

Sie lassen sich allesamt beweisen, wenn man sich auf die Definitionen der Mengenoperationen bezieht. Im Zusammenhang mit der Aussagenlogik ergibt sich aber eine elegante Beweismethode, auf die dann verwiesen werden soll.

Hingewiesen sei jetzt auf die bemerkenswerte Dualität der Formeln. Das hat eine interessante Konsequenz: Wenn man in einer richtigen Gleichung für Mengen überall alle Terme des Typs, wie sie etwa im linken Teil der Liste vorkommen, durch die entsprechenden Terme aus dem rechten Teil ersetzt, erhält man wieder ein richtige Gleichung. Gleiches gilt, wenn man die Ersetzung von rechts nach links vornimmt.

Ferner sei hervorgehoben, dass die Gleichungen Eigenschaften darstellen, wie sie in analoger Weise beim Rechnen mit Zahlen benutzt werden.

In Zeile (1) haben wir die Kommutativität, d.h. die Vertauschbarkeit der Operanden.

Die leere Menge (Zeile (2)) spielt bezüglich Durchschnitt bzw. Vereinigung von Mengen dieselbe Rolle wie die Null bezüglich Multiplikation bzw. Addition von Zahlen.

In Zeile (3) haben wir die Assoziativität, die es gestattet, von jeder endlichen Menge von Mengen die Vereinigung oder den Durchschnitt zu bilden, ohne Klammern benutzen zu müssen:

$$\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \ldots \cup A_n$$

$$\bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap \ldots \cap A_n$$

4.1.3 Die Potenzmenge einer Menge

Zu jeder Menge A kann man die Menge aller ihrer Teilmengen bilden:

$$\mathfrak{P}(A) := \{M \mid M \subseteq A\}$$

Für $A = \{a, b, c\}$ ist zum Beispiel

$$\mathfrak{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Wenn man bei einer endlichen Menge A mit $\#A$ die Anzahl ihrer Elemente bezeichnet, so gilt immer $\#\mathfrak{P}(A) = 2^{\#A}$. Das hat zu der Bezeichnung $\mathfrak{P}(A) = 2^A$ geführt, die auch verwendet wird, wenn A unendlich viele Elemente enthält. In diesem Falle kann man mit dem Begriff der Kardinalzahl einer Menge ganz analoge Gleichungen gewinnen.

4.1.4 Paare und das kartesische Produkt zweier Mengen

Dem polnischen Mathematiker **Tarski** verdanken wir die folgende geistreiche Definition: zu zwei Elementen a, b beliebiger Mengen wird

$$(a, b) := \{\{a\}, \{a, b\}\}$$

das aus a und b gebildete **Elementpaar** oder einfach **Paar** genannt. a und b werden die erste bzw. die zweite **Komponente** des Paares genannt.

Folgerung 4.1.1 *Zwei Paare sind gleich genau dann, wenn ihre Komponenten paarweise gleich sind:*

$$(a, b) = (c, d) \text{ gdw } a = b \text{ und } c = d$$

Bemerkung: Die Floskel "Aussage1 genau dann, wenn Aussage2" wird immer dann verwendet, wenn sich aus Aussage1 die Aussage2 herleiten lässt und umgekehrt. Der folgende Beweis muss demnach aus zwei Teilen bestehen. Wir werden dafür auch häufig das Kürzel "gdw" einsetzen.

Beweis:

1. Der Beweis "von rechts nach links" ist trivial. Die Mengen $\{\{a\},\{a,b\}\}$ und $\{\{c\},\{c,d\}\}$ stimmen Element für Element überein, wenn $a = b$ und $c = d$ ist

2. Jetzt wird

$$(a,b) = \{\{a\},\{a,b\}\} = \{\{c\},\{c,d\}\} = (c,d)$$

vorausgesetzt. Zu zeigen ist $a = b$ und $c = d$. Das erfordert eine Fallunterscheidung.

(a) Zunächst sei $\{a\} = \{a,b\}$. Nach der Definition der Gleichheit von Mengen ist dann $\{a,b\} \subseteq \{a\}$, also $a = b$ und demnach

$$(a,b) = (a,a) = \{\{a\}\} = \{\{c\},\{c,d\}\}.$$

Wegen

$$\{\{c\},\{c,d\}\} \subseteq \{\{a\}\}$$

folgt $\{c\} = \{c,d\} = \{a\}$, mithin $a = b = c = d$

(b) Jetzt wird $\{a\} \neq \{a,b\}$ angenommen. Wegen $\{a\} \subseteq \{a,b\}$ muss dann, man vergleiche die Definition für die Gleichheit von Mengen, $\{a,b\} \not\subseteq \{a\}$ gelten, was zu $a \neq b$ führt. Dann ist aber auch $\{a,b\} \neq \{c\}$. Wegen $\{a,b\} \subseteq \{\{c\},\{c,d\}\}$ muss dann $\{a,b\} = \{c,d\}$ sein. Da nun $\{a,b\}$ zwei verschiedene Elemente enthält, kann nicht $c = d$ sein. Dann ist aber zwingend $\{a\} = \{c\}$, folglich $a = c$ und schließlich $b = d$.

Damit ist der Beweis erbracht. Dieses Konstruktion ist deshalb erstaunlich, weil sie zu einer Reihenfolge der Elemente führt. Bei Mengen ist es völlig gleichgültig, wie die Elemente angeordnet werden:

$$\{\{a\},\{a,b\}\} = \{\{a\},\{b,a\}\} = \{\{a,b\},\{a\}\} = \{\{b,a\},\{a\}\}$$

Beim Vergleich der Elemente zweier Mengen, die nach dieser Vorschrift gebildet sind, haben wir aber eine Zuordnung zu beachten, wie sie in der Folgerung zum Ausdruck kommt.

Der Begriff der Paares von Elementen gibt uns die Möglichkeit, Elemente unterschiedlicher Mengen miteinander zu verknüpfen:

Definition 4.1.1 *Das kartesische Produkt zweier Mengen ist die Menge aller Paare, die sich aus den Elementen beider Mengen bilden lassen:*

$$A \times B := \{(a,b) \mid a \in A \text{ und } b \in B\}$$

Das Attribut "kartesisch" erinnert an den französischen Mathematiker, Philosophen und Militär **René Descartes** (1596-1650). Er hat u.a. die analytische Geometrie begründet und dazu Koordinatensysteme eingeführt. Damit kann man die Punkte einer Ebene durch Paare reeller Zahlen, also durch die Elemente der Menge $\mathbb{R} \times \mathbb{R}$ ausdrücken.

4.2 Relationen und Funktionen

Eine Relation R stellt Elemente einer Menge A mit Elementen einer Menge B in Beziehung. Eine Relation manifestiert sich als eine Menge von Paaren beider Mengen:

Definition 4.2.1 *Eine Relation R von A nach B ist eine Teilmenge des kartesischen Produktes der Mengen A und B:*

$$R \subseteq A \times B$$

Dieser Begriff ist einerseits sehr simpel und elementar und besitzt andererseits einige interessante Eigenschaften, so dass er vielseitig anwendbar ist und damit ein universelles Instrument für die Beschreibung ganz unterschiedlicher Sachverhalte, und zwar nicht nur in der Mathematik darstellt.

Wir treffen Vereinbarungen über die Schreibweise und zwei zusätzliche Begriffe:

- Statt $R \subseteq A \times B$ wird häufig $R : A \to B$ verwendet. Damit wird die Reihenfolge bei der Bildung der Paare ausgedrückt.
- $(a, b) \in R$ wird mit aRb identifiziert.
- Im allgemeinen bilden die Komponenten der Paare einer Relation echte Teilmengen von A bzw. von B. Um die Mengen der tatsächlich verwendeten Elemente zu bezeichnen, vereinbaren wir:

$$dom\ R := \{a \in A \mid \text{es gibt ein } b \in B \text{ mit } aRb\}$$
$$im\ R := \{b \in B \mid \text{es gibt ein } a \in A \text{ mit } aRb\}$$

(dom und im sind Abkürzungen der englischen Wörter *domain* und *image*)

Die Inklusion von Mengen ist ein Beispiel für eine Relation, wo $A = B$ ist, eine "Relation auf einer Menge": Wenn X eine beliebige Menge ist und 2^X deren Potenzmenge, so ist

$$include := \{(M, N) \in 2^X \times 2^X \mid M \subseteq N\}.$$

Die zweite Vereinbarung bedeutet hier $include = \subseteq$ oder

$$M \subseteq N \text{ gdw } (M, N) \in include = \subseteq$$

Eine besonders einfache und sehr wichtige Relation ist die Identität auf einer Menge:

$$\mathbb{1}_A := \{(a, a) \mid a \in A\}$$

Datenbanken sind relational aufgebaut. Ein sehr simples Beispiel soll zur Illustration dienen. Eine Universität, wenn sie gut verwaltet ist, muss jederzeit über den Bestand an Möbeln Auskunft geben können. Sei M die Menge aller Möbel und Q die Menge der Räume. Dann muss aus der entsprechenden Datei jederzeit ersichtlich sein, welches Möbelstück sich in welchem Raum befindet. Das ist eine Relation, die mit IST_IN bezeichnet werden soll. Dass der Stuhl $st101$ zur Zeit gerade im Hörsaal $H3.2$ steht, sieht in unserer Symbolik dann so aus:

$$st101 \text{ IST\_IN } H3.2 \text{ oder } (st101, H3.2) \in \text{IST\_IN} \subseteq M \times Q$$

Mit diesem Beispiel kann man die folgenden Definitionen anschaulich machen.

Zu jeder Relation gehört deren inverse:

Definition 4.2.2 *Zu einer Relation $R \in A \times B$ ist*

$$R^{-1} := \{(b, a) \in B \times A \mid (a, b) \in R\}$$

In unserem Beispiel ist $(H3.2, st101) \in ist\_in^{-1}$. Nennen wir

$$\text{ENTHÄLT} := ist\_in^{-1},$$

so haben wir

$$H3.2 \text{ ENTHÄLT } st101 \text{ weil } st101 \text{ IST\_IN } H3.2$$

Offenbar ist $\text{ENTHÄLT}^{-1} = \text{IST\_IN}$ oder, im allgemeinen Fall, $(R^{-1})^{-1} = R$.

Relationen kann man miteinander verknüpfen. Unser Beispiel ergänzen wir durch durch eine weitere Relation. Wir nehmen an, dass gewisse Räume der Universität jeweils gewissen Instituten zugeordnet sind. Wenn I die Menge der Institute ist, so stellt das eine Relation ZUGEORDNET $\in Q \times I$ dar. Falls nun der Hörsaal $H3.2$ vom Institut für Informatik, abgekürzt durch $IfIn$ verwaltet wird, steht der Stuhl $st101$ dem Institut für Informatik zur Verfügung, weil er ja in einem Raum steht, für den dieses Institut zuständig ist. Das ist eine neuer Relation GENUTZT_VON, die sich aus den beiden anderen zwangsläufig ergibt. In Zeichen:

Weil

$$(st101 \text{ IST\_IN } H3.2) \text{ und } (H3.2 \text{ ZUGEORDNET } IfIn),$$

ist

$$st101 \text{ GENUTZT\_VON } IfIn$$

Definition 4.2.3 *Zu zwei Relationen*

$$R_1 : A \to B \text{ und } R_2 : B \to C \text{ ist } R = R_2 \circ R_1 : A \to C$$

definiert durch

$$R_2 \circ R_1 := \{(a, c) \in A \times C \mid \text{ es gibt ein } b \in B \text{ so, dass } (a, b) \in R_1 \text{ und } (b, c) \in R_2\}$$

Folgerung 4.2.1 *Seien $R_1 : A \to B$, $R_2 : B \to C$ und $R_3 : C \to D$ Relationen. Es gilt*

$$(R_3 \circ R_2) \circ R_1 = R_3 \circ (R_2 \circ R_1) \tag{4.6}$$

$$\mathbb{1}_B \circ f = f \circ \mathbb{1}_A = f \tag{4.7}$$

$$R_1^{-1} \circ R_1 = \mathbb{1}_{(dom\ R_1)}; \quad R_1 \circ R_1^{-1} = \mathbb{1}_{(im\ R_1)} \tag{4.8}$$

$$(R_2 \circ R_1)^{-1} = R_1^{-1} \circ R_2^{-1} \tag{4.9}$$

Beweis: Wir beschränken uns auf den Nachweis der letzten Gleichung. Die anderen lassen sich ebenso einfach beweisen. Man muss lediglich die Definitionen ganz konsequent anwenden.

$(c, a) \in (R_2 \circ R_1)^{-1}$ gdw $(a, c) \in R_2 \circ R_1$
gdw ein $b \in B$ existiert mit $(a, b) \in R_1$ und $(b, c) \in R_2$
gdw ein $b \in B$ existiert mit $(b, a) \in R_1^{-1}$ und $(c, b) \in R_2^{-1}$
gdw $(c, a) \in R_1^{-1} \circ R_2^{-1}$

Äquivalenzrelationen

Definition 4.2.4 *Eine Relation $R : A \to A$ auf einer Menge A heißt **Äquivalenzrelation**, wenn sie folgende Eigenschaften hat:*

- *Für alle $a \in A$ ist $(a, a) \in R$ bzw. aRa.*
- *Aus aRb folgt stets bRa.*
- *Wenn aRb und bRc, so ist auch aRc.*

Die drei Eigenschaften werden in dieser Reihenfolge **Reflexivität**, **Symmetrie** und **Transitivität** genannt.

Die simpelste Äquivalenzrelation ist die Gleichheit. Die bekannteste ist wohl die folgende:

$$mod\_m = \{(u, v) \in \mathbb{Z} \times \mathbb{Z} \mid \text{ es gibt ein } k \in \mathbb{Z} \text{ derart, dass } u - v = k \cdot m\}^1$$

Man sagt, u sei äquivalent *modulo* m zu v, wenn $u \; mod\_m \; v$ ist. Es stellt sich heraus, dass zwei Zahlen äquivalent modulo m sind genau dann, wenn sie bei Division durch m den selben Rest liefern.

Darin liegt ein Ordnungsprinzip, das Äquivalenzrelationen generell innewohnt. Mann kann eine Menge strukturieren, indem man äquivalente Element in Klassen zusammenfasst. In diesem Beispiel sind die Elemente einer Klasse gerade jenen, welche die Eigenschaft gleicher Reste haben Das ist eine Abstraktionsmethode, die sich vielfach bewährt.

Definition 4.2.5 *Sei R eine Äquivalenzrelation auf der Menge A. Die Menge*

$$[a]_R := \{b \in A \mid bRa\}$$

heißt ***Äquivalenzklasse*** *der Relation R. Das Element a wird* ***Repräsentant*** *der Klasse genannt.*

Definition 4.2.6 *Eine Teilmenge $\mathfrak{U} \subset 2^A$ der Potenzmenge einer Menge $A \neq \emptyset$ soll* ***disjunkte Überdeckung*** *von A genannt werden, wenn die folgenden beiden Bedingungen erfüllt sind:*

1. *A ist in der Vereinigung der Elemente von $\mathfrak{U}$ enthalten: $A \subseteq \bigcup_{U \in \mathfrak{U}} U$*
2. *Für alle $U_1, U_2 \in \mathfrak{U}$ ist entweder $U_1 = U_2$ oder $U_1 \cap U_2 = \emptyset$*

So wie $\mathfrak{U}$ definiert ist, gilt $\bigcup_{U \in \mathfrak{U}} U \subseteq A$. Aus der ersten Bedingung folgt also $\bigcup_{U \in \mathfrak{U}} U = A$

Satz 4.2.1

- *Die Äquivalenzklassen einer Äquivalenzrelation auf einer Menge A bilden eine disjunkte Überdeckung von A*
- *Eine disjunkte Überdeckung $\mathfrak{U}$ einer Menge A generiert eine Äquivalenzrelation auf A*

Beweis:

1. Sei R eine Äquivalenzrelation auf A und $\mathfrak{U} := \{[a]_R \mid a \in A\}$. Ist $a \in A$ beliebig gewählt, so ist wegen der Reflexivität der Relation $a \in [a]_R \in \mathfrak{U}$, also $A \subseteq \bigcup_{a \in A} [a]_R$.

   Um die zweite Eigenschaft einer disjunkten Überdeckung zu zeigen, wird angenommen, dass zwei Klassen nicht disjunkt sind. Daraus muss sich ihre Gleichheit ergeben. Sei also $c \in [a]_R \cap [b]_R$. Falls nun $x \in [a]_R$ beliebig gewählt ist, hat man

   $$xRa, \; cRa \text{ bzw. wegen der Symmetrie } aRc \text{ und } cRb.$$

   Wegen der Transitivität folgt xRb, mithin $x \in [b]_R$ und weiter $[a]_R \subseteq [b]_R$. In derselben Weise bekommt man $[b]_R \subseteq [a]_R$. Beide Klassen sind gleich.

2. Ist $\mathfrak{U}$ eine disjunkte Überdeckung von A, setzt man

$$R = \{(a,b) \in A \times A \mid \text{ es gibt ein } U \in \mathfrak{U} \text{ mit } a \in U \text{ und } b \in U\}$$

Das ist offensichtlich eine Äquivalenzrelation.

4.2.1 Funktionen

Funktionen sind solche Relationen $R : A \to B$, die jedes Element $a \in dom\ R$ mit einem einziges Element $b \in B$ zu einem Paar verknüpfen.

Definition 4.2.7 *Eine Relation $f : A \to B$ ist eine **Funktion** oder **Abbildung** per Definition genau dann, wenn aus $(a,b_1) \in f$ und $(a,b_2) \in f$ folgt, dass $b_1 = b_2$ ist.*

Vereinbarungen und Folgerungen:

- Ist $dom\ f = A$, sprechen wir von einer **totalen** Funktion oder einfach von einer Funktion. Besonders in der Informatik sind solche Funktionen wichtig, die nicht überall definiert ist, wenn also $dom\ f \neq A$ ist. Dann handelt es sich um eine **partielle Funktion**, für die wir das Symbol $f : A \overset{\supset}{\longrightarrow} B$ wählen.

- Da es bei Funktionen zu jedem $a \in dom\ f$ genau ein Element $b \in B$ mit $(a,b) \in f$ gibt, wird dafür eine besondere Bezeichnung eingeführt: $f(a) := b$. Wir werden auch sagen, dass a ein **Argumentwert** der Funktion sei, welcher auf den **Funktions-** oder **Bildwert** $f(a)$ abgebildet wird.

  Wenn $f : A \overset{\supset}{\longrightarrow} B$ eine partielle Funktion ist und $a \notin dom\ f$, dann sagt man, f sei für a nicht definiert und schreibt $f(a) = \perp$

- Symbolik:

$$\begin{aligned} f : A &\overset{\supset}{\longrightarrow} B \qquad (\text{bzw.} f : A \to B \text{ wenn } f \text{ total ist.}) \\ a &\mapsto f(a) \end{aligned}$$

- Wenn die Funktionen $f : A \to B$ und $g : B \to C$ gegeben sind, so ist

$$\begin{aligned} g \circ f : A &\to C \\ a &\mapsto c = (g \circ f)(a) = f(g(a)) \end{aligned}$$

  denn: $(a,c) \in g \circ f$ genau dann, wenn es ein $b \in B$ gibt mit $(a,b) \in f$ und $(b,c) \in g$. Das ist die für Relationen benutzte Symbolik. Für Funktionen sieht das so aus:

  $c = (g \circ f)(a)$ genau dann, wenn $b = f(a)$ und $c = g(b)$, also $c = g(f(a))$. Dass $b = f(a)$ existiert, ist bei Funktionen evident.

- Bei partiellen Funktionen $f : A \stackrel{\supset}{\longrightarrow} B$ und $g : B \stackrel{\supset}{\longrightarrow} C$ kann es bei der Verknüpfung zu Einschränkungen des Definitionsbereiches kommen: $(g \circ f)(a) = g(f(a))$ ist nur dann definiert, wenn $a \in dom\ f$ und $f(a) \in im\ g$. Ist zufällig $im\ f \subseteq dom\ g$, so gilt $dom\ f = dom\ (g \circ f)$

Beispiele für Funktionen:

- Zu einer festen Zeit ist jedem Punkt des uns umgebenden Raumes genau eine Temperatur zugeordnet.
- IST_IN, ZUGEORDNET und GENUTZT_VON im Beispiel oben sind Funktionen, wogegen ENTHÄLT keine ist.
- Die Vergabe polizeilicher Kennzeichen für Fahrzeuge ist (oder sollte es jedenfalls sein) eine Funktion.
- Das Gewicht eines Körpers ist eine Funktion, wenn man sich an einem definierten Ort der Erde aufhält.

Die Liste kann man beliebig erweitern und macht deutlich, wie nützlich der Begriff ist.

Funktionen können noch mit besonderen Eigenschaften ausgestattet sein.

Definition 4.2.8

- *Eine Funktion $f : A \to B$ heißt **surjektiv**, wenn es zu jedem $b \in B$ ein $a \in A$ gibt derart, dass $b = f(a)$ ist.*
- *Eine Funktion $f : A \to B$ heißt **injektiv**, wenn aus $f(a_1) = f(a_2)$ die Gleichheit $a_1 = a_2$ folgt.*
- *Eine Funktion heißt **bijektiv**, wenn sie injektiv und surjektiv ist.*

Surjektivität bedeutet also, dass jedes Element der Bildmenge auch tatsächlich einen Funktionswert darstellt. Man hätte diese Eigenschaft auch durch die Gleichung $im\ f = B$ ausdrücken können. Und injektiv bedeutet, dass zu jedem Funktionswert ein und nur ein Argumentwert existiert. Mit anderen Worten, f^{-1} ist ebenfalls eine Funktion. Damit ist der folgende Satz fast bewiesen:

Satz 4.2.2 *Eine Funktion $f : A \to B$ ist bijektiv genau dann, wenn es eine Funktion $g : B \to A$ gibt mit*

$$g \circ f = \mathbb{1}_A \text{ und } f \circ g = \mathbb{1}_B \tag{4.10}$$

Daraus folgt sofort, dass g ebenfalls bijektiv ist.

Beweis des Satzes:

1. Zu der bijektiven Funktion f wird die Relation

$$g := f^{-1} = \{(b, a) \in B \times A \mid b = f(a)\}$$

ins Auge gefasst. Da f surjektiv ist, gilt $dom\ g = B$.

Ist $(b, a_1) \in g$ und $(b, a_2) \in g$, also $b = f(a_1) = f(a_2)$, folgt aus der Injektivität von f, dass $a_1 = a_2$ ist.

Damit ist f eine totale Funktion. Die Gleichungen 4.10 ergeben sich aus 4.8

2. Jetzt werden die Gleichungen 4.10 vorausgesetzt. Wenn $b \in B$ beliebig gewählt ist, folgt $f(g(b)) = b$. f ist surjektiv.

Aus $f(a_1) = f(a_2)$ ergibt sich $a_1 = g(f(a_1)) = g(f(a_2)) = a_2$. f ist auch injektiv.

Das war zu zeigen.

4.2.2 Das kartesische Produkt beliebig vieler Mengen - Operationen

Der Begriff des Elementpaares hat er ermöglicht, das kartesische Produkt zweier Mengen und damit Relationen und Funktionen zu definieren. Man kann auf diese Weise auch Tupel definieren, die eine beliebige endliche Anzahl von Elementen unterschiedlicher Mengen enthalten. Allerdings kann man auf diesem Wege nicht, wie das vergleichsweise bei Summen oder Produkten von Zahlen möglich ist, das Produkt unendlich vieler Mengen bilden. Mit Hilfe des Begriffs der Funktion kann man eine begrifflich etwas komplizierte, dafür aber sehr elegante und verallgemeinerungsfähige Konstruktion angeben. Zur Motivation ein Beispiel:

Sei $A = \{u, v, w\}$ und $B = \{x, y\}$. Gemäß unserer Definition ist

$$A \times B = \{(u, x), (u, y), (v, x), (v, y), (w, x), (w, y)\}$$

Außerdem sei

$$A \otimes B := \{f : \{1, 2\} \to (A \sqcup B) \mid f(1) \in A \text{ und } f(2) \in B\}$$

und $f_{(a,b)}$ diejenige Funktion aus $A \otimes B$, für welche $f(1) = a \in A$ und $f(2) = b \in B$ ist. Die in $A \otimes B$ enthaltenen Funktionen sind in der folgenden Tabelle aufgelistet.

	$f_{(u,x)}$	$f_{(u,y)}$	$f_{(v,x)}$	$f_{(v,y)}$	$f_{(w,x}$	$f_{(w,y)}$
1	u	u	v	v	w	w
2	x	y	x	y	x	y

Die beiden Mengen $A \times B$ und $A \otimes B$ sind offenbar sehr ähnlich. Sie lassen sich bijektiv aufeinander abbilden. Man kann die Elemente der einen Menge

mit denen der anderen kodieren. Da bei allen Elementen von $A \otimes B$ der Argumentbereich derselbe ist, reicht es aus, zum Beispiel die Funktion $f_{(u,x)}$ durch das formale Symbol (u, x) zu ersetzen. Das ist zwar inhaltlich etwas anderes als das mit Mengen definierte Paar (u, x). Wir nehmen uns aber die Freiheit, beide miteinander zu identifizieren. Außerdem werden wir durchgehend die in der folgenden Definition angegebene Schreibweise für das kartesische Produkt von Mengen verwenden.

Die folgenden Schreibweise wird als Abkürzung benutzt:

$$[1..n] := \{1, 2, 3, \ldots, n\}$$

Definition 4.2.9 *Das kartesische Produkt der Mengen A_i; $1 \leq i \leq n$ ist die Menge aller Funktion von $[1..n]$ auf die freie Vereinigung der Mengen A_i mit der Bedingung, dass i auf A_i abgebildet wird für $1 \leq i \leq n$:*

$$\begin{aligned}\prod_{i=1}^{n} A_i &= A_1 \times A_2 \times \ldots \times A_n \\ &:= \{f : [1..n] \to \bigsqcup_{i=1}^{n} A_i \mid f(i) \in A_i \ \ (1 \leq i \leq n)\}\end{aligned}$$

Wie im Beispiel werden wir

$$(a_1, a_2, \ldots, a_n) \in A_1 \times A_2 \times \ldots \times A_n$$

schreiben, wenn die Funktion

$$\begin{aligned}f : [1..n] &\to \bigsqcup_{i=1}^{n} A_i \\ i &\mapsto f(i) = a_i \in A_i\end{aligned}$$

gemeint ist.

Jetzt können wir sagen, was eine Operation ist:

Definition 4.2.10 *Seien $A_1, A_2, \ldots, A_n, A$ beliebige Mengen. Eine Abbildung*

$$f : A_1 \times A_2 \times \ldots \times A_n \to A$$

ist eine ***n-äre Operation***

Wir werden es meistens mit binären Operationen

$$f : A_1 \times A_2 \to A$$

zu tun haben. Außerdem wird häufig $A_1 = A_2$, oft sogar $A_1 = A_2 = A$ sein. Man denke beispielsweise an die jedermann bekannten arithmetischen Operation

in allen möglichen Zahlbereichen. Von daher ist auch bekannt, dass bei binären Operationen im allgemeinen die so genannte Infixschreibweise verwendet wird, wie sie oben bei Relationen schon eingeführt wurde. Nach unserer Definition ist etwa für die Addition im Bereich der natürlichen Zahlen wie folgt zu schreiben:

$$\begin{aligned} + : \mathbb{N} \times \mathbb{N} &\to \mathbb{N} \\ (p, q) &\mapsto +(p, q) \end{aligned}$$

Üblich ist aber natürlich

$$(p, q) \mapsto p + q$$

Hinweisen wollen wir auf Operationen, die wir schon eingeführt haben. Wenn zum Beispiel $Rel_{(M,N)}$ die Menge aller Relationen $R : M \to N$ ist, so ist die Verknüpfung

$$\circ : Rel_{(A,B)} \times Rel_{(B,C)} \to Rel_{(A,C)}$$

ein Beispiel einer binären Operation, wo alle beteiligten Mengen verschieden sind.

Die Inklusion von Mengen auf einer Potenzmenge kann man ebenfalls als Operation interpretieren. Sei $A \neq \emptyset$ eine beliebige Menge:

$$\begin{aligned} \subseteq &: 2^A \times 2^A \to \{wahr, falsch\} \\ (M, N) \mapsto M \subseteq N &= \begin{cases} wahr, & \text{wenn für alle } a \in M \text{ gilt: } a \in N \\ falsch & \text{sonst} \end{cases} \end{aligned}$$

In Analogie zum kartesischen Produkt von Mengen verwendet man eine aus verschiedenen Funktionen kombinierte komplexe Funktion:

Definition 4.2.11 *Zu den Funktionen $\varphi_i : A_i \to B_i$; $1 \leq i \leq n$ ist*

$$\begin{aligned} \varphi_1 \times \varphi_2 \times \ldots \times \varphi_n : A_1 \times A_2 \times \ldots \times A_n &\to B_1 \times B_2 \times \ldots \times B_n \\ (a_1, a_2, \ldots, a_n) &\mapsto (\varphi_1(a_1), \varphi_2(a_2), \ldots \varphi_n(a_n)) \end{aligned}$$

4.3 Homomorphie, Isomorphie

Die beiden Worte sind aus der griechischen Sprache entlehnt: *homos* = vergleichbar, von gleicher Qualität, *isos* = gleich, nicht unterscheidbar, *morphe* = Gestalt, Form.

Wir beginnen mit einem Beispiel. In der Menge der ganzen Zahlen ist die Multiplikation eine Operation $* : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. Dieses Paar $(\mathbb{Z}, *)$ wird Algebra genannt, genauer, weil nur eine Menge beteiligt ist, eine einsortige Algebra. Eine

zweite Algebra wird durch die rationalen Zahlen $\mathbb{P}$ und die in diesem Zahlbereich übliche Multiplikation gebildet. Aus langer Gewohnheit ist man sich im allgemeinen gar nicht mehr bewusst, dass es sich um zwei völlig unterschiedliche Mengen und um zwei ganz verschiedene Operationen handelt. Man darf diese Unterschiede auch vergessen, weil es möglich ist, die eine Algebra in die andere "homomorph einzubetten". Um das zu verstehen, sollen die Unterschiede beider Algebren bewusst hervorgehoben werden. Die rationalen Zahlen, die normalerweise als Brüche ganzer Zahlen geschrieben werden können, wobei der Nenner verschieden von Null sein muss, notieren wir jetzt als Paare:

$$\mathbb{P} = \{(p, q) \in \mathbb{Z} \times \mathbb{Z} \mid q \neq 0\} = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$$

Die Multiplikation wird mit dem Zeichen $\otimes$ dargestellt. Die Algebra der rationalen Zahlen mit der Multiplikation ist demnach $(\mathbb{P}, \otimes)$, wobei bekanntlich

$$\begin{aligned} \otimes &: \mathbb{P} \times \mathbb{P} \to \mathbb{P} \\ (r_1, r_2) = ((p_1, q_1), (p_2, q_2)) &\mapsto r_1 \otimes r_2 = (p_1 * p_2, q_1 * q_2) \end{aligned}$$

gilt.

Um die Angelegenheit etwas zu vereinfachen, betrachten wir nur solche Zahlen $r = (p, q)$, wo der größte gemeinsame Teiler von p und q eins ist. Nun werden die ganzen Zahlen in die rationalen Zahlen abgebildet:

$$\begin{aligned} \varphi &: \mathbb{Z} \to \mathbb{P} \\ p &\mapsto (p, 1) \end{aligned}$$

Es stellt sich heraus, dass

$$\varphi(p * q) = (p * q, 1) = (p, 1) \otimes (q, 1) = \varphi(p) \otimes \varphi(q) \tag{4.11}$$

ist. Mit anderen Worten: es ist gleichgültig, ob man erst das Produkt in $\mathbb{Z}$ errechnet und das Ergebnis nach $\mathbb{P}$ projiziert oder ob man erst die beiden Faktoren nach $\mathbb{P}$ abbildet und dann dort multipliziert. Das Ergebnis ist dasselbe.

Noch anders formuliert: die ganzen Zahlen verhalten sich bezüglich ihrer Multiplikationsregel ganz analog wie diejenigen rationalen Zahlen, die den Nenner eins haben. Die Struktur der Algebra $(\mathbb{Z}, *)$ findet sich in gleicher Gestalt, also homomorph in der Algebra $(\mathbb{P}, \otimes)$ wieder. Man drückt diesen Sachverhalt auch sehr suggestiv grafisch wie folgt aus:

$$\begin{array}{ccc} \mathbb{Z} \times \mathbb{Z} & \xrightarrow{\varphi \times \varphi} & \mathbb{P} \times \mathbb{P} \\ * \downarrow & \circlearrowright & \downarrow \otimes \\ \mathbb{Z} & \xrightarrow{\varphi \times \varphi} & \mathbb{P} \end{array}$$

Der Sinn der Gleichung 4.11 spiegelt sich in diesem Diagramm in der Weise wider, dass es gleichgültig ist, in welcher Reihenfolge man den Pfeilen von $\mathbb{Z} \times \mathbb{Z}$

nach $\mathbb{Q}$ folgt, man kommt zum selben Ergebnis. Man sagt, dass Diagramm sei kommutativ und bezeichnet das durch den kreisförmigen Pfeil.

Nimmt man statt φ die bijektive Abbildung

$$\begin{aligned}\tilde{\varphi} : \mathbb{Z} &\to \tilde{\mathbb{P}} = \{(p,q) \in \mathbb{P} \mid q = 1\} \\ p &\mapsto (p,1),\end{aligned}$$

erhält man Homomorphie in beiden Richtungen. Die Algebren $(\mathbb{Z}, *)$ und $(\tilde{\mathbb{P}}, \otimes)$ unterscheiden sich lediglich durch die Bezeichnung ihrer Elemente und der Operationen, nicht mehr in ihrem strukturellen Verhalten. Sie sind isomorph und können gegeneinander ausgetauscht werden.

Die folgende Definition verallgemeinert dieses Beispiel. Sie wird formuliert für mehrsortige Algebren mit partiellen Operationen, wie sie bei Automaten auftreten. Selbstverständlich ist das Beispiel als spezieller Fall enthalten.

Definition 4.3.1 *Seien*

$$\Lambda = (A, A_1, A_2, \ldots, A_n, f); \qquad f : A_1 \times A_2 \times \ldots \times A_n \xrightarrow{\supset} A$$
$$\Gamma = (B, B_1, B_2, \ldots, B_n, g); \qquad g : B_1 \times B_2 \times \ldots \times B_n \xrightarrow{\supset} B$$

zwei Algebren und seien $\Phi_{(\Lambda,\Gamma)} = \{\varphi, \varphi_1, \varphi_2, \ldots, \varphi_n\}$ *Abbildungen*

$$\begin{aligned}\varphi &: A \to B \\ \varphi_i &: A_i \to B_i\,; \quad (1 \le i \le n)\end{aligned}$$

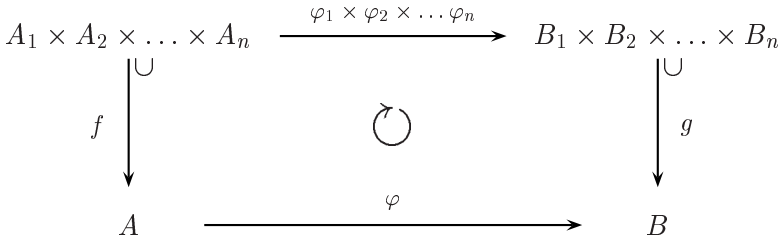
$\Phi_{(\Lambda,\Gamma)}$ *heißt* ***Homomorphismus*** *von* Λ *nach* Γ*, wenn folgendes gilt:*

1. *Ist* $(a_1, a_2, \ldots, a_n) \in$ *dom* f *so auch* $(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \in$ *dom* g *und*

2. $$\varphi(f(a_1, a_2, \ldots, a_n)) = g(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \tag{4.12}$$

Vereinbarung: Statt $\Phi_{(\Lambda,\Gamma)}$ werden wir auch die Bezeichnung $\Phi : \Lambda \to \Gamma$ verwenden. Wenn $\Lambda = \Gamma$ ist, schreiben wir Φ_Λ.

Das kommutative Diagramm bekommt jetzt die folgende Gestalt:



Folgerung 4.3.1 $\mathbb{I}_\Lambda = \{\mathbb{1}_A, \mathbb{1}_{A_1}, \mathbb{1}_{A_2}, \ldots, \mathbb{1}_{A_n}\}$ ist eine Homomorphismus für jede Operation $f : A_1 \times A_2 \times \ldots \times A_n \xrightarrow{\supset} A$.

Beweis:

$$\mathbb{1}_A(f(a_1, a_2, \ldots, a_n)) = f(a_1, a_2, \ldots, a_n) = f(\mathbb{1}_{A_1}(a_1), \mathbb{1}_{A_2}(a_2), \ldots, \mathbb{1}_{A_n}(a_n))$$

für beliebige Tupel $(a_1, a_2, \ldots, a_n) \in dom\ f$

Satz 4.3.1 *Homomorphie ist transitiv. Genauer, sind*

$$\Lambda = (A, A_1, A_2, \ldots, A_n, f); \qquad f : A_1 \times A_2 \times \ldots \times A_n \xrightarrow{\supset} A$$
$$\Gamma = (B, B_1, B_2, \ldots, B_n, g); \qquad g : B_1 \times B_2 \times \ldots \times B_n \xrightarrow{\supset} B$$
$$\Omega = (C, C_1, C_2, \ldots, C_n, g); \qquad h : C_1 \times C_2 \times \ldots \times C_n \xrightarrow{\supset} C$$

Algebren und

$$\Phi_{(\Lambda,\Gamma)} = \{\varphi, \varphi_1, \varphi_2, \ldots, \varphi_n\}\ ; \qquad \varphi : A \to B\ ; \varphi_i : A_i \to B_i\ ;$$
$$\Psi_{(\Gamma,\Omega)} = \{\psi, \psi_1, \psi_2, \ldots, \psi_n\}\ ; \qquad \psi : B \to C\ ; \psi_i : B_i \to C_i\ ; \quad (1 \le i \le n)$$

Homomorphismen, so ist

$$\Theta_{\Lambda,\Omega} = \Psi_{(\Gamma,\Omega)} \circ \Phi_{(\Lambda,\Gamma)} := \{\psi \circ \varphi, \psi_1 \circ \varphi_1, \psi_2 \circ \varphi_2, \ldots, \psi_n \circ \varphi_n\}$$

ebenfalls Homomorphismus.

Beweis: Sei $(a_1, a_2, \ldots, a_n) \in dom\ f$. Da $\Phi_{(\Lambda,\Gamma)}$ Homomorphismus ist, folgt $(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \in dom\ g$. Da auch $\Psi_{(\Gamma,\Omega)}$ diese Eigenschaft hat, ist schließlich

$$(\psi_1(\varphi_1(a_1)), \psi_2(\varphi_2(a_2)), \ldots \psi_n(\varphi_n(a_n))) \in dom\ h$$

Damit sind alle Terme in der folgenden Kette von Gleichungen definiert. Die einzelnen Gleichungen ergeben sich, wenn man die Gleichung 4.12 der Definition 4.3.1 sinngemäß anwendet.

$$\begin{aligned}(\psi \circ \varphi)(f(a_1, a_2, \ldots, a_n)) &= \psi(\varphi(f(a_1, a_2, \ldots, a_n)) \\ &= \psi(g(\varphi_2(a_2), \ldots, \varphi_n(a_n))) \\ &= h(\psi_1(\varphi_1(a_1)), \psi_2(\varphi_2(a_2)), \ldots, \psi_n(\varphi_n(a_n))) \\ &= h((\psi_1 \circ \varphi_1)(a_1), (\psi_2 \circ \varphi_2)(a_2), \ldots (\psi_n \circ \varphi_n)(a_n))\end{aligned}$$

Das war zu zeigen.

Es ist hilfreich, wenn man sich den Inhalt dieses Satzes in ein Diagramm der Art übersetzt, wie sie oben benutzt worden sind. Wir wollen auch auf die Analogien zwischen Funktionen und Homomorphismen aufmerksam machen, die sich hier und im weiteren zeigen.

Folgerung 4.3.2 *Seien* $\Phi_1 : \Lambda \to \Omega$, $\Phi_2 : \Omega \to \Delta$ *und* $\Phi_3 : \Delta \to \Pi$ *Homomorphismen. Dann gilt*

- $(\Phi_3 \circ \Phi_2) \circ \Phi_1 = \Phi_3 \circ (\Phi_2 \circ \Phi_1)$
- $\mathbb{I}_\Omega \circ \Phi_1 = \Phi_1 \circ \mathbb{I}_\Lambda$

Der Beweis ist einfach. Man bedient sich der entsprechenden Eigenschaften bei Funktionen (vergl. die Folgerung 4.2.1, Seite 146)

Bei Funktionen wurde die Eigenschaft "bijektiv" definiert und als notwendige und hinreichende Bedingung für die Existenz einer Umkehrfunktion erkannt. Jetzt gehen wir umgekehrt vor. Es wird definiert, was ein inverser Homomorphismus leisten soll und abgeleitet, dass die Bijektivität der beteiligten Funktionen eine wichtige Rolle spielt.

Definition 4.3.2 *Ein Homomorphismus* $\Phi_{(\Lambda,\Omega)}$ *heißt* ***Isomorphismus****, wenn es einen Homomorphismus* $\Psi_{(\Omega,\Lambda)}$ *derart gibt, dass*

$$\Psi_{(\Omega,\Lambda)} \circ \Phi_{(\Lambda,\Omega)} = \mathbb{I}_\Lambda \text{ und } \Phi_{(\Lambda,\Omega)} \circ \Psi_{(\Omega,\Lambda)} = \mathbb{I}_\Omega$$

ist.

Der Homomorphismus $\Psi_{(\Omega,\Lambda)}$ ist wohlbestimmt. Wenn $\tilde{\Psi}_{(\Omega,\Lambda)}$ diese Gleichungen erfüllt, ergibt sich

$$\begin{aligned}\tilde{\Psi}_{(\Omega,\Lambda)} &= \mathbb{I}_\Lambda \circ \tilde{\Psi}_{(\Omega,\Lambda)} = (\Psi_{(\Omega,\Lambda)} \circ \Phi_{(\Lambda,\Omega)}) \circ \tilde{\Psi}_{(\Omega,\Lambda)} \\ &= \Psi_{(\Omega,\Lambda)} \circ (\Phi_{(\Lambda,\Omega)} \circ \tilde{\Psi}_{(\Omega,\Lambda)}) = \Psi_{(\Omega,\Lambda)} \circ \mathbb{I}_\Omega \\ &= \Psi_{(\Omega,\Lambda)}\end{aligned}$$

Deshalb darf dieser Homomorphismus eine eigene Bezeichnung bekommen:

$$\Phi^{-1}_{(\Lambda,\Omega)} := \Psi_{(\Omega,\Lambda)}$$

Offenbar ist mit $\Psi_{(\Omega,\Lambda)}$ auch $\Phi^{-1}_{(\Lambda,\Omega)}$ Isomorphismus:

$$\Phi^{-1^{-1}}_{(\Lambda,\Omega)} = \Phi_{(\Lambda,\Omega)}$$

Satz 4.3.2 *Seien*

$$\begin{aligned}\Lambda &= (A, A_1, A_2, \ldots, A_n, f); \qquad f : A_1 \times A_2 \times \ldots \times A_n \xrightarrow{\supset} A \\ \Gamma &= (B, B_1, B_2, \ldots, B_n, g); \qquad g : B_1 \times B_2 \times \ldots \times B_n \xrightarrow{\supset} B\end{aligned}$$

Algebren und sei

$$\Phi_{(\Lambda,\Gamma)} = \{\varphi, \varphi_1, \varphi_2, \ldots, \varphi_n\}\,; \qquad \varphi : A \to B\,; \varphi_i : A_i \to B_i\,; \quad (1 \le i \le n)$$

ein Homomorphismus. $\Phi_{(\Lambda,\Gamma)}$ *ist Isomorphismus genau dann, wenn*

1. *φ und alle φ_i bijektiv sind und*
2. *$f(a_1, a_2, \ldots, a_n)$ definiert ist genau dann, wenn $g(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n))$ definiert ist.*

Beweis:

1. Wenn $\Phi_{(\Lambda,\Gamma)} = \{\varphi, \varphi_1, \varphi_2, \ldots, \varphi_n\}$ Isomorphismus ist, dann gibt es Funktionen ψ, ψ_i ; $(1 \leq i \leq n)$ mit

$$\begin{aligned} \varphi \circ \psi &= \mathbb{1}_B \; ; \quad \varphi_i \circ \psi_i = \mathbb{1}_{B_i} \\ \psi \circ \phi &= \mathbb{1}_A \; ; \quad \psi_i \circ \phi_i = \mathbb{1}_{A_i} \end{aligned}$$

   Nach Satz 4.2.2 auf Seite 149 ergibt sich die Bijektivität von φ und aller φ_i.

   Ist $(a_1, a_2, \ldots, a_n) \in dom\ f$, so auch $(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \in dom\ g$, weil $\Phi_{(\Lambda,\Gamma)}$ Homomorphismus ist.

   Umgekehrt, wenn $(\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \in dom\ g$, ist

$$(\psi_1(\varphi_1(a_1)), \psi_2(\varphi_2(a_2)), \ldots, \psi_n(\varphi_n(a_n))) = (a_1, a_2, \ldots, a_n) \in dom\ f,$$

   weil $\Psi_{(\Gamma,\Lambda)} = \Phi^{-1}_{(\Lambda,\Gamma)}$ Homomorphismus ist.

2. Jetzt wird angenommen, dass $\Phi_{(\Lambda,\Gamma)} = \{\varphi, \varphi_1, \varphi_2, \ldots, \varphi_n\}$ ein Homomorphismus ist und die Menge dieser Funktionen die oben formulierten Eigenschaften hat. Wegen der Bijektivität existiert dann eine andere Menge $\{\psi, \psi_1, \psi_2, \ldots, \psi_n\}$ von Funktionen so, dass es für alle i und jedes Element $b_i \in B_i$ ein $a_i \in A_i$ gibt mit $\varphi_i(a_i) = b_i$ und $\psi_i(b_i) = a_i$. Außerdem ist $\psi \circ \varphi = \mathbb{1}_A$. Ist nun $(b_1, b_2, \ldots, b_n) = (\varphi_1(a_1), \varphi_2(a_2), \ldots, \varphi_n(a_n)) \in dom\ g$, so auch $(a_1, a_2, \ldots, a_n) \in dom\ f$, und es gilt

$$\begin{aligned} \psi(g(b_1, b_2, \ldots, b_n)) &= \psi(\varphi(f(a_1, a_2, \ldots, a_n))) \\ &= f(\psi_1(b_1), \psi_2(b_2), \ldots, \psi_n(b_n)) \end{aligned}$$

   Das ist die Homomorphieeigenschaft der Elemente von $\Psi_{(\Gamma,\Lambda)} = \{\psi, \psi_1, \psi_2, \ldots, \psi_n\}$. Dass $\Psi_{(\Gamma,\Lambda)} = \Phi^{-1}_{(\Lambda,\Gamma)}$ ist, folgt aus den entsprechenden Eigenschaften der Funktionen.