Prinzipien, Algorithmen und Modelle der Nebenläufigen Programmierung

Wintersemester 2020/21

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67
80538 München
Email: david.sabel@ifi.lmu.de

Stand: 19. Februar 2021

Inhaltsverzeichnis

1 Einleitung

Wir motivieren in diesem Kapitel, warum Nebenläufigkeit und Nebenläufige Programmierung eine wichtige Rolle in der Informatik spielen, die zunehmend an Bedeutung gewinnt. Außerdem wird ein kurzer Überblick über den (geplanten) Inhalt der Vorlesung und einige Literaturempfehlungen gegeben. In Abschnitt 1.5 werden Annahmen über das betrachtete Modell von Systemen bzw. Programmiersprachen erläutert, insbesondere dazu, wie nebenläufige Programme ausgeführt werden. Wir schließen das Kapitel mit einer kleiner Übersicht über nebenläufiges Programmieren in Java, da wir Java für einige Beispielprogramme verwenden möchten.´

1.1 Warum nebenläufige Programmierung?

Programme, wie sie in den ersten Semestern des Studiums behandelt werden, sind sequentielle Programme, d.h. spätestens auf Maschinen-Code-Ebene bestehen diese aus einer Folge von (Maschinen-)Befehlen, die sequentiell (also nacheinander) ausgeführt werden und dabei den den Zustand des Rechners (also z.B. den Hauptspeicher und die Register) verändern.

Nebenläufige oder parallele Programme, die mehrere Befehle (quasi) gleichzeitig durchführen, werden in den Grundvorlesungen hingegen nur wenig behandelt. Nichtsdestotrotz gibt es einige gute Gründe, solche Programme und die dazu gehörigen Programmiermethoden und Programmierabstraktionen genauer zu studieren.

Eine erste Motivation ist, dass aktuelle reale Systeme in den meisten Fällen nie rein sequentiell sind. So führen z.B. Betriebssysteme verschiedene Aufgaben aus Sicht des Benutzers gleichzeitig aus: Beispielsweise kann ein Dokument gedruckt werden, während man verschiedene Webseiten aufruft, dabei die Maus bewegt und Musik hört. All diese Aufgaben werden (zumindest aus Sicht des Benutzers) gleichzeitig durchgeführt und sind ohne nebenläufige Programmierung undenkbar. Genau wie Betriebssysteme benötigen einfache Anwendungen mit graphischer Benutzeroberfläche Nebenläufigkeit, um das gleichzeitige Erledigen verschiedener Aufgaben (Fensteraktualisierung, Eingabeverarbeitung, usw.) zu realisieren.

Nebenläufige Programme können sogar dann schneller sein als rein sequentielle Programme, wenn nur ein Prozessor vorhanden ist: Manchmal können sie das System besser auslasten. Als einfaches Beispiel betrachte man ein Programm, das zwei Aufgaben erfüllen soll: Zum einen soll eine große Datei auf ein langsames Bandlaufwerk geschrieben werden, zum anderen sollen einige große wissenschaftliche Berechnungen durchgeführt werden. Ein sequentielles Programm muss beide Aufgaben nacheinander durchführen, was für die Auslastung des Prozessors schlecht sein kann, da er beim Schreiben in die Datei gar nicht rechnet, sondern nur darauf wartet, dass die Daten auf das Band geschrieben sind. Ein nebenläufiges Programm kann hier schneller sein: Während die Schreiboperationen auf das Band stattfinden, kann nebenläufig ein anderer Prozess die aufwändigen Berechnungen durchführen.

Ein anderer Bereich ist das Web-Programming. Im Client-Server-Modell bietet der Server Dienste für den Client an (z.B. den Abruf einer Webseite). Natürlich sollte der Server mehrere Clients quasi gleichzeitig bedienen können (und z.B. beim Verbindungsverlust eines Clients nicht aufhören, mit den anderen Clients zu kommunizieren), was nebenläufige Programmierung des Servers erfordert.

Ein weiterer Anwendungsbereich der nebenläufigen bzw. parallelen Programmierung sind solche Probleme, deren Instanzen sich nur durch parallele bzw. auch verteilte Berechnung auf vielen Computern (z.B. mittels GRID-Computing) lösen lassen.

Zu guter Letzt gewinnt die nebenläufige Programmierung aufgrund der Entwicklungen im Hardwaredesign immer mehr an Bedeutung. Seit die Taktfrequenzen am Rande ihrer physikalischen Möglichkeiten sind, sind die Chip-Hersteller dazu übergegangen mehrere (parallel laufende) Prozessoren in ein System zu integrieren, d.h. der aktuelle Trend geht zu Multiprozessor-/ bzw Multikernsystemen und dieser Trend wird aller Voraussicht nach anhalten.

Um diese Systeme vorteilhaft zu nutzen, ist nebenläufige Programmierung unabdingbar[1].

Es bleibt zu klären, was nebenläufige Programmierung von der üblichen Programmierung unterscheidet, und warum nebenläufige Programmierung „schwerer“ ist als sequentielle Programmierung. Hier gibt es mehrere Problemfaktoren zu diskutieren.

Einerseits ist es oft nicht offensichtlich, wie Probleme mit einer sequentiellen Lösung (Implementierung) parallelisiert werden können (z.B. ist paralleles Sortieren keineswegs trivial, gilt aber dennoch gemeinhin als leicht zu parallelisierendes Problem). Andererseits gibt es Probleme, deren Lösung ganz natürlich nebenläufig ist, wie z.B. ein Betriebssystem: Dort sollten einzelne Aufgaben quasi-parallel ausgeführt werden, d.h. das Problem ist gerade zu gemacht für nebenläufige Programmierung. Eine Implementierung in einer rein sequentiellen Programmiersprache müsste hierfür Nebenläufigkeit simulieren, und wäre damit vermutlich schwerer zu implementieren als das direkte Verwenden einer Programmiersprache, die Konstrukte zur nebenläufigen Programmierung bereitstellt.

Die wesentliche Schwierigkeit des nebenläufigen Programmierens liegt jedoch an einer anderen Stelle: Die einzelnen Prozesse müssen miteinander kommunizieren (Daten austauschen) bzw. sich synchronisieren (aufeinander warten). In diesem Bereich passieren die meisten Programmierfehler. Falsche Synchronisation (oder gar fehlende Synchronisation) wird insbesondere dann zum Fehler führen, wenn auf gemeinsame Ressourcen zugegriffen wird. Betrachten wir als Beispiel einen Speicherplatz, der den Kontostand eines Bankkontos darstellt. Greifen nun zwei Prozesse auf den Kontostand zu (z.B. einer macht eine Abbuchung, der andere eine Zubuchung), so kann bei falscher (in diesem Falle naiver) Programmierung der Kontostand nach beiden Aktionen einen falschen Wert erhalten:

[1] Ein interessanter Artikel hierzu ist
„The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software“
von Herb Sutter (zu finden unter http://www.gotw.ca/publications/concurrency-ddj.htm), der aufgrund der Trends zu Multiprozessorsystemen einen Paradigmenwechsel für Programmierer und Programmiersprachen fordert.

(Der Wert von `konto` vor der Ausführung sei 100)

Prozess P:	Prozess Q:
(P1) X:= konto;	(Q1) X := konto;
(P2) konto:= X-10;	(Q2) konto := X+10;

Die Ausführung der Befehle in der Reihenfolge (P1),(P2),(Q1),(Q2) ergibt das gewünschte Ergebnis (der Wert von `konto` ist 100), genau wie die Reihenfolge (Q1),(Q2),(P1),(P2). Allerdings ergeben die Reihenfolgen (P1),(Q1),(P2),(Q2) und (Q1),(P1),(P2),(Q2) einen zu hohen Kontostand (der Wert von `konto` ist am Ende 110), während die Reihenfolgen (P1),(Q1),(Q2),(P2) und (Q1),(P1),(Q2),(P2) einen zu niedrigen Kontostand ergeben (der Wert von `konto` ist 90).

Die Programme sollten derart geändert werden, dass nur die beiden ersten Sequenzen ermöglicht werden, während die anderen Ausführungsreihenfolgen unmöglich sein solten. Hierbei kann man beispielsweise noch in Betracht ziehen, wie viele Prozesse maximal auf die gleiche Ressource zugreifen können. Das Beispiel suggeriert zunächst nur zwei Prozesse, die Lösung sollte jedoch für eine beliebige Anzahl von Prozessen funktionieren, usw.

Dieses einfache Beispiel verdeutlicht auch, dass traditionelles Debuggen (und auch Testen) für nebenläufige Programme nahezu unmöglich ist, da die fehlerhaften Ergebnisse nicht eindeutig reproduziert werden können. So kann z.B. während dem Debugging der Fehler nicht gefunden werden, wenn stets die ersten beiden Auswertungssequenzen ausgeführt werden. Nicht selten tritt eine ungewollte Reihenfolge auf Rechner A mit Compiler C_A nicht auf, aber auf Rechner B mit Compiler C_B tritt die Reihenfolge auf. Daher reicht reines Testen nicht aus, sondern die Korrektheit der Programme sollte für alle Reihenfolgen gelten (und unabhängig vom Rechner und Compiler gelten).

Ein weiteres Problem der Nebenläufigkeit ist es, den Datenaustausch zu organisieren. Gibt es z.B. einen „Produzenten“-Prozess, der Ergebnisse an einen „Konsumenten“-Prozess liefert, so kann der Fall auftreten, dass der Produzent wesentlich schneller Ergebnisse berechnen kann, als der Konsument sie verarbeiten kann. In diesem Fall sollte ermöglicht werden, dass der Produzent trotzdem nach Fertigstellung eines Ergebnisses bereits mit der Berechnung des nächsten Resultats beginnen kann. Hierfür wird ein Zwischenspeicher benötigt, in welchem der Produzent sein Ergebnis ablegen kann. Der Zugriff auf diesen Speicher sollte abgesichert werden, so dass keine fehlerhaften Speicherzustände auftreten können. Die nebenläufige Programmierung beschäftigt sich mit solchen Strukturen.

1.2 Begriffe der nebenläufigen Programmierung

In diesem Abschnitt werden einige grundlegende Begriffe aus dem Gebiet der nebenläufigen Programmierung bzw. nebenläufigen Systeme definiert, erläutert und voneinander abgegrenzt.

Parallelität und Nebenläufigkeit Im Allgemeinen versteht man unter *parallelen Programmen* solche Programme, die auf mehreren Prozessoren gleichzeitig ausgeführt werden, so dass einzelne Berechnungen überlappend sind. Bei der *nebenläufigen Programmierung* (engl. concurrent programming) ist die parallele Ausführung nur ein mögliches Szenario, vielmehr wird davon ausgegangen, dass Berechnungen überlappen *können* aber nicht *müssen*. Nebenläufige Systeme müssen nicht notwendigerweise auf mehreren Prozessoren ausgeführt werden, sie

können auch auf einem einzigen Prozessor quasi-parallel (also stückweise) ausgeführt werden. Das im folgenden verwendete *Modell* von nebenläufigen Systemen geht noch einen Schritt weiter und geht davon aus, dass die kleinsten Berechnungsschritte *atomar* sind, und dass nur ein atomarer Berechnungsschritt zu einer Zeit durchgeführt werden kann, d.h. die überlappende Ausführung von atomaren Berechnungsschritten ist in diesem Modell verboten.

Nebenläufige und verteilte Systeme Unter einem *verteilten System* versteht man ein System, das auf mehreren (oft auch örtlich getrennten) Prozessoren abläuft und keinen gemeinsamen Speicher besitzt. Der Austausch von Daten zwischen den Prozessen erfolgt alleinig über das Senden und Empfangen von Nachrichten. Nebenläufige Systeme können hingegen auch auf einem Prozessor laufen, und Prozesse können gemeinsamen Speicher verwenden.

Prozesse und Threads Die Bezeichnung „Prozess“ wird oft in der Theorie der nebenläufigen Programmierung verwendet, während Programmiersprachen eher das Wort „Thread“ verwenden. Eine weitere Unterteilung in Prozesse und Threads findet oft danach statt, wer die Ausführung der Berechnung kontrolliert: Ist dies das Betriebssystem, so spricht man oft von einem Prozess, ist dies ein Programm (welches selbst als Prozess läuft), so spricht man oft von Threads. Im folgenden werden beide Begriffe verwendet, ohne auf diese strikte Unterteilung zu achten. Programmiersprachen, die Konstrukte zum Erzeugen und Verwalten von Threads zur Verfügung stellen, unterstützen damit das so genannte *Multi-Threading*. Ein verwandter Begriff, der häufig im Rahmen von Betriebssystemen verwendet wird, ist *Multi-Tasking*, welches das gleichzeitige Ausführen verschiedener Aufgaben (wie z.B. Maus bewegen, drucken, etc.) meint. Auf einem Einprozessor-System werden diese Aufgaben nicht echt parallel ausgeführt (sondern stückchenweise abwechselnd), aber für den Benutzer erscheint die Ausführung parallel.

Message-Passing- und Shared-Memory-Modell Das *Message-Passing-Modell* geht davon aus, dass die einzelnen Prozesse nur unterschiedliche Speicherbereiche benutzen und die gesamte Kommunikation zwischen den Prozessen über den Austausch von Nachrichten, d.h. das Senden und Empfangen von Nachrichten, vollzogen wird. Wie bereits erwähnt, ist dieses Modell Grundlage für verteilte Systeme.

Man kann die Kommunikation zwischen Prozessen noch weiter klassifizieren: Bei *synchroner* Kommunikation findet der Austausch der Nachricht zwischen zwei Prozessen direkt (ohne Zeit) statt, während bei *asynchroner* Kommunikation ein Prozess seine Nachricht absendet, die Nachricht aber nicht sofort von einem zweiten Prozess empfangen werden muss (d.h. es darf dabei Zeit vergehen). Für viele Systeme z.B. das Übertragen einer Webseite zu einem Client, ist asynchrone Kommunikation natürlicher. Ein Beispiel zur Unterscheidung der beiden Kommunikationsarten ist die Kommunikation per Telefon oder per Email dar: Beim Telefonieren findet die Kommunikation synchron statt, während Email-Verkehr asynchron ist.

Das *Shared-Memory-Modell* hingegen geht davon aus, dass Prozesse einen gemeinsamen Speicherbereich haben, den alle Prozesse lesen und beschreiben dürfen. Die Kommunikation findet dabei direkt über den gemeinsamen Speicher statt. Hierbei müssen Vorkehrungen getroffen werden, um Konflikte, wie ungewolltes Lesen eines Prozesses vor dem Schreiben eines anderen Prozesses, zu vermeiden. In einem späteren Kapitel werden verschiedene primitive

Operationen zum Zugriff auf den gemeinsamen Speicher eingeführt, die je nach Modell (oder auch der echten Hardware) abweichen können.

1.3 Inhalt der Veranstaltung

An Inhalten sind für die Veranstaltung geplant:

- **Synchronisation und wechselseitiger Ausschluss bei verschiedenen Annahmen über vorhandene Speicheroperationen.** Hierbei wird zunächst angenommen, dass nur ein atomares Lesen und Schreiben auf den Hauptspeicher möglich ist. Dabei wird zunächst das Problem des wechselseitigen Ausschlusses bei genau zwei nebenläufigen Prozessen betrachtet, und die Begriffe Deadlock-Freiheit, Starvation-Freiheit werden eingeführt. Zur Lösung des Problem werden drei klassische Algorithmen besprochen: Die Algorithmen von Dekker, Peterson und Kessels. Schließlich wird das Problem auf beliebig viele (n) Prozesse erweitert und zur Lösung der Algorithmus von Lamport und der Bakery-Algorithmus vorgestellt.

  Nach drei Komplexitätsresultaten zum Problem vom wechselseitigen Ausschluss, werden stärkere Speicheroperationen, wie z.B. Test-and-Set Bits, Read-Modify-Write Objekte und Compare-and-Swap-Objekte, betrachtet. Diese führen zu einfacheren Lösungen zum Problem des wechselseitigen Ausschlusses. Zur Einordnung und Klassifikation der Speicherprimitiven wird schließlich die Konsensuszahl und die sogenannte Herlihy-Hierarchie erörtert.
- **Programmierprimitiven zur nebenläufigen Programmierung.** In diesem Teil werden gebräuchliche Programmierprimitive zur nebenläufigen Programmierung erörtert, wie sie in vielen Programmiersprachen durch Programmbibliotheken zur Verfügung gestellt werden. Dabei werden Semaphoren, Monitore, Kanäle und Tuple Spaces behandelt und Lösungen für einige klassische Probleme der Nebenläufigen Programmierung mittels dieser Datenstrukturen erörtert: Das Problem der Speisenden Philosophen, das Sleeping-Barber Problem, das Cigarette Smokers Problem, Erzeuger-Verbraucher Probleme, die Implementierung von Barrieren und das Readers-& Writers-Problem. Neben Implementierungen in Pseudo-Code, werden auch Implementierungen in der Programmiersprache Java betrachtet.
- **Zugriff auf mehrere Ressourcen.** In diesem Kapitel wird das Problem behandelt, dass mehrere Prozesse auf mehrere Ressourcen zugreifen möchten und die Möglichkeit von daraus resultierenden Deadlocks werden behandelt. Dazu werden Lösungen zur Deadlock-Verhinderung (insbesondere das Zwei-Phasen Sperrprotokoll) und zur Deadlock-Vermeidung (insbesondere der Bankiers-Algorithmus) behandelt. Schließlich wird Transaktionaler Speicher als Programmiermodell eingeführt und einige Unterscheidungskriterien solcher Modelle werden erörtert.
- **Nebenläufigkeit in der Programmiersprache Haskell.** Stark-getypte funktionale Programmiersprachen eignen sich für die parallele und nebenläufige Programmierung, da sie zwischen reinen Berechnungen und Seiteneffekt-behafteten Berechnungen trennen. Haskell ist eine der führenden und populärsten funktionalen Programmiersprachen. Als Beispiel wird in diesem Kapitel die die Umsetzung von nebenläufiger Programmierung in Haskell erörtert. Insbesondere wird es eine Einführung in Concurrent Haskell geben, aber auch in Software Transactional Memory in Haskell.

- **Semantische Modelle nebenläufiger Programmiersprachen.** Nach einer allgemeinen Einführung zur Semantik von Programmiersprachen, wird der π-Kalkül als Message-Passing-Modell eingeführt und seine Varianten analysiert. Als Shared-Memory-Modell wird der CHF-Kalkül erörtert, der eine Kernsprache von Concurrent Haskell ist. Hierbei werden auch Bezüge zwischen den Modellen hergestellt und je nach gegebener Zeit noch weitere Modelle betrachtet.

1.4 Literatur

Im folgenden sind einige Literaturhinweise aufgelistet.

– Über Grundprinzipien der nebenläufigen Programmierung

Ben-Ari, M. (**2006**). *Principles of concurrent and distributed programming*. Addison-Wesley, Harlow, UK.

Herlihy, M. und Shavit, N. (**2008**). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Raynal, M. (**2013**). *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Berlin Heidelberg.

Reppy, J. H. (**2007**). *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England..

Taubenfeld, G. (**2006**). *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

– Über Prozesskalküle, insbesondere den π-Kalkül:

Milner, R. (**1999**). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge, England..

Sangiorgi, D. & Walker, D. (**2001**). *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

– Zu Concurrent Haskell und STM-Haskell:

Marlow, S. (**2011**). Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, & R. Plasmeijer, editors, *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, volume 7241 of *Lecture Notes in Computer Science*, pages 339–401. Springer.

Marlow, S. (**2013**). *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. O'Reilly Media, Inc.

Peyton Jones, S. & Singh, S. (**2009**). A tutorial on parallel and concurrent programming in Haskell. In P. Koopman, R. Plasmeijer, & D. Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 267–305. Springer. ISBN-13 978-3-642-04651-3.

Peyton Jones, S. L. (**2001**). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, & R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, Amsterdam, The Netherlands.

Peyton Jones, S. L., Gordon, A., & Finne, S. (**1996**). Concurrent Haskell. In *POPL '96: 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308. ACM, St Petersburg Beach, Florida.

Kapitel 24 und 28 aus: **O'Sullivan, B., Goerzen, J., & Stewart, D.** (**2008**). *Real World Haskell*. O'Reilly Media, Inc.

Kapitel 24 (Beautiful Concurrency, Simon Peyton-Jones) aus: **Oram, A. & Wilson, G.** (**2007**). *Beautiful code*. O'Reilly Media, Inc.

Schmidt-Schauß, M. & Sabel, D. (**2013**). Correctness of an STM Haskell implementation. In G. Morrisett & T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 161–172. ACM.

– Zum CHF-Kalkül:

- **Sabel, D. & Schmidt-Schauß, M.** (**2011**). A contextual semantics for Concurrent Haskell with futures. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, pages 101–112. ACM, New York, NY, USA.

  Sabel, D. (**2012**). An abstract machine for concurrent haskell with futures. In S. Jähnichen, B. Rumpe, & H. Schlingloff, editors, *Software Engineering 2012 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin*, volume 199 of *LNI*, pages 29–44. GI.

- **Sabel, D. & Schmidt-Schauß, M.** (**2012**). Conservative concurrency in Haskell. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 561–570. IEEE.

1.5 Modellannahmen

Als wichtigste Annahme für sämtliche Modelle der nebenläufigen Programmierung wird angenommen, dass nur ein Berechnungsschritt gleichzeitig durchgeführt werden kann. D.h. es wird davon ausgegangen, dass nie zwei Berechnungsschritte zur gleichen Zeit stattfinden oder sich zeitlich überlappen. Hierbei ist die Definition eines solchen Berechnungsschrittes vom Modell abhängig und somit nicht allgemein definierbar. Die Berechnungsschritte werden dann zwar sequentiell durchgeführt, jedoch wird (fast) beliebiges „Interleaving" (Vermischen bzw. Verzahnen) der Berechnungsschritte erlaubt, d.h. die Reihenfolge, wann welcher Prozess einen Schritt durchführt, ist nicht festgelegt, sondern wahllos. Das „fast" im letzten Satz rührt daher, dass eine gewisse Fairness bei der Berechnung unterstellt wird, die später erläutert wird.

Die folgende Definition von nebenläufigen Programmen, beachtet gerade die Interleaving-Annahme.

Definition 1.5.1 (Nebenläufiges Programm, Interleaving-Annahme)**.** *Ein* nebenläufiges Programm *besteht aus einer endlichen Menge von Prozessen, wobei ein einzelner Prozess aus einem sequentiellen Programm besteht, welches in* atomaren Berechnungsschritten *ausgeführt wird. Die Ausführung des nebenläufigen Programms besteht aus einer Sequenz der atomaren Berechnungsschritte der Prozesse, die* beliebig verzahnt *sein können.*

Da nicht ausschließlich imperative Programme betrachtet werden, wurde der Begriff „atomarer Befehl“ oder „atomare Anweisung“ in der Definition vermieden, denn es sollen auch z.B. funktionale Programme erfasst werden, deren Auswertung im Berechnen von Werten von Ausdrücken besteht. Die Definition erfasst auch solche Programme, da lediglich das Berechnen des Wertes in atomaren Schritten definiert sein muss.

Im folgenden wird die Interleaving-Annahme diskutiert und es werden Begründungen dafür gegeben, warum diese zunächst sehr einschränkende (und zum Teil auch unrealistisch erscheinende) Annahme durchaus sinnvoll ist.

Granularität Da die Interleaving-Annahme keine Aussage darüber macht, was ein Berechnungsschritt ist, können diese beliebig klein gewählt werden. Somit könnte man immer noch ein Modell bilden, indem die Schritte derart klein sind, dass die nicht-parallele Ausführung der Schritte nicht bemerkbar ist.

Multitasking-Systeme Für Einprozessor-Systeme, die Multitasking durchführen, passt die Interleaving-Annahme sehr gut, da dort sowieso nur ein Schritt zur gleichen Zeit durchgeführt werden kann. Allerdings sind die in der Realität anzutreffenden Implementierungen meist derart, dass jeder Prozess eine Zeit lang ausgeführt wird, bis er durch einen globalen Scheduler unterbrochen wird. Hier ist die Annahme jeder beliebigen Reihenfolge der Ausführung eigentlich wieder zu stark, da im Allgemeinen immer mehr als ein Befehl pro Prozess ausgeführt wird. Nichtsdestotrotz kann dieses Verhalten durch Verwenden einer anderen Granularität simuliert werden.

Der wichtigste Grund für beliebiges Interleaving ist jedoch, dass man ansonsten zum einen den Begriff „eine Zeit lang“ formalisieren müsste, was das formale Modell wesentlich verkomplizieren würde, und dass das Modell abhängig von der Implementierung des Schedulers und auch von der Hardware wäre. Eventuell müsste man bei neuer, schnellerer Hardware, die es sich leisten kann, mehr Unterbrechungen durchzuführen, das komplette Modell ändern. Als Konsequenz hätte man, dass vorher korrekte (oder robuste) Algorithmen aufgrund neuer Hardware nicht mehr korrekt wären. Ein weiterer Grund ist, dass der Begriff „eine Zeit lang“ eventuell von äußeren Umständen oder Umwelteinwirkungen abhängen könnte bzw. durch Phasenschwankungen kurzzeitig geändert werden könnte.

Wenn die Korrektheit bezüglich aller möglichen Reihenfolgen nachgewiesen ist, brauchen derartige Umstände nicht beachtet zu werden.

Multiprozessor-Systeme Für Systeme mit mehreren Prozessoren scheint die Interleaving-Annahme zunächst sehr unrealistisch, da in Realität durchaus Berechnungsschritte parallel, d.h. zeitgleich oder zeitüberlappend, stattfinden können. Jedoch werden die gemeinsam genutzten Ressourcen, wie gemeinsamer Speicher, im Allgemeinen auf Hardwareebene vor parallelen Zugriffen geschützt. Damit wird eine Sequentialisierung für solche Berechnungsschritte erzwungen. Es bleibt, dass autonome Berechnungen (ohne Zugriff auf gemeinsame Ressourcen) einzelner Prozesse zeitlich überlappen können. Aber in diesem Fall stört die Interleaving-Annahme nicht, da – von außen betrachtet – nicht beobachtet werden kann, ob zwei Prozesse ihre Berechnungen parallel oder sequentiell mit Interleaving durchführen.

1.5.1 Die Fairness-Annahme

Das durch Definition 1.5.1 erlaubte beliebige Interleaving wird im Allgemeinen durch eine weitere Annahme eingeschränkt. Man fordert im Allgemeinen, dass die Auswertung eine gewisse Art von Fairness beachtet. Es gibt unterschiedlich starke Begriffe von Fairness (siehe z.B. (Varacca & Völzer, 2006)). Im folgenden wird ein einfacher, eher als schwach zu bezeichnender Begriff von Fairness unterstellt:

Definition 1.5.2 (Fairnessannahme)**.** *Jeder Prozess, für den ein Berechnungsschritt (beliebig lange) möglich ist, führt in der Gesamt-Auswertungssequenz diesen Schritt nach* endlich vielen *Berechnungsschritten durch.*

Das folgende Beispiel soll verdeutlichen, welche Auswertungsfolgen durch die Fairness-Annahme verboten sind:

(Der Wert von X vor der Ausführung sei 0)

Prozess P:	Prozess Q:
(P1) **if** $X \neq 10$ **then goto** (P1);	(Q1) X:=10;

Der Prozess P ruft sich ständig selbst auf, bis die Variable X mit dem Wert 10 belegt ist. Der Prozess Q setzt die Variable X auf den Wert 10.

Eine unfaire Auswertungssequenz ist die unendliche Folge (beachte die Fairness-Annahme schränkt nur *unendliche* Auswertungsfolgen ein): $\underbrace{(P1), (P1), (P1), \ldots}_{\text{unendlich oft}}$. Die Sequenz ist unfair, da Q einen Berechnungsschritt machen kann, dieser aber nicht nach endlich vielen Schritten durchgeführt wird. Jede Sequenz $\underbrace{(P1), (P1), \ldots, (P1)}_{n\text{-mal}}(Q1)$ für festes n ist fair: Dies allerdings sogar schon deswegen, da diese Sequenzen alle endlich sind. Da weitere als die vorgestellten Sequenzen für dieses Beispiel nicht möglich sind, erzwingt die Fairness-Annahme in diesem Fall sogar die Terminierung aller Prozesse. Dies ist im Allgemeinen nicht der Fall, wie das folgende, leicht abgewandelte, Beispiel zeigt:

(Der Wert von X vor der Ausführung sei 0)

Prozess P:	Prozess Q:
(P1) **if** X $\neq$ 10 **then goto** (P1); (P2) **goto (P1)**	(Q1) X:=10;

Nun verbietet die Fairness-Annahme weiterhin die unendliche Folge $\underbrace{(P1), (P1), (P1), \ldots}_{\text{unendlich oft}}$, aber die (ebenfalls unendlichen) Folgen $\underbrace{(P1), (P1), \ldots, (P1)}_{n\text{-mal}}(Q1)\ \underbrace{(P1), (P2), (P1), (P2), \ldots}_{\text{unendlich oft}}$ sind alle fair (für beliebiges festes n).

Das „beliebig lange“ in Definition 1.5.2 soll verdeutlichen, dass Schritte, die nur ab und zu möglich sind, aber nicht andauernd, durch die Fairnessannahme nicht erzwungen werden. Bei

einfachen Speicher- und Prozessmodellen sind Schritte immer möglich, aber bei komplizierteren Modelle können Befehle (z.B. das Setzen eines Locks) nur manchmal durchgeführt werden (wenn kein anderer Prozess den Lock hält). Die Fairnessannahme garantiert nicht, dass der Prozess im richtigen Moment rechnen darf (und damit den Lock erhält).

1.5.2 Bekannte Prozesse und atomare Aktionen

Ein weitere Annahme, die *nicht* bis zum Ende des Skripts beibehalten wird, ist, dass ein Programm aus einer festen Anzahl an Prozessen besteht und keine Prozesse neu erzeugt werden können. Diese Annahme ist eher unrealistisch. Sie wird jedoch zunächst benutzt, um Problemstellungen und Lösungen möglichst einfach zu halten. Eine weitere Annahme ist zunächst, dass Prozesse, Programme einer kleinen imperativen Sprache ausführen (einige der Befehle wurden schon in den Beispielen verwendet, z.B. **`if`**-**`then`**, **`goto`**, Zuweisungen $X := 10$, usw.). Die Syntax und Semantik der Sprache wird hier nicht formal festgelegt, da diese bekannt sein sollte. Allerdings wird angenommen, dass jede Befehlszeile (diese Zeilen sind Nummern der Form (P1), (P2) usw. versehen) eine Anweisung darstellt, die atomar – also in einem Berechnungsschritt – ausgeführt wird.

1.6 Nebenläufigkeit in Java

Da für einige der Beispiele auch Implementierungen in der Programmiersprache Java angegeben werden, erläutern wir an dieser Stelle, wie man in Java nebenläufige Threads erzeugt.

Leichtgewichtige Threads sind in Java nativ eingebaut und über die Klasse `Thread` verfügbar. Die Java-Packages in `java.util.concurrent` stellen eine Reihe von Nebenläufigkeitsprimitive zur Verfügung, die zum Teil teilweise im Laufe der Vorlesung besprochen werden. An dieser Stelle beschränken wir uns auf die Erzeugung von Threads. Im Wesentlichen gibt es in Java zwei Techniken, um nebenläufige Threads zu erzeugen, die in den nächsten beiden Unterabschnitten beschrieben werden.

1.6.1 Unterklasse von Thread ableiten

Die einfachere aber nicht so flexible Methode ist es, eine eigene Unterklasse von `Thread` zu implementieren. Diese sollte die `run`-Methode überschreiben, um zu definieren, welcher Code beim Starten des Threads ausgeführt wird (die `run`-Methode ist analog zur `main`-Methode des Hauptprogramms zu sehen).

Ein kleines Beispiel zeigt der folgende Java-Code:

```
class EinThread extends Thread {
  public void run() {
    System.out.println("Hallo vom Thread " + this.getId());
  }
}

public class Main {
  public static void main(String args[]) {
```

```
    for (int k = 1; k <= 10; k++) {
      (new EinThread()).start();
     }
  }
}
```

Die Klasse `EinThread` ist Unterklasse von `Thread` und ihre `run`-Methode druckt einen Text aus, wobei mit `getId()` die eigene Thread-Id (eine Zahl) ermittelt und ausgegeben wird. Die `main`-Methode erzeugt 10 solcher Threads und startet diese Threads durch Aufruf der `start()`-Methode. D.h. erzeugte Threads laufen zunächst nicht und müssen explizit durch diese Methode gestartet werden.

Eine Ausgabe obiges Programms ist:

```
> java Main
Hallo vom Thread 9
Hallo vom Thread 14
Hallo vom Thread 13
Hallo vom Thread 11
Hallo vom Thread 12
Hallo vom Thread 10
Hallo vom Thread 18
Hallo vom Thread 17
Hallo vom Thread 16
Hallo vom Thread 15
```

1.6.2 Das Interface `Runnable`

Ein andere Möglichkeit eigene Threads in Java zu definieren und zu erzeugen, besteht darin, für eine beliebige Klasse das Interface `Runnable` zu implementieren (dafür muss die Klasse die Methode `run`) implementieren. Anschließend können Objekte dieser Klasse dem Konstruktor der Klasse `Thread` übergeben werden. D.h. anstelle die eigene Klasse zur Unterklasse von `Thread` zu machen, werden Objekte der Klasse zum Konstruieren eines Threads übergeben. Das obige Beispiel kann dann wie folgt implementiert werden:

```
class EinThread implements Runnable {
  public void run() {
    System.out.println("Hallo vom Thread "
                       + (Thread.currentThread()).getId());
    }
}

public class Main {
  public static void main(String args[]) {
    for (int k = 1; k <= 10; k++) {
      (new Thread(new EinThread())).start();
    }
```

```
  }
}
```

Die wesentlichen Unterschiede gegenüber der vorherigen Implementierung sind: In der `main`-Methode wird mit `new EinThread()` ein neues Objekt der Klasse `EinThread` und mit `new Thread(...)` ein neues Objekt der Klasse `Thread` erzeugt, welches das Objekt der Klasse `EinThread` als Argument erhält. Das Ausgeben der Thread-Id muss auch angepasst werden: Die Klasse `EinThread` stellt selbst keine Methode `getId()` bereit (da sie ja nicht Unterklasse von `Thread` ist). Deshalb wird mit der Methode `currentThread()` aus der Klasse `Thread` zunächst der Thread ermittelt, indem der Code gerade ausgeführt wird und anschließend für diesen Thread die `getId()`-Methode aufgerufen.

1.6.3 Warten

Die Klasse Thread stellt die Methode `sleep` zur Verfügung, die eine Zahl erwartet und dementsprechend lange in Millisekunden den Thread warten lässt. Hierbei können InterruptedExceptions empfangen werden und daher muss die Methode `sleep` in einem `try-catch`-Block aufgerufen werden, welche diese Exception abfängt.

Das folgende Beispiel lässt jeden Thread $i*100$ Millisekunden warten, wobei i die entsprechende Thread-Id ist. Die `InterruptedException` wird mit `try` und `catch` abgefangen, aber nicht weiter verarbeitet.

```
class EinThread implements Runnable {
    public void run() {
    long myThreadId = (Thread.currentThread()).getId();
    try {
      (Thread.currentThread()).sleep(myThreadId*100);
    }
    catch (InterruptedException e) { };
    System.out.println("Hallo vom Thread " + myThreadId);
  }
}
public class Main {
  public static void main(String args[]) {
    for (int k = 1; k <= 10; k++) {
      (new Thread(new EinThread())).start();
    }
  }
}
```

Da Threads länger warten, wenn sie eine höhere Thread-Id haben, erhält man i.A. die folgende Ausgabe:

```
> java Main
Hallo vom Thread 9
Hallo vom Thread 10
```

```
Hallo vom Thread 11
Hallo vom Thread 12
Hallo vom Thread 13
Hallo vom Thread 14
Hallo vom Thread 15
Hallo vom Thread 16
Hallo vom Thread 17
Hallo vom Thread 18
```

1.6.4 Warten auf Thread-Ende

Mit der Methode `join` kann auf die Terminierung eines Threads gewartet werden. D.h. wenn ein Thread t_1 einen Aufruf t_2`.join()` macht, dann wartet t_1 solange bis, t_2 terminiert ist. Auch hierbei können Interrupted-Exceptions empfangen werden und daher muss die Methode `join` in einem `try-catch`-Block aufgerufen werden.

1.6.5 Variablenwerte

Das Java-Memorymodell erlaubt es, Werte für Variablen in Prozessorcaches zwischenzuspeichern. Dabei kann es durch Codeoptimierungen o.ä. passieren, dass im Prozessorcache nicht der aktuellste Werte der Variablen steht. Durch Verwendung des Qualifiers `volatile` kann eine Variable gekennzeichnet werden, sodass diese nicht im Cache gespeichert wird. Java garantiert dann, dass der gelesene Wert mit dem zuletzt geschriebenen Wert übereinstimmt. Ansonsten wird kein weiterer Schutz (wie z.B. wechselseitiger Ausschluss (siehe nächstes Kapitel) dadurch garantiert.

2

Synchronisation

In diesem Kapitel wird in Abschnitt 2.1 das Problem vom wechselseitigen Ausschluss (Mutual-Exclusion) betrachtet: Das Problem wird formalisiert und Anforderungen an eine Lösung des Problems bzw. Eigenschaften, die eine Lösung erfüllen muss oder kann, werden festgelegt. In Abschnitt 2.2 werden mehrere Algorithmen erläutert, die das Mutual-Exclusion-Problem für zwei Prozesse unter der Annahme lösen, dass an primitiven unteilbaren (d.h. atomaren) Operationen nur Lese- und Schreiboperationen auf den gemeinsamen Speicher zugelassen sind. In Abschnitt 2.3 wird diese Annahme beibehalten und es werden Algorithmen vorgestellt, die das Mutual-Exclusion-Problem für n, d.h. beliebig viele Prozesse lösen. In Abschnitt 2.4 werden einige Komplexitätsresultate erläutert, die für das Mutual-Exclusion-Problem nachgewiesen wurden. Das Modell, welches annimmt, dass atomar nur gelesen oder geschrieben werden kann, wird in Abschnitt 2.5 verlassen und andere atomare Speicheroperationen, die von manchen Rechnerarchitekturen bereit gestellt werden, werden diskutiert. Es werden einige Algorithmen zur Lösung des Mutual-Exclusion-Problems mithilfe dieser neuen Operationen vorgestellt. Schließlich werden die atomaren Speicheroperationen in Abschnitt 2.6 qualitativ anhand der sogenannten Konsensuszahl klassifiziert. Hierbei wird das Konsensusproblem eingeführt und die Konsensushierarchie wird erläutert.

2.1 Das Mutual-Exclusion-Problem

In diesem Abschnitt wird das grundlegende Problem vom wechselseitigen Ausschluss (mutual exclusion). Als einleitendes Beispiel betrachten wir das „Too-Much-Milk“-Problem:

Alice und Bob wohnen zusammen und wollen, dass stets Milch in ihrem Kühlschrank ist. Deshalb geht jeder der beiden Milch kaufen, wenn er sieht, dass keine Milch im Kühlschrank ist. Allerdings funktioniert dies nicht, da sie stets zu viel Milch im Kühlschrank haben, z.B. wenn folgendes passiert:

	Alice	Bob
17:00	kommt nachhause	
17:05	bemerkt: keine Milch im Kühlschrank	
17:10	geht zum Supermarkt	
17:15		kommt nachhause
17:20		bemerkt: keine Milch im Kühlschrank
17:25		geht zum Supermarkt
17:30	kommt vom Supermarkt zurück, packt Milch in den Kühlschrank	
17:40		kommt vom Supermarkt zurück, packt Milch in den Kühlschrank

Zu viel Milch im Kühlschrank

Um dem „Zu viel Milch“-Problem zu entgehen, haben Alice und Bob ausgemacht, Notizen am Kühlschrank zu hinterlassen, die beide lesen können. Dadurch soll sichergestellt werden, dass:

- Höchstens eine der beiden Personen geht Milch kaufen, wenn keine Milch im Kühlschrank ist.
- Eine der beiden Personen geht Milch kaufen, wenn keine Milch im Kühlschrank ist.

Die Lösung, dass z.B. nur Bob Milch kaufen geht, ist nicht möglich, da dann z.B. Alice nachhause kommen könnte, feststellt, dass keine Milch vorhanden ist, und dann unendlich lange darauf wartet, dass Bob Milch kauft. Das verletzt die zweite Bedingung. Eine weitere (unrealistische) Annahme ist, dass Alice und Bob sich nicht sehen können, auch dann wenn beide gleichzeitig zuhause sind.

Erster Lösungsversuch: Wenn keine Notiz am Kühlschrank und keine Milch im Kühlschrank ist, dann schreibe Notiz an den Kühlschrank, gehe Milch kaufen, stelle die Milch in den Kühlschrank und entferne danach die Notiz am Kühlschrank.

Als Prozesse geschrieben:

Programm für Alice:

```
(A1) if keine Notiz then
(A2)   if keine Milch then
(A3)     schreibe Notiz;
(A4)     kaufe Milch;
(A5)     entferne Notiz;
```

Programm für Bob:

```
(B1) if keine Notiz then
(B2)   if keine Milch then
(B3)     schreibe Notiz;
(B4)     kaufe Milch;
(B5)     entferne Notiz;
```

Dummerweise funktioniert dieser Ansatz nicht, da immer noch zu viel Milch im Kühlschrank vorhanden sein kann: Betrachte die Reihenfolge (A1),(A2),(B1),(B2),.... In diesem Fall stellen Alice und Bob jeweils fest, dass keine Notiz am Kühlschrank und keine Milch im Kühlschrank ist. Beide hinterlassen Notizen und gehen Milch kaufen, und entfernen danach ihre Notiz.

Zweiter Lösungsansatz: Hinterlasse als erstes eine Notiz am Kühlschrank, dann prüfe ob eine Notiz des anderen vorhanden ist. Nur wenn keine weitere Notiz vorhanden ist, prüfe ob Milch

vorhanden ist und gehe Milch kaufen, wenn keine Milch vorhanden ist. Danach entferne die Notiz. Als Prozesse geschrieben:

Programm für Alice:	Programm für Bob:
(A1) schreibe Notiz „Alice“;	(B1) schreibe Notiz „Bob“;
(A2) **if** keine Notiz von Bob **then**	(B2) **if** keine Notiz von Alice **then**
(A3) **if** keine Milch **then**	(B3) **if** keine Milch **then**
(A4) kaufe Milch;	(B4) kaufe Milch;
(A5) entferne Notiz „Alice“;	(B5) entferne Notiz „Bob“;

Diese Lösung funktioniert allerdings auch nicht. Es kann passieren, dass keine Milch gekauft wird. Betrachte die Folge (A1),(B1),(A2),(B2),(A5),(B5). Beide hinterlassen ihre Notiz, anschließend stellen beide fest, dass eine Notiz vom jeweils anderen vorhanden ist. Deswegen schauen beide nicht in den Kühlschrank, sondern beide entfernen ihre Notiz.

Dritter Lösungsversuch: Alice macht das gleiche wie vorher. Bob: Schreibe Notiz, warte bis keine Notiz von Alice am Kühlschrank hängt, dann prüfe, ob Milch leer ist, gehe Milch kaufen, entferne eigene Notiz. Als Prozesse:

Programm für Alice:	Programm für Bob:
(A1) schreibe Notiz „Alice“;	(B1) schreibe Notiz „Bob“;
(A2) **if** keine Notiz von Bob **then**	(B2) **while** Notiz von Alice **do skip;**
(A3) **if** keine Milch **then**	(B3) **if** keine Milch **then**
(A4) kaufe Milch;	(B4) kaufe Milch;
(A5) entferne Notiz „Alice“;	(B5) entferne Notiz „Bob“;

Der wesentliche Unterschied zur vorherigen Lösung ist, dass, sobald Bob eine Notiz an den Kühlschrank geschrieben hat, wartet er darauf, dass Alice ihre Notiz entfernt. Diese Lösung funktioniert, wenn man annimmt, dass Alice nicht zu schnell ist: Betrachte die Sequenz $(A1), (B1), (B2), \underbrace{(A2), (A5), (A1), (B2)}_{\text{unendlich oft}}$. In diesem Fall wird nie Milch gekauft werden, da Alice ihre Notiz zwar entfernt, Bob dies jedoch nicht (genauer: nie) bemerkt, da Alice schnell genug eine neue Notiz an den Kühlschrank schreibt.

Eine korrekte Lösung: Für eine korrekte Lösung, verwenden Alice und Bob jeweils zwei Notizen, diese seien mit Alice1, Alice2 bzw. Bob1 und Bob2 bezeichnet.

(A1) schreibe Notiz Alice1;
(A2) **if** Bob2
(A3) **then** schreibe Notiz Alice2;
(A4) **else** entferne Notiz Alice2;
(A5) **while** Bob1 und ((Alice2 und Bob2) oder (nicht Alice2 und nicht Bob2))
(A6) **do skip**;
(A7) **if** keine Milch
(A8) **then** kaufe Milch;
(A9) entferne Alice1;

Programm für Alice

(B1) schreibe Notiz Bob1;
(B2) **if** nicht Alice2
(B3) **then** schreibe Notiz Bob2;
(B4) **else** entferne Notiz Bob2
(B5) **while** Alice1 und ((Alice2 und nicht Bob2) oder (nicht Alice2 und Bob2))
(B6) **do skip**;
(B7) **if** keine Milch
(B8) **then** kaufe Milch;
(B9) entferne Bob1;

Programm für Bob

Um die Korrektheit des Algorithmus zur verdeutlichen, ist die wichtigste Beobachtung, dass für die Notizen Alice2 und Bob2 gilt: Bob bekommt den Vorzug, falls entweder beide Notizen vorhanden sind oder beide nicht vorhanden sind. In beiden anderen Fällen erhält Alice den Vorzug. Die beiden Notizen Alice2 und Bob2 erhalten allerdings nur dann Beachtung, wenn wirklich beide in den Kühlschrank schauen wollen, d.h. die Notizen Alice1 und Bob1 vorhanden sind.

An dieser Stelle wird Korrektheit des Algorithmus nicht weiter diskutiert oder gar bewiesen, da später verschiedene Algorithmen für das Problem des wechselseitigen Ausschlusses erörtert werden. Das „Too-much milk“-Beispiel sollte hier nur verdeutlichen, wie schnell man falsche Algorithmen im Rahmen der Nebenläufigkeit konstruieren kann, und einen ersten Eindruck gegeben, welche Bedingungen gestellt werden sollten, um von einer korrekten Lösung zu sprechen. Diese Bedingungen werden im Folgenden durch verschiedene Begriffe festlegt.

Das Mutual-Exclusion-Problem ergibt sich durch die Garantie, dass der exklusive Zugriff auf eine gemeinsam genutzte Ressource sichergestellt wird, d.h. während ein einzelner Prozess auf eine solche Ressource zugreift, dürfen keine anderen Prozesse auf diese zugreifen. Der Zugriff auf eine solche Ressource (bzw. der dazu gehörige Programmcode) wird als *kritischer Abschnitt* bezeichnet.

Der Zugriff auf gemeinsame Ressourcen tritt häufig auf, z.B. bei Betriebssystemen, Datenbanken, Netzwerken usw. Wie bereits in der Einleitung erläutert, kann der ungeschützte Zugriff zu ungewollten Ergebnissen führen. Situationen, in denen mehrere Prozesse auf eine gemeinsame Ressource zugreifen und in denen der Wert der Ressource von Ausführungsreihenfolge der Prozesse abhängt, nennt man *race conditions*. Das wesentliche Ziel des Mutual-Exclusion-Problems ist es, race conditions zu vermeiden bzw. auszuschließen.

2.1.1 Korrektheitskriterien

Um das Problem des wechselseitigen Ausschlusses formaler zu fassen, wird folgende Annahme über die nebenläufig ablaufenden Prozesse getroffen: Der Code jedes Prozesses hat die Form

```
loop forever
  restlicher Code
  Initialisierungscode
  Kritischer Abschnitt
  Abschlusscode
end loop
```

D.h. ein Prozess läuft in einer Endlosschleife, führt dabei zunächst irgendwelchen Code aus (bezeichnet als „restlicher Code“), der nicht näher interessiert, und betritt dann den kritischen Abschnitt in drei Phasen: Zunächst wird ein Code zur Initialisierung bzw. zur Synchronisierung ausgeführt (Initialisierungscode), danach der kritische Abschnitt betreten, und am Ende ein Abschlusscode durchgeführt, der z.B. dafür benutzt wird, den anderen Prozessen mitzuteilen, dass der kritische Abschnitt verlassen wurde.

Das Mutual-Exclusion-Problem wird *gelöst*, indem man den Code für den Initialisierungscode und den Abschlusscode angibt, wobei die folgenden Anforderungen erfüllt werden müssen:

- *Wechselseitiger Ausschluss*: Es sind niemals zwei oder mehr Prozesse zugleich in ihrem kritischen Abschnitt.
- *Deadlock-Freiheit*: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann betritt irgendein Prozess schließlich den kritischen Abschnitt.

Man beachte, dass die Deadlock-Freiheit nur erfordert, dass *irgendein* Prozess den kritischen Abschnitt betritt, nicht jedoch notwendigerweise der durch die Definition der Deadlock-Freiheit *erwähnte* Prozess, welcher den kritischen Abschnitt betreten möchte. Die Deadlock-Freiheit garantiert, dass das nebenläufige Programm (als Gesamtsystem) nicht „hängen bleibt“, d.h. nie die Situation eintritt, dass alle Prozesse auf das Betreten des kritischen Abschnitts unendlich lange aufeinander warten. Deadlock-Freiheit garantiert jedoch *nicht*, dass ein wartender Prozess nach endlicher Zeit seinen kritischen Abschnitt betreten wird. Es könnte z.B. sein, dass immer wieder ein anderer Prozess den kritischen Abschnitt betritt und daher Deadlock-Freiheit garantiert wird.

Starvation-Freiheit (Starvation = Verhungern) ist gegenüber der Deadlock-Freiheit eine stärkere Forderung:

- *Starvation*-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann muss er ihn nach endlich vielen Berechnungsschritten betreten.

Manchmal kann auf Starvation-Freiheit verzichtet werden, wenn es im Gesamtablauf des Programms nur „wenige“ Situationen geben wird, in denen einzelne Prozesse auf einen kritischen Abschnitt zugreifen möchten.

Offensichtlich gilt:

Lemma 2.1.1. *Ein Starvation-freier Algorithmus ist auch Deadlock-frei.*

Die Eigenschaft des wechselseitigen Ausschlusses ist eine so genannte Sicherheitseigenschaft (safety property), eine Eigenschaft, die immer erfüllt sein muss. Die anderen beiden Eigenschaften sind hingegen so genannte Lebendigkeitseigenschaften (liveness property), was Eigenschaften sind, die irgendwann wahr werden müssen. Mithilfe von Temporallogik kann man

für Algorithmen nachweisen, dass solche Eigenschaften (formuliert als temporallogische Formeln) erfüllt sind. Hierbei kann das so genannte Model-Checking verwendet werden, um diesen Nachweis zu führen. Im folgenden werden diese Nachweise eher informell durchgeführt (ohne Verwendung von Temporallogik).

Für die genauere Spezifikation des Mutual-Exclusion-Problems werden folgende Annahmen getroffen:

- Im restlichen Code kann ein Prozess keinen Einfluss auf andere Prozesse nehmen. Ansonsten werden keine Annahmen über den restlichen Code getroffen (Er kann Endlosschleifen enthalten oder terminieren usw.)
- Ressourcen (Programmvariablen), die im Initialisierungscode oder Abschlusscode benutzt werden, dürfen durch den Code im kritischen Abschnitt und den restlichen Code *nicht* verändert werden.
- Es treten keine Fehler bei der Ausführung des Initialisierungscodes, des Codes im kritischen Abschnitt und im Abschlusscode auf.
- Der Code im kritischen Abschnitt und im Abschlusscode besteht nur aus endlich vielen Ausführungsschritten, insbesondere enthält der Code keine Endlosschleife und ist „wait-free“, d.h. es gibt keine Warteschleifen, die potentiell unendlich lange laufen können. Das impliziert (aufgrund der getroffenen Fairnessannahme), dass der Prozess nach dem Betreten des kritischen Abschnitts diesen wieder nach endlich vielen Berechnungsschritten verlässt.

2.2 Mutual-Exclusion Algorithmen für zwei Prozesse

Im folgenden werden einige bekannte Algorithmen für das Mutual-Exclusion-Problem dargestellt. Hierbei wird von *zwei* nebenläufigen Prozessen ausgegangen unter der Annahme, dass es Lese- und Schreibbefehle für Speicherregister gibt, die atomar ausgeführt werden.

Der Befehl **`await`** *Bedingung*; wird als Abkürzung für **`while`** ¬*Bedingung* **`do skip`**; verwendet, wobei pro Berechnungsschritt nur einmal die Bedingung ausgewertet werden darf.

2.2.1 Der Algorithmus von Dekker

Der Algorithmus von Dekker ist historisch gesehen der erste Algorithmus, der das Mutual-Exclusion-Problem für zwei Prozesse löst. Der Code für beide Prozesse ist in Abbildung 2.1 dargestellt.

Die Prozesse P und Q benutzen jeweils eine boolesche Variable wantp (bzw. wantq), um erkennbar zu machen, dass sie den kritischen Abschnitt betreten möchten. Des Weiteren benutzen sie eine gemeinsame Variable turn, die angibt, welcher Prozess – im Falle des gleichzeitigen Zutritts auf den kritischen Abschnitt – den Abschnitt betreten darf. Wenn turn den Wert 1 hat, so wird P den kritischen Abschnitt betreten, anderenfalls (turn = 2) wird Q den Abschnitt betreten.

Lemma 2.2.1. *Der Algorithmus von Dekker erfüllt den wechselseitigen Ausschluss.*

Beweis. Nehme an, die Aussage sei falsch. Dann gibt es einen Zustand des Programms, so dass sich sowohl P als auch Q in ihrem kritischen Abschnitt befinden. Das impliziert, dass beide

Initial: wantp = False, wantq = False, turn =1

Prozess P:

```
loop forever
(P1)   restlicher Code
(P2)   wantp := True;
(P3)   while wantq do
(P4)      if turn=2 then
(P5)         wantp := False;
(P6)         await turn=1;
(P7)         wantp := True;
(P8)   Kritischer Abschnitt
(P9)   turn := 2;
(P10)  wantp := False
end loop
```

Prozess Q:

```
loop forever
(Q1)   restlicher Code
(Q2)   wantq := True;
(Q3)   while wantp do
(Q4)      if turn=1 then
(Q5)         wantq := False;
(Q6)         await turn=2;
(Q7)         wantq := True;
(Q8)   Kritischer Abschnitt
(Q9)   turn := 1;
(Q10)  wantq :=False
end loop
```

Abbildung 2.1: Der Algorithmus von Dekker

Prozesse die **while**-Schleife irgendwann übersprungen haben, d.h. die Bedingung (in Zeilen (P3), bzw. (Q3)) durchlaufen haben. Es lassen sich die folgenden Fälle unterscheiden:

- Beide Prozesse sind nie durch den Schleifenkörper gelaufen. Das ist unmöglich, da dann wantp und wantq falsch sein müssen. O.B.d.A. sei Q der zweite Prozess, der die **while**-Bedingung prüft. Dann hat P wantp vorher auf wahr gesetzt und kann danach wantp nicht mehr auf False setzen (und da der kritische Abschnitt wantp, wantq und turn nicht verändern darf!)
- Ein Prozess hat den Schleifenkörper durchlaufen während der andere im kritischen Abschnitt ist. Dann lässt sich leicht nachprüfen, dass er den kritischen Abschnitt nicht mehr betreten kann.
- Beide Prozesse durchlaufen den Schleifenkörper. Dann muss einer der beiden Prozesse am **await**-Statement hängen bleiben (da turn erst nach Durchlaufen des kritischen Abschnitts verändert wird).

Die Fälle zeigen, dass die Annahme falsch ist, d.h. der Algorithmus erfüllt den wechselseitigen Ausschluss. □

Lemma 2.2.2. *Der Algorithmus von Dekker ist Starvation-frei.*

Beweis. Nehme an, die Aussage sei falsch. Dann betritt einer der beiden Prozesse den Initialisierungscode (Zeilen 3-7), aber nie den kritischen Abschnitt. Da sich beide Prozesse symmetrisch verhalten, nehme ohne Beschränkung der Allgemeinheit an, dass P verhungert. Hierfür kann es nur zwei Gründe geben: Die **while**-Schleife wird unendlich oft durchlaufen, oder der Prozess P kommt nicht aus einem **await**-Konstrukt heraus (Zeile (P6)). Betrachte zunächst den zweiten Fall. Dann muss turn den Wert 2 haben. Es lassen sich die folgenden Fälle unterscheiden, jenachdem, wo sich Q befindet:

- Q ist in seinem restlichen Code. Das ist unmöglich, da wantq wahr gewesen sein muss als P die **while**-Schleife betreten hat. Zu diesem Zeitpunkt muss Q in den Zeilen (Q2) bis

(Q9) gewesen sein. Um danach wieder in (Q1) zu kommen, hätte Q den Wert turn in Zeile (Q9) auf 1 setzen müssen. Es wurde aber angenommen, dass turn den Wert 2 hat.

- Q ist im kritischen Abschnitt oder im Abschlusscode (Zeile (Q8) bis (Q10)). Dann wird der Wert von turn auf 1 gesetzt (in Zeile (Q9)). Da Q anschließend den Wert von turn nicht mehr verändern kann, muss P das **`await`**-Statement nach endlich vielen Schritten verlassen.
- Q ist im Initialisierungscode. Da P in Zeile (P6) ist, muss wantp falsch sein. Q wird somit in den kritischen Abschnitt eintreten (und kann auch nicht an seinem **`await`**-Statement in Zeile (Q6) verhungern, da turn den Wert 2 hat). Nun kann der gleiche Schluss wie vorher gezogen werden.

Betrachten nun den ersten Fall: P durchläuft endlos die **`while`**-Schleife, bleibt aber nicht endlos am **`await`**-Konstrukt hängen. Dadurch lässt sich folgern, dass nach einer endlichen Folge von Berechnungsschritten turn den Wert 1 haben muss und dieser Wert nicht geändert wird (das kann nur P im Abschlusscode, den P per Annahme aber nie erreicht). Das impliziert, dass entweder wantq falsch sein muss (das ist nicht möglich, da P die **`while`**-Schleife durchläuft), oder dass Q an seinem **`await`**-Konstrukt in Zeile (Q6) warten muss. Da dann aber wantq vorher in Zeile (Q5) auf falsch gesetzt wurde, ist es unmöglich, dass P seine **`while`**-Schleife unendlich lange durchläuft. D.h. die Annahme war falsch, der Algorithmus ist Starvation-frei. □

Da der Algorithmus von Dekker den wechselseitigen Ausschluss erfüllt und Starvation-frei (und damit auch Deadlock-frei) ist, gilt:

Satz 2.2.3. *Der Algorithmus von Dekker löst das Mutual-Exclusion Problem.*

2.2.2 Der Algorithmus von Peterson

Der Algorithmus von Peterson ähnelt dem Algorithmus von Dekker, aber ihm genügt ein **`await`**-Statement (ohne weitere **`while`**-Schleife) und er ist daher etwas einfacher. Der Algorithmus ist in Abbildung 2.2 dargestellt.

Initial: wantp = False, wantq = False, turn egal

Prozess P:	Prozess Q:
`loop forever`	**`loop forever`**
(P1) restlicher Code	(Q1) restlicher Code
(P2) wantp := True;	(Q2) wantq := True;
(P3) turn := 1;	(Q3) turn := 2;
(P4) **`await`** wantq = False **`or`** turn = 2	(Q4) **`await`** wantp = False **`or`** turn = 1
(P5) Kritischer Abschnitt	(Q5) Kritischer Abschnitt
(P6) wantp := False;	(Q6) wantq := False;
`end loop`	**`end loop`**

Abbildung 2.2: Der Algorithmus von Peterson

Es lässt sich leicht nachweisen, dass gilt:

Satz 2.2.4. *Der Algorithmus von Peterson löst das Mutual-Exclusion-Problem. Er garantiert den wechselseitigen Ausschluss und ist Starvation-frei.*

Beweis. Der Algorithmus garantiert den wechselseitigen Ausschluss: Nehme an, dies sei falsch. Dann gibt es einen Zustand, so dass P und Q beide im kritischen Abschnitt sind. Die beiden jeweils letzten Tests in Zeilen (P4) und (Q4) können nicht direkt nacheinander ausgeführt worden sein, da turn dann nur einen der beiden Prozesse durchgelassen hätte und wantp und wantq beide wahr sein müssen. D.h. der zweite Prozess, der den kritischen Abschnitt betreten hat, muss mindestens eine Zuweisung durchgeführt haben, *nachdem* der erste Prozess den kritischen Abschnitt betreten hat. Die letzte dieser Zuweisungen setzt aber den Wert von turn so, dass der setzende Prozess nicht über die **`await`**-Bedingung hinweg kommt, da sie stets dem anderen Prozess den Vorrang gibt.

Der Algorithmus ist Starvation-frei: Nehme an, das sei falsch. Dann verbleibt ein Prozess für immer in seinem Initialisierungscode. Hierfür kann nur das **`await`**-Konstrukt in Frage kommen, da es sonst keine Schleifen gibt. Für den anderen Prozess können drei Fälle zutreffen:

1. Er verbleibt für immer in seinem restlichen Code. Dann ist seine want-Variable falsch und damit kann der angeblich verhungernde Prozess nicht verhungern.
2. Er verbleibt auch für immer in seinem Initialisierungscode. Das kann nicht sein, da für einen der beiden Prozesse die **`await`**-Bedingung stets wahr ist.
3. Er durchläuft wiederholend seinen kritischen Abschnitt. Das kann nicht sein, da er nach dem ersten Durchlaufen, dann beim zweiten Ausführen des Initialisierungscode die turn-Variable so setzt, dass der angeblich verhungernde Prozess für seine **`await`**-Bedingung den Wert „wahr“ erhalten muss. □

Es ist noch anzumerken, dass die Bedingung des **`await`**-Konstrukts dabei nicht notwendigerweise atomar ausgewertet werden muss, d.h. man kann auch dann die Korrektheit des Algorithmus zeigen, wenn die beiden Teilbedingungen nacheinander (und nicht atomar) ausgewertet werden.

2.2.3 Der Algorithmus von Kessels

Als dritten Algorithmus wird der Algorithmus von Kessels vorgestellt, der im Gegensatz zu den beiden anderen Algorithmen keine Variablen (oder Speicherplätze) benötigt, die von mehreren Prozessen *beschrieben* werden. Solche *single-writer, multiple reader*-Algorithmen gelten als gut umsetzbar in Message-Passing-Modellen. Die Idee des Algorithmus von Kessels ist ähnlich zum bereits gezeigten Algorithmus zum Too-Much-Milk-Problem und stellt eine Abwandlung des Peterson-Algorithmus dar: Anstatt die Variable turn von beiden Prozessen beschreiben zu lassen, werden zwei Variablen turnp und turnq verwendet, sodass gilt

$$\text{turn} = 1, \quad \text{wenn turnp} = \text{turnq}$$
$$\text{turn} = 2, \quad \text{wenn turnp} \neq \text{turnq}$$

Beide Prozesse verwenden des Weiteren jeweils eine lokale Variable (localp, bzw localq). Der Code des Algorithmus von Kessels ist in Abbildung 2.3 dargestellt. Der Algorithmus von Kessels garantiert wechselseitigen Ausschluss und ist Starvation-frei. Der Beweis ist ähnlich zum Korrektheitsbeweis des Algorithmus von Peterson.

Initial: wantp = False, wantq = False, turnp, turnq egal

Prozess P:

```
loop forever
(P1)  restlicher Code
(P2)  wantp := True;
(P3)  localp := turnq;
(P4)  turnp := localp;
(P5)  await
        wantq = False
        or localp ≠ turnq
(P6)  Kritischer Abschnitt
(P7)  wantp := False;
end loop
```

Prozess Q:

```
loop forever
(Q1)  restlicher Code
(Q2)  wantq := True;
(Q3)  localq := if turnp = 1
                  then 2
                  else 1;
(Q4)  turnq := localq;
(Q5)  await
        wantp = False
        or localq = turnp
(Q6)  Kritischer Abschnitt
(Q7)  wantq := False;
end loop
```

Abbildung 2.3: Kessels' Algorithmus

2.3 Mutual-Exclusion Algorithmen für n Prozesse

In diesem Abschnitt werden Algorithmen für das Mutual-Exclusion-Problem betrachtet, die für eine beliebige Anzahl von Prozessen funktionieren. Dabei wird weiterhin die Modellannahme beibehalten, dass nur (atomare) Lese- und Schreiboperationen auf den Speicher verfügbar sind, und dass die Anzahl n der Prozesse bekannt ist.

Eine einfache Lösung des Mutual-Exclusion-Problems für n Prozesse besteht darin, die Algorithmen für zwei Prozesse auf n Prozesse zu erweitern, indem man einen Divide-and-Conquer-Ansatz verfolgt: n Prozesse werden in zwei Gruppen zu je $n/2$ Prozesse aufgeteilt und anschließend wird mit beiden Gruppen rekursiv fortgefahren, sodass aus jeder der beiden Gruppen ein „Gewinner“ ermittelt wird, der noch die Chance hat, den kritischen Abschnitt zu betreten. Anschließend wird mit einem Mutual-Exclusion-Algorithmus für 2 Prozesse entschieden, welcher der beiden Gewinner den kritischen Abschnitt betreten darf.

Tatsächlich lässt sich damit aus einem Starvation-freien Algorithmus für das Mutual-Exclusion Problem zweier Prozesse ein Algorithmus für das Mutual-Exclusion Problem von n Prozessen erzeugen, der wiederum Starvation-frei ist.

Nachteilig bei solchen Algorithmen ist jedoch die Laufzeit: Egal, ob mehrere Prozesse den kritischen Abschnitt betreten möchten oder nicht, jeder einzelne Prozess, der den kritischen Prozess betritt, muss vorher $\lceil \log_2 n \rceil$-mal „Gewinner“ sein, also $\lceil \log_2 n \rceil$-mal den Initialisierungscode für die 2-Prozess-Lösung durchführen, entsprechendes gilt für den Abschlusscode im Anschluss an den kritischen Abschnitt.

Da in vielen nebenläufigen Programmen das gleichzeitige Zugreifen auf einen kritischen Abschnitt eher selten zu erwarten ist (und gar erst das gleichzeitige Zugreifen aller n Prozesse), sind bessere Lösungen erwünscht.

2.3.1 Lamports schneller Algorithmus

Wir betrachten nun den Algorithmus von Lamport, der eine schnellere Lösung bereitstellt. Das Programm des i. Prozesses ist in Abbildung 2.4 dargestellt.

Initial: y=0, für alle $i \in \{1, \ldots, n\}$: want[i]=False, Wert von x egal

Programm des i. Prozesses

```
loop forever
(1)   restlicher Code
(2)   want[i] := True;
(3)   x := i;
(4)   if y ≠ 0 then
(5)       want[i] := False;
(6)       await y = 0;
(7)       goto (2);
(8)   y := i;
(9)   if x ≠ i then
(10)      want[i] := False;
(11)      for j := 1 to n do await ¬ want[j];
(12)      if y ≠ i then
(13)              await y = 0;
(14)              goto (2);
(15)  Kritischer Abschnitt
(16)  y := 0;
(17)  want[i] := False;
end loop
```

Abbildung 2.4: Lamports Algorithmus für n Prozesse

Zur Verständlichkeit des Algorithmus ist ein vereinfachtes Flussdiagramm in Abbildung 2.5 dargestellt.

Die einzelnen Teile des Algorithmus lassen sich wie folgt erklären:

In Zeile (2) setzt Prozess i zunächst seine want-Variable auf True, um anzuzeigen, dass er den kritischen Abschnitt betreten möchte. Direkt danach (Zeile (3)) setzt er x auf den Wert i, um später beobachten zu können, ob weitere Prozesse den kritischen Abschnitt betreten möchten. Die Abfrage in Zeile (4) dient dazu, festzustellen, ob ein anderer Prozess bereits im kritischen Abschnitt ist, oder versucht diesen zu erreichen. Ist dies der Fall, so bricht Prozess i den Versuch ab, den kritischen Abschnitt direkt zu erreichen: Er setzt seine want-Variable auf falsch, wartet bis $y = 0$ gilt (dann hat ein Prozess den kritischen Abschnitt erfolgreich verlassen) und startet anschließend einen neuen Versuch (all das geschieht in den Zeilen (5)-(7)).

In Zeile (8) setzt Prozess i die Variable y auf den Wert i. Die Variable y wirkt wie eine Barriere: Sobald sie auf einen Wert verschieden von 0 gesetzt wird, wird kein anderer Prozess mehr an der Abfrage in Zeile (4) vorbeikommen. Wenn mehrere Prozesse die Abfrage (4) erfolgreich durchlaufen, und dann die Zuweisung $y := i$ in Zeile (8) durchführen, dann wird (wie später

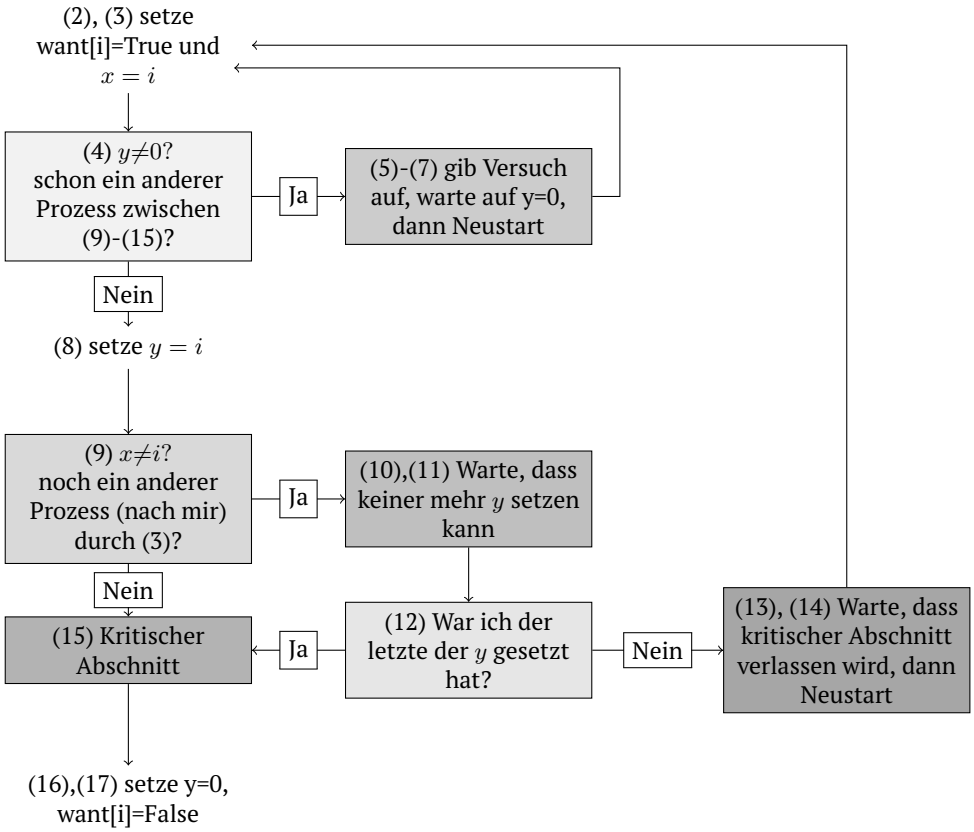


Abbildung 2.5: Flussdiagramm zum Lamport-Algorithmus

gezeigt wird) derjenige Prozess in den kritischen Abschnitt gelangen, der *zuletzt* den Wert von y gesetzt hat.

Durch die Abfrage in Zeile (9) nach dem Wert von x, kann ein Prozess direkt in den kritischen Abschnitt gelangen: Falls ein Prozess i in Zeile (9) liest, dass $x = i$ gilt, dann weiß er, dass kein anderer Prozess j, der y noch verändern könnte, in dessen Zeile (9) $x = j$ lesen wird. Deshalb kann Prozess i in diesem Fall sicher in den kritischen Abschnitt eintreten.

Falls die Abfrage in Zeile (9) falsch ist, dann gibt es weitere Prozesse, die x verändert haben. In diesem Fall wird durch die `for`-Schleife in Zeile (11) solange gewartet, bis alle Prozesse warten (oder gar nicht in den kritischen Abschnitt wollen). Das sichert zu, dass niemand den Wert von y verändern kann. In Zeile (12) schließlich fragt Prozess i, ob er der letzte war, der y gesetzt hat. Ist dies der Fall, so darf er in den kritischen Abschnitt, andernfalls bricht er seinen Versuch ab (14), wartet aber zuvor, dass ein anderer Prozess den kritischen Abschnitt verlässt (deshalb das `await` in Zeile (13)). Der Abschlusscode in Zeilen (16) und (17) setzt die Variable y auf 0 und zeigt damit das Verlassen des kritischen Abschnitts an und zum Schluss wird die want-Variable von Prozess i auf False gesetzt.

Einige einfache Beobachtungen zu Lamports Algorithmus sind:

- Wenn nur ein einzelner Prozess in den kritischen Abschnitt eintreten möchte, so werden nur konstant viele Operationen im Initialisierungs- und im Abschlusscode durchgeführt: Im Initialisierungscode 5 Operationen: (2), (3), (4), (8) und (9); im Abschlusscode zwei: (16) und (17).

- Der Algorithmus ist nicht Starvation-frei: Wenn ein Prozess in einen der beiden Abbruch-Zweige gerät, wird er von vorne beginnen. Das kann ihm beliebig oft passieren, da immer andere Prozesse den kritischen Abschnitt erreichen.
- Der Algorithmus bietet dem Prozess i zwei Möglichkeiten den kritischen Abschnitt zu erreichen. Entweder über (9), oder über (12).
- Ein weiterer interessanter Aspekt ist, dass Prozess i, falls er über (12) den kritischen Abschnitt erreicht, seine want-Variable auf False gesetzt hat, während er im kritischen Abschnitt ist.

Im folgenden wird gezeigt, dass Lamports Algorithmus den wechselseitigen Ausschluss garantiert und Deadlock-frei ist.

Satz 2.3.1. *Lamports Algorithmus garantiert den wechselseitigen Ausschluss.*

Beweis. Man sagt ein Prozess *erreicht den kritischen Abschnitt über Pfad* α, wenn er die `if`-Bedingung in Zeile (9) zu True auswertet (also direkt zu (15) springen darf). Erreicht ein Prozess den kritischen Abschnitt nachdem er die `if`-Bedingung in Zeile (12) zu True ausgewertet hat, so sagt man der Prozess *erreicht den kritischen Abschnitt über Pfad* β. Wie bereits erwähnt, sind dies die einzigen beiden Wege, den kritischen Abschnitt zu erreichen.

Zum Beweis der Aussage wird angenommen, dass (mindestens) zwei Prozesse gleichzeitig im kritischen Abschnitt sind. O.B.d.A. sei Prozess i der erste im kritischen Abschnitt und Prozess j der zweite. Es lassen sich zwei Fälle unterscheiden:

- Prozess i erreicht den kritischen Abschnitt über Pfad α. Wenn Prozess i die Abfrage in Zeile (9) durchläuft, muss gelten: $x = i$ und $y \neq 0$. Das impliziert, dass einer der folgenden beiden Fälle zutreffen muss:
  - Prozess j hat Zeile (3) noch nicht ausgeführt. Da Prozess j dann beim Ausführen von Zeile (4) $y \neq 0$ lesen wird, wird Prozess j seinen Versuch abbrechen und kann nicht zugleich mit Prozess i in den kritischen Abschnitt gelangt sein.
  - Prozess j hat Zeile (3) ausgeführt *bevor* Prozess i Zeile (3) ausführte. Dann gilt für Prozess j beim Ausführen von (9) $x \neq j$, da Prozess j den Wert von x nicht mehr setzen kann und Prozess i diesen Wert bereits auf $x = i$ verändert hat. Somit kann Prozess j den kritischen Abschnitt nicht über Pfad α erreichen. Prozess j kann den kritischen Abschnitt allerdings erst dann über den Pfad β erreichen, wenn Prozess i seine want-Variable auf False setzt (solange beendet Prozess j keinesfalls die `for`-Schleife). Da Prozess i seine want-Variable erst auf False setzt *nachdem* er den kritischen Abschnitt erreicht hat, ist es auch für diesen Fall unmöglich, dass beide Prozesse i und j zugleich im kritischen Abschnitt sind.
- Prozess i erreicht den kritischen Abschnitt über Pfad β. Nachdem Prozess i die `for`-Schleife in Zeile (11) durchlaufen hat, kann der Wert von y erst wieder verändert werden, wenn Prozess i den Abschlusscode ausführt: Da Prozess i gewartet hatte, dass alle want-Einträge False sind, muss jeder andere Prozess an einem `await`-Konstrukt hängen bleiben. Wenn ein anderer Prozess in Zeile (6) hängen bleibt, hat er y nicht verändert, und kann dies auch nicht mehr tun. Wenn ein Prozess erst später den restlichen Code verlässt, wird er in Zeile (4) immer zum `await` in Zeile (6) gelenkt, da $y \neq 0$ gilt. Falls ein anderer Prozess in Zeile (11) oder (13) hängen bleibt, dann hat er erst y verändert und *danach* seine want-Variable auf False gesetzt. Da Prozess i aber in Zeile (11) darauf wartet, dass alle want-Einträge auf False gesetzt sind, wird sich anschließend y nicht mehr verändern.

Das impliziert insbesondere, dass Prozess j den Wert von y nicht verändern kann, und deshalb nicht entlang Pfad β den kritischen Abschnitt erreichen wird, solange sich Prozess i im kritischen Abschnitt befindet.

Prozess j kann den kritischen Abschnitt auch nicht über Pfad α erreichen: Betrachte die Auswertungsfolge zum Zeitpunkt als Prozess i Zeile (12) ausführt. Wenn Prozess j noch nicht (4) ausgeführt hat, wird er in Zeile (6) warten, bis Prozess i den kritischen Abschnitt verlässt. Wenn Prozess j (4) ausgeführt hat, aber noch nicht (8) ausgeführt hat, dann kann Prozess i die `for`-Schleife nicht durchlaufen haben (da want[j]=True gilt). Wenn Prozess j (8) ausgeführt hat, dann kann unmöglich $y = i$ für Prozess i bei der Ausführung von Zeile (12) gegolten haben.

Da alle Fälle zum Widerspruch führen, ist die Annahme, dass Prozess i und j gleichzeitig im kritischen Abschnitt sind, falsch. Somit erfüllt Lamports Algorithmus den wechselseitigen Ausschluss. □

Für den Beweis der Deadlock-Freiheit wird die Notation $(i_1, j_1), (i_2, j_2), (i_3, j_3), \ldots$ für eine Berechnungssequenz verwendet, wobei für (i_k, j_k):

- i_k ist die Nummer des Prozesses, der einen Berechnungsschritt macht
- j_k ist die Codezeile, die Prozess i_k ausführt.
- es wird angenommen, dass für jeden anderen Prozess der momentane Codezeiger bekannt ist und außerdem ist die Belegung der Programmvariablen bekannt. Die gesamte Beschreibung dieser Belegung wird mit ‘Zustand k“ bezeichnet.

Satz 2.3.2. *Lamports Algorithmus ist Deadlock-frei.*

Beweis. Nehme an, die Aussage ist falsch. Sei $\mathcal{P} = (i_1, j_1), (i_2, j_2), (i_3, j_3), \ldots$ eine Berechnunssequenz, welche die Deadlock-Freiheit widerlegt.

- Da $\mathcal{P}$ die Deadlock-Freiheit widerlegt muss gelten: Es gibt $k_1 \in \mathbb{N}$, so dass für alle $k_1' \geq k_1$ gilt: $j_{k_1'} \neq 15$, d.h. kein Prozess erreicht den kritischen Abschnitt in der Folge $(i_{k_1'}, j_{k_1'}), (i_{k_1'+1}, j_{k_1'+1}), \ldots$ und für mindestens ein $l \geq k_1'$ gilt $j_l \in \{2, \ldots, 14\}$ d.h. mindestens ein Prozess möchte in den kritischen Abschnitt.
- Aufgrund der Fairness-Annahme folgt auch, dass irgendwann alle Prozesse ihren Abschlusscode beendet haben müssen, d.h. es gibt $k_2 \geq l$, so dass für alle $k_2' \geq k_2$ gilt: $j_{k_2'} \in \{1 - 14\}$.
- Da ab k_2 kein Prozess im Abschlusscode und im kritischen Abschnitt ist, kann der Wert von y nicht auf 0 gesetzt werden für alle Nachfolgezustände. Da zusätzlich Prozess i_l im Initialisierungscode ist, muss irgendwann y einen Wert verschieden von 0 annehmen, denn entweder ist $y \neq 0$ oder Prozess i_l führt irgendwann die Zuweisung für y in Zeile (8) aus. D.h. es gibt ein $k_3 \geq k_2$ ab dem die Variable y stets einen Wert ungleich 0 besitzt.
- Für alle Zustände ab Zustand k_3 gilt: Wenn der Programmzeiger eines Prozesses nicht direkt vor Zeile (8) steht, dann kann der Programmzeiger in allen Folgezuständen nicht mehr dort hin gelangen. Dies ist offensichtlich, denn ab Zustand k_3 müssen alle Ausführungen von Zeile (4) die `if`-Bedingung zu falsch auswerten (y ist ungleich 0 und kann nicht mehr auf 0 gesetzt werden).
- Aufgrund der Fairness-Bedingung werden alle Prozesse, deren Programmzeiger direkt vor Zeile (8) steht, den Code für (8) nach endlich vielen Schritten durchführen. D.h. es gibt

einen Zustand $m \geq k_3$, indem das letzte Mal Zeile (8) ausgeführt wird. D.h. Prozess i_m setzt y auf den Wert i_m. Danach müssen nach endlichen vielen Schritten alle Prozesse ihren want-Eintrag auf False setzen (ohne ihn wieder umsetzen zu können), wobei der Wert von y nicht verändert werden kann (alle Prozesse sind entweder in Zeile (1) oder in (5)-(6) oder in (9)-(14)).

- Der letzte Schritt impliziert, dass Prozess i_m die `for`-Schleife nach endlich vielen Schritten durchlaufen muss. Da stets $y = i_m$ gilt, muss Prozess i_m den kritischen Abschnitt nach endlich vielen Schritten betreten. Damit folgt der Widerspruch. □

2.3.2 Der Bakery-Algorithmus

Abschließend wird ein weiterer Algorithmus zum Mutual-Exclusion Problem für n Prozesse erläutert, der so genannte *Bakery-Algorithmus*. Seine besondere Eigenschaft ist, dass er nicht nur Starvation-frei ist, sondern auch die so genannte FIFO-Eigenschaft (oder auch „0-bounded waiting“ genannt) erfüllt. Zur Wiederholung nochmal die Definition der Starvation-Freiheit:

> *Starvation*-Freiheit: Wenn ein Prozess seinen kritischen Abschnitt betreten möchte, dann muss er ihn nach endlich vielen Berechnungsschritten betreten.

Diese Eigenschaft sichert dem Prozess jedoch nicht zu, wie lange er warten muss, bis er den kritischen Abschnitt betreten darf, bzw. wie viele andere Prozesse vor ihm wie oft den kritischen Abschnitt betreten dürfen.

Ein *wartender Prozess* bezeichnet in den folgenden Definitionen einen Prozess, der ein „busy-wait“ (aktives Warten) durchführt, also eine Bedingung ständig prüft, bis sie erfüllt ist, indem ein anderer Prozess einen Variablenwert verändert. (Hierfür wurden bisher meist die `await`-Konstrukte benutzt). Der Initialisierungscode in Mutual-Exclusion Algorithmen kann dementsprechend aufgeteilt werden: Der Code vor dem Warten (dieser muss wait-frei sein) wird dabei als „Doorway“ bezeichnet und der Code zum Warten wird als „Waiting“ bezeichnet. In Abbildung 2.6 wird diese Unterteilung für Mutual-Exclusion-Algorithmen illustriert.

Der Begriff des “bounded-waiting” und weitere damit zusammenhängende Begriffe sind wie folgt definiert:

Definition 2.3.3. *Ein Mutual-Exclusion Algorithmus erfüllt*

r-bounded waiting, *wenn für jeden Prozess i gilt: Wenn Prozess i den Doorway verlässt, bevor Prozess j (mit $j \neq i$) den Doorway betritt, dann betritt Prozess j den kritischen Abschnitt höchstens r Mal, bevor Prozess i den kritischen Abschnitt betritt.*

bounded waiting, *wenn es ein $r \in \mathbb{N}$ gibt, so dass der Algorithmus r-bounded waiting erfüllt.*

die FIFO-Eigenschaft, *wenn er 0-bounded waiting erfüllt. (FIFO = first-in-first-out)*

Beachte, dass alle Eigenschaften nicht Deadlock-Freiheit (und damit auch nicht die Starvation-Freiheit) implizieren. Informell bedeutet die FIFO-Eigenschaft, dass der erste Prozess, der den Doorway überschritten hat, auch als erster den kritischen Abschnitt betreten darf.

Lamports Algorithmus erfüllt kein bounded-waiting, da ein Prozess beliebig oft in den kritischen Abschnitt eintreten kann, bevor ein anderer Prozess in den kritischen Abschnitt darf. Die Tournament-Algorithmen erfüllen ebenso kein bounded-waiting.

Der im folgenden erläuterte Bakery-Algorithmus hingegen erfüllt 0-bounded waiting. Die Idee des Bakery-Algorithmus ist relativ einfach zu erklären. Jeder Prozess zieht im Doorway eine

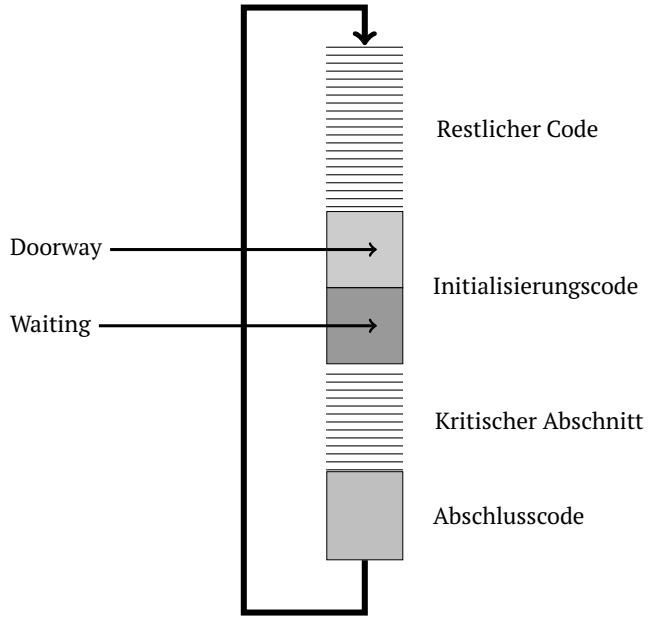


Abbildung 2.6: Unterteilung in Doorway und Waiting

Nummer, die größer (oder gleich s.u.) ist als alle bisher vergebenen Nummern. In den kritischen Abschnitt darf stets der Prozess mit der kleinsten Nummer. Der Name des Bakery-Algorithmus stammt daher, dass dieses Verfahren der Nummernvergabe in manchen Bäckereien durchgeführt wird (in Deutschland kennt man das Verfahren wohl eher von Ämtern). Zunächst wird die einfachste Variante des Bakery-Algorithmus erläutert. Der Code für Prozess i ist in Abbildung 2.7 dargestellt.

Initial: für $i = 1, \ldots n$ number[i]=0

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  number[i] := 1+max(number);
(3)  for j:=1 to n do
(4)      await number[j]=0 or (number[j],j) ≥lex (number[i],i)
(5)  Kritischer Abschnitt
(6)  number[i]=0
end loop
```

Abbildung 2.7: Einfache Variante des Bakery-Algorithmus

Hierbei meint $\geq_{lex}$ die lexikographische Ordnung, d.h. $(a, b) \geq_{lex} (c, d)$ genau dann, wenn $a > c$ oder ($a = c$ und $b \geq d$). Nimmt man für den einfachen Algorithmus an, dass das Maximum des number-Feldes atomar berechnet werden kann (einschließlich der Zuweisung), so kann man

nachweisen, dass er wechselseitigen Ausschluss garantiert, Starvation-frei ist und die FIFO-Eigenschaft erfüllt. Beachte, wenn das Maximum atomar berechnet wird, aber die Zuweisung erst im nächsten Schritt geschieht, dann erfüllt der Algorithmus keinen wechselseitigen Ausschluss: Zwei Prozesse i und j mit $i < j$ können beide denselben Wert für number[i] bzw. number[j] berechnen. Anschließend setzt Prozess j seine Ausführung fort und kommt durch Zeile (4), da die number[i] noch auf 0 gesetzt ist, d.h. j ist im kritischen Abschnitt. Wenn nun Prozess i weiterrechnet, dann kommt i auch durch Zeile (4), da number[j] = number[i] und $i < j$ gilt.

Allerdings ist selbst die atomare Berechnung des Maximums sehr unrealistisch. Im erweiterten Bakery-Algorithmus wird diese Annahme nicht benutzt, allerdings ist der Algorithmus etwas komplizierter. Der Code für Prozess i ist in Abbildung 2.8 abgebildet. Zusätzlich zeigt die Abbildung den Code für die Berechnung des Maximums.

Initial: für $i = 1, \ldots n$ number[i]=0, choosing[i]=False

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  choosing[i] := True;
(3)  number[i] := 1+maximum(number);
(4)  choosing[i] := False;
(5)  for j:=1 to n do
(6)      await choosing[j]=False;
(7)      await number[j]=0 or (number[j],j) ≥lex (number[i],i);
(8)  Kritischer Abschnitt
(9)  number[i]=0
end loop
```

Die Berechnung des Maximums erfolgt dabei folgendermaßen :

```
(1)  max := 0
(2)  for j:=1 to n do
(3)      current := number[j];
(4)      if max < current then max := current;
(5)  number[i] := 1+max;
```

Abbildung 2.8: Erweiterte Variante des Bakery-Algorithmus

Mithilfe des choosing-Feldes wird der Eintritt und der Austritt aus dem Doorway markiert, Die zusätzliche `await`-Abfrage auf dem choosing-Feld dient dazu, abzuwarten bis der jeweilige Prozess im Doorway seinen number-Wert gesetzt hat.

Im folgenden wird bewiesen, dass der erweitere Bakery-Algorithmus wechselseitigen Ausschluss garantiert, Starvation-frei ist, und die FIFO-Eigenschaft erfüllt. Hierfür werden zunächst einige Hilfssätze bewiesen.

Es lässt sich einfach nachprüfen, dass gilt:

Lemma 2.3.4. *Wenn der Wert von number[k] nicht geändert wird, während Prozess i das Maximum berechnet, dann ist der Wert number[i] anschließend größer als der Wert von number[k].*

Ein Prozess i *befindet sich im Doorway*, wenn sein Programmzeiger zwischen den Zeilen (2) und (4) ist. Er *ist in der Bäckerei*, wenn er im Waiting oder im Kritischen Abschnitt ist (Zeilen (5)-(8)). Eine direkte Konsequenz aus Lemma 2.3.4 ist:

Lemma 2.3.5. *Wenn Prozess i und Prozess k beide in der Bäckerei sind und i in die Bäckerei eintritt, bevor k in den Doorway eintritt, dann gilt number[i] $<$ number[k].*

Das nächste Lemma impliziert wechselseitigen Ausschluss:

Lemma 2.3.6. *Wenn Prozess i im kritischen Abschnitt ist und Prozess k in der Bäckerei ist, dann gilt (number[i],i) $<_{lex}$ (number[k],k).*

Beweis. Sei T_6^i der Schritt, in dem Prozess i in Zeile (6) durch das `await`-Statement für $j = k$ hin durchkommt (d.h. Prozess i liest zum letzten Mal choosing[k]). Analog sei T_7^i der Schritt, indem Prozess i das letzte Mal number[k] gelesen hat. Dann muss gelten T_6^i liegt vor T_7^i (notiert als $T_6^i < T_7^i$).
Für Prozess k seien T_2^k, T_3^k, T_4^k jeweils die Schritte, in denen er die Programmzeilen (2), (3), (4) zum letzten Mal abgearbeitet hat. Es muss gelten $T_2^k < T_3^k < T_4^k$.
Im Schritt T_6^i hatte choosing[k] den Wert False und zu den Schritten T_2^k, T_3^k und T_4^k hatte choosing[k] den Wert True. Daraus ergibt sich, dass einer der beiden folgenden Fälle gelten muss:

1. $T_6^i < T_2^k$: In diesem Fall impliziert Lemma 2.3.5, dass number[i] < number[k] gelten muss, und die Aussage ist bewiesen.
2. $T_4^k < T_6^i$: In diesem Fall gilt $T_3^k < T_4^k < T_6^i < T_7^i$, d.h. zum Zeitpunkt als Prozess i Zeile (7) zum letzten Mal ausführt und number[k] liest, kann dessen Wert nicht 0 sein (er wurde zum Zeitpunkt T_3^k gesetzt!). Da die `await`-Bedingung in Zeile (7) wahr ist, muss (number[i],i) $<_{lex}$ (number[k],k) gegolten haben.

□

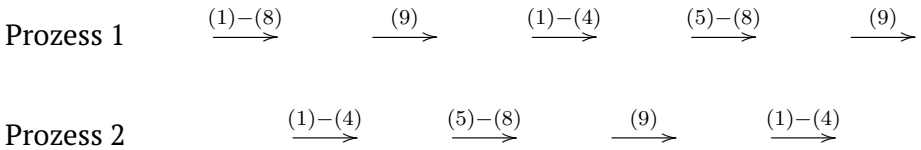
Satz 2.3.7. *Der erweiterte Bakery-Algorithmus garantiert wechselseitigen Ausschluss, ist Starvation-frei, und erfüllt die FIFO-Eigenschaft.*

Beweis. Mutual-Exclusion folgt direkt aus Lemma 2.3.6, da sonst ein Widerspruch hergeleitet werden kann. Die FIFO-Eigenschaft folgt aus den Lemmas 2.3.5 und 2.3.6. Der Beweis der Starvation-Freiheit ist einfach: Betrachte zunächst Deadlock-Freiheit: Wenn in einer unendlich langen Berechnungsfolge irgendwann kein Prozess mehr in den kritischen Abschnitt gelangt, aber mindestens ein Prozess in der Bäckerei ist, dann gibt es einen Zeitpunkt zudem kein Prozess mehr in den Doorway und die Bäckerei eintritt. Ab diesem Zeitpunkt muss ein Prozess die `for`-Schleife durchlaufen können. Starvation-Freiheit folgt aus der Deadlock-Freiheit und der FIFO-Eigenschaft. □

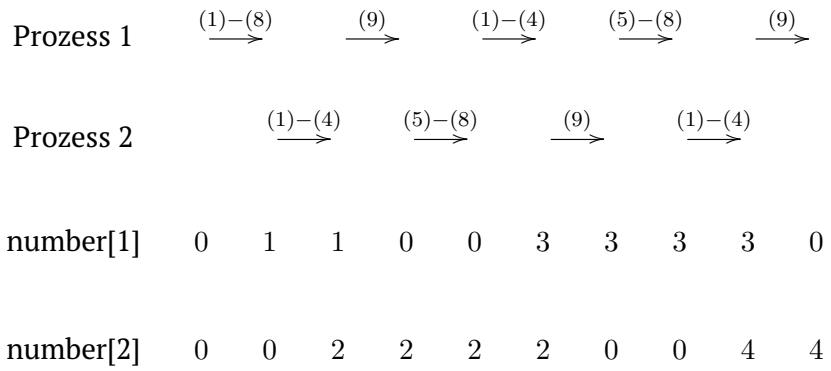
Neben diesen Vorteilen des Bakery-Algorithmus gibt es einige Nachteile:

Nachteil 1: Der Algorithmus ist nicht schnell für den Fall, dass nur ein Prozess in den kritischen Abschnitt will. Er muss im Initialisierungscode $O(n)$ Berechnungsschritte durchführen.

Nachteil 2: Der vorgestellte Algorithmus benötigt unbegrenzt große Zahlen für das number-Feld. Das lässt sich leicht überlegen, wenn man nur zwei Prozesse betrachtet: Prozess 1 und Prozess 2 besuchen abwechselnd den kritischen Abschnitt, wobei die Ausführung wie folgt durchmischt ist:



Wichtig hierbei ist, dass Zeile (9) des einen Prozesses immer erst dann ausgeführt wird, nachdem der andere Prozess das Maximum berechnet hat. Notiert man die Werte von number hinzu, sieht man den Effekt:



Es gibt jedoch (kompliziertere) Varianten des Bakery-Algorithmus, die mit begrenzter Zahlgröße auskommen (z.B. der so genannte Black-White-Bakery-Algorithmus).

Als letzte Anmerkung zum Bakery-Algorithmus erwähnen wir noch, dass man auch bei der Berechnung des Maximums der number-Werte aufpassen muss. Ersetzt man der vorherigen Algorithmus durch den folgenden Code:

(1) maxpos := i
(2) **for** j:=1 **to** n **do**
(3) **if** number[maxpos] $<$ number[j] then maxpos := j
(4) number[i] := 1+number[maxpos]:

dann wird der erweiterte Bakery-Algorithmus falsch, da wechselseitiger Ausschluss nicht länger garantiert. Ein Gegenbeispiel kann mit drei Prozessen 1,2,3 konstruiert werden. Nehme an, dass Prozesse 2 und 3 zunächst verzahnt ihren neuen number-Wert berechnen, sodass danach gilt number[2] = 1 und number [3] = 1. Anschließend sei Prozess 2 im kritischen Abschnitt, während Prozess 3 darauf wartet, dass dieser ihn verlässt (d.h. der number[3]-Wert auf 0 gesetzt wird). Wenn nun zunächst Prozess 1 alle Schritte der maximum-Berechnung durchführt außer Schritt 4, so gilt maxpos=2. Wenn nun Prozess 2 den kritischen Abschnitt verlässt, number[2] = 0 setzt und Prozess 3 in den kritischen Abschnitt wechselt und anschließend Prozess 1 Zeile (4) der maximum-Berechnung ausführt, so setzt diese Ausführung number[1] = 1+number[2] = 1+0 =

1. Anschließend kann Prozess 1 in den kritischen Abschnitt eintreten, obwohl sich Prozess 3 noch in selbigem befindet.

2.4 Drei Komplexitätsresultate zum Mutual-Exclusion Problem

In diesem Abschnitt werden einige Aussagen zur Komplexität von Mutual-Exclusion Algorithmen vorgestellt. Größtenteils wird auf die entsprechenden Beweise verzichtet. Man beachte, dass diese Resultate weiterhin vom gleichen Modell ausgehen, d.h. es gibt nur atomare Lese- und Schreibbefehle für den gemeinsamen Speicher.

Im Jahr 1980 haben J. Burns und N.A.Lynch eine untere Schranke für den benötigten Platz an gemeinsamen Speicher nachgewiesen:

Theorem 2.4.1. *Jeder Deadlock-freie Mutual-Exclusion Algorithmus für n Prozesse benötigt mindestens n gemeinsam genutzte Speicherplätze.*

Tatsächlich kann man nachweisen, dass diese n Speicherplätze auch tatsächlich ausreichen, wobei ein solcher Speicherplatz nur Bits also (wahr oder falsch) speichern können muss:

Theorem 2.4.2. *Es gibt einen Deadlock-freien Mutual-Exclusion Algorithmus für n Prozesse der n gemeinsame Bits verwendet.*

Der Beweis besteht aus dem so genannten Ein-Bit-Algorithmus, der in Abbildung 2.9 dargestellt ist.

Er wurde sowohl von J.E.Burns im Jahr 1981 als auch von L.Lamport im Jahr 1986 entwickelt. Man kann für diesen Algorithmus die Deadlock-Freiheit und die Garantie des wechselseitigen Ausschlusses nachweisen. Die Vorgehensweise des Algorithmus aus Sicht des Prozesses mit der Identifikation i lässt wie folgt erläutern:

In den Zeilen (2)-(10) setzt Prozess i seinen want-Eintrag auf True, um anzuzeigen, dass er den kritischen Abschnitt betreten möchte. Dabei prüft er die want-Einträge aller Prozesse, die eine kleinere Identifikationsnummer haben. Sind diese Einträge alle falsch (und sein eigener Eintrag wahr), so verlässt er die Schleife erfolgreich und fährt mit Zeile (11) fort. Ist einer dieser want-Einträge wahr, so setzt Prozess i seinen eigenen Eintrag auf False, wartet bis der andere Eintrag falsch wird und startet anschließend die komplette Schleife neu. In Zeilen (11)-(12) prüft Prozess i die want-Einträge aller Prozesse, die eine größere Identifikationsnummer als er haben. Sind diese alle falsch, dann darf Prozess i in den kritischen Abschnitt eintreten. Er hat dann alle want-Einträge mindestens einmal als falsch gelesen, seitdem er in die Schleife in Zeile (2) eingetreten war. Im Abschlusscode (Zeile (14)) setzt Prozess i seinen want-Eintrag auf falsch.

Es sei noch anzumerken, dass der Algorithmus keine Starvation-Freiheit erfüllt, nicht schnell ist (wenn nur ein Prozess in den kritischen Abschnitt will, so muss dieser alle n Bits lesen), und auch nicht symmetrisch ist, da Prozesse mit kleineren Prozessidentifikationen bevorzugt in den kritischen Abschnitt eintreten können (Z.B. wird Prozess 1 die **`repeat`**-Schleife immer nur einmal durchlaufen).

Das nächste Theorem (von R. Alur und G.Taubenfeld) besagt, dass Prozesse stets beliebig lang „warten“ müssen, bevor sie in den kritischen Abschnitt eintreten können:

Initial: für $i = 1, \ldots n$ want[i]= False,

Programm des i. Prozesses

```
loop forever
(1)   restlicher Code
(2)   repeat
(3)     want[i]:= True;
(4)     local := 1:
(5)     while (want[i] = True) and (local < i) do
(6)       if want[local] = True then
(7)         want[i] := False;
(8)         await want[local] = False;
(9)       local := local + 1;
(10)  until want[i] = True;
(11)  for local:= i+1 to n do
(12)    await want[local] = False;
(13)  Kritischer Abschnitt
(14)  want[i] = False
end loop
```

Abbildung 2.9: Ein-Bit Algorithmus

Theorem 2.4.3. *Unter der Annahme, dass es nur atomare Lese- und Schreiboperationen gibt, gilt: Es gibt keinen (Deadlock-freien) Mutual-Exclusion Algorithmus für 2 (oder auch n) Prozesse, der eine obere Schranke hat für die Anzahl an Speicherzugriffen (des gemeinsamen Speichers), die ein Prozess ausführen muss, bevor er den kritischen Abschnitt betreten darf.*

Beweis. Sei $\mathcal{M}$ ein Mutual-Exclusion Algorithmus für 2 Prozesse. Seien P_1 und P_2 die beiden Prozesse. Der *Berechnungsbaum* T_M von M ist ein binärer Baum dessen Knoten den Zuständen des Programms entsprechen. Der Baum ist durch die folgenden Regeln definiert:

- Die Wurzel von T_M ist der Initialzustand, wobei der restliche Code ignoriert wird, d.h. beide Prozesse P_1 und P_2 sind direkt vor dem Eintritt in den Initialisierungscode.
- Das linke Kind eines Knotens ist der Zustand des Knotens nach Ausführung des nächsten Schritts von P_1 auf dem gemeinsamen Speicher
- Das rechte Kind eines Knotens ist der Zustand des Knotens nach Ausführung des nächsten Schritts von P_2 auf dem gemeinsamen Speicher
- Ein Knoten ist ein Blatt genau dann, wenn einer der beiden Prozesse in seinem kritischen Abschnitt ist.

D.h. der Berechnungsbaum erfasst alle Zustände nach Eintritt in den Initialisierungscode bis zum Erreichen des kritischen Abschnitts, wobei „prozessinterne Schritte“, die nur lokale Variablen lesen oder ändern, nicht explizit dargestellt werden. Die Knoten des Berechnungsbaumes werden mit 1, 2 oder (1 und 2) markiert, wobei folgende Bedingungen eingehalten werden müssen:

- Ein Blatt ist mit 1 oder 2 markiert, je nach dem welcher Prozess im kritischen Abschnitt ist
- Ein innerer Knoten ist mit den Markierungen seiner beiden Kinder markiert (als Menge gesehen, daher mit 1, 2 oder (1 und 2)

Zwei Knoten v und w seien *ähnlich bzgl.* P_i (geschrieben als $v\langle P_i\rangle w$) gdw.

1. die Folge der Schritte, die Prozess P_i auf dem Pfad von der Wurzel zu v macht, genau die gleiche Folge ist, wie die Folge der Schritte, die Prozess P_i auf dem Pfad von der Wurzel zu w macht, und
2. Für die zu v und w zugehörigen Zustände gilt: Die gemeinsam benutzten Variablen sowie die lokalen Variablen von P_i sind für v und w gleich belegt.

Für den Beweis des Theorems genügt es zu zeigen, dass es für jedes $n > 0$ und $i \in \{1, 2\}$ ein Blatt v mit Markierung i gibt, so dass auf dem Pfad von der Wurzel bis zu v mehr als n Schritte für Prozess P_i ausgeführt werden. Dies reicht aus, da dadurch sichergestellt wird, dass man keine obere Schranke für die (aktive) Wartezeit bis zum Betreten des kritischen Abschnitts angeben kann.

Wenn es einen unendlich langen Pfad in T_M gibt, der unendlich viele Knoten enthält, die mit i markiert sind, und Prozess i macht auf dem Pfad unendliche viele Schritte, dann folgt sofort, dass obige Aussage gilt, indem man einen Präfix des Pfades wählt, der mehr als n Schritte von P_i enthält und anschließend den Präfix zu einem Blatt verlängert, in dem Prozess i im kritischen Abschnitt ist. Ab jetzt werden nur noch solche Bäume T_M betrachtet, in denen solche Pfade nicht existieren. Als Referenz sei dies die Annahme A. Da der kritische Abschnitt erreicht werden muss (Deadlock-Freiheit), gilt: Es gibt einen Knoten v der mit 1 und 2 markiert ist, so dass ein Kind von v mit 1 und das andere Kind von v mit 2 markiert ist. Sei v_1 das linke und v_2 das rechte Kind von v und seien e_1 bzw. e_2 die Berechnungsschritte von v zu v_1 bzw. von v zu v_2. Es werden zwei Fälle betrachtet, und gezeigt, dass beide Fälle zum Widerspruch führen. Daher folgt, dass Annahme A nie erfüllt ist, und damit ist das Theorem bewiesen.

Fall 1: v_1 ist mit 1 markiert, v_2 ist mit 2 markiert. Der Baum ist in Abbildung 2.10 skizziert. Der

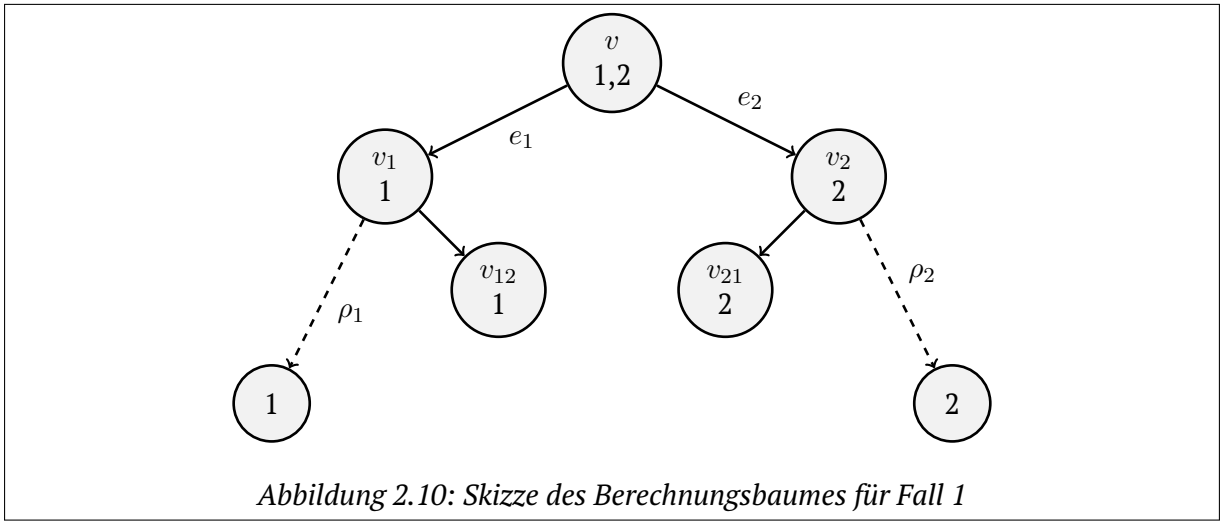


Abbildung 2.10: Skizze des Berechnungsbaumes für Fall 1

linkeste Pfad beginnend von v_1 muss aufgrund von Annahme A endlich lang sein, da auf diesem Pfad nur P_1 Berechnungsschritte durchführt. Dieser Pfad muss zudem mit einem Blatt enden, welches mit 1 markiert ist. Sei ρ_1 die Folge der Berechnungsschritte von v_1 zum Blatt (diese kann

man sich als Kantenmarkierung des Pfades vorstellen). Für den rechtesten Pfad beginnend ab v_2 gilt die analoge Folgerung nur für Prozess 2: Dieser Pfad ist endlich, und endet mit einem Blatt mit Markierung 2. Sei ρ_2 die entsprechende Berechnungsfolge von v_2 zum rechtesten Blatt. Die folgenden Fälle können dabei unterschieden werden:

- Der Schritt e_1 ist eine Lese-Operation: Dann sind die Werte der gemeinsamen Variablen in den Zuständen v und v_1 identisch und somit $v\langle P_2\rangle v_1$. Daraus folgt auch sofort $v_2\langle P_2\rangle v_{12}$. Da ρ_2 nur Schritte von P_2 durchführt und in einem Blatt mit Markierung 2 endet, folgt, dass v_{12} auf seinem rechtesten Pfad auch in einem Blatt mit Markierung 2 enden muss. Dies ist ein Widerspruch, da v_{12} nur mit 1 markiert ist.
- e_2 ist eine Lese-Operation: Dann kann symmetrisch zum vorhergehenden Fall argumentiert werden.
- e_1 und e_2 sind Schreibe-Operationen in verschiedene Variablen: Dann gilt für $i = 1, 2$: $v_{12}\langle P_i\rangle v_{21}$. Dann ist es aber unmöglich, dass v_{12} nur mit 1 und v_{21} nur mit 2 markiert ist.
- e_1 und e_2 sind Schreibe-Operationen in die gleiche Variable. Dann gilt $v_2\langle P_2\rangle v_{12}$, da Ausführen von e_2 im Zustand v_1 die Schreiboperationen von e_1 rückgängig macht. Nun kann genau wie im ersten Fall argumentiert werden.

Fall 2: v_1 ist mit 2 und v_2 ist mit 1 markiert. Der Berechnungsbaum ist in Abbildung 2.11 skizziert. Da P_2 nicht im kritischen Abschnitt in v ist, ist P_2 auch in v_1 nicht im kritischen Abschnitt

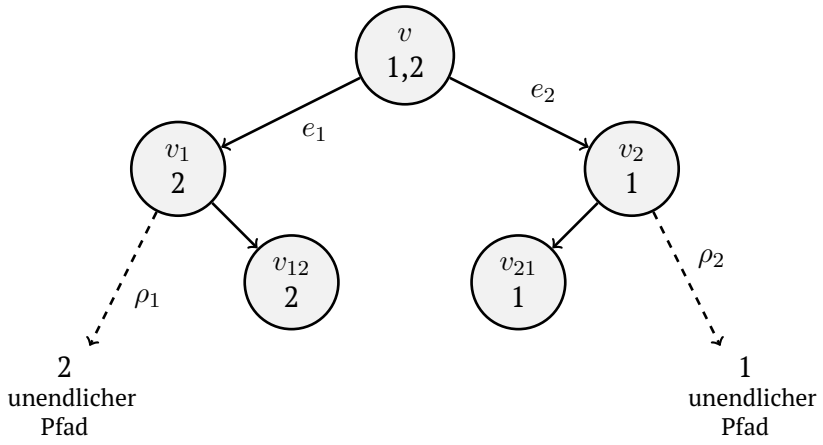


Abbildung 2.11: Skizze des Berechnungsbaumes für Fall 2

(auf dem Weg von v zu v_1 führt P_1 einen Schritt durch!). Das impliziert, dass jeder Pfad beginnend von v_1, der in einem Blatt endet, mindestens einen Berechnungsschritt für P_2 durchführen muss. Daher muss der linkeste Pfad beginnend von v_1 unendlich lang sein (dort werden keine Schritte für P_2 durchgeführt!). Sei ρ_1 die Sequenz der Schritte auf diesem Pfad. Analog kann für v_2 geschlossen werden: Der rechteste Pfad ist unendlich lang. Sei ρ_2 die entsprechende Folge von Berechnungsschritten. Unterscheidung in verschiedene Fälle:

- e_1 ist eine Leseoperation. Dann gilt $v_2\langle P_2\rangle v_{12}$. Daher kann ρ_2 auch von v_{12} aus ausgeführt werden. Da ρ_2 nur Berechnungen für P_2 durchführt erhält man einen unendlich langen Pfad, der ausschließlich aus P_2-Schritten besteht und in dem alle Knoten mit 2 markiert sind. Dies verletzt Annahme A und ergibt einen Widerspruch.
- e_2 ist eine Leseoperation: Symmetrisch zum vorherigen Fall.

- e_1 und e_2 sind Schreiboperationen in verschiedene Variablen. Dann gilt für $i = 1, 2$: $v_{12}\langle P_i\rangle v_{21}$. Das ist unmöglich, da v_{12} und v_{21} jeweils mit unterschiedlichen Zahlen markiert sind.
- e_1 und e_2 sind Schreiboperationen in die gleiche Variable. Dann gilt $v_{12}\langle P_2\rangle v_2$, da die Schreiboperation e_1 von v zu v_1 durch die Operation e_2 quasi wieder unsichtbar gemacht wird. Nun kann genauso wie im ersten Fall argumentiert werden. □

2.5 Stärkere Speicheroperationen

In diesem Abschnitt wird das bisherige Modell, in dem es nur atomare Lese- und Schreibbefehle auf den gemeinsamen Speicher gibt, verlassen. Es werden verschiedene andere so genannte primitive Operationen dargestellt, deren atomare Ausführung durch manche Hardwarearchitekturen unterstützt werden. Deadlock-freie Mutual-Exclusion-Algorithmen sind mit diesen stärkeren Primitiven relativ einfach zu implementieren.

Die weiteren atomaren (oder auch unteilbaren) Operationen werden als Funktionen definiert, die als erstes Argument jeweils ein gemeinsam benutztes Speicherregister (dessen Adresse) erhalten, welches explizit durch Angabe des Typs **Register** für das Argument gekennzeichnet wird. Je nach Operationen erhalten die Funktionen weitere Argumente (von den Typen **Register** (gemeinsames Speicherregister), **Lokales Register** (lokales Speicherregister), **Wert** (Ein Wert wie True, 1, usw), **Funktion** (eine Seiteneffekt-freie Funktion auf Werten)). Auch der Typ des zurück gegebenen Ergebnisses wird angegeben (mittels **`returns`**:Typ), wenn die Funktion einen Wert zurück gibt. Für die Ausführung der Funktionen gilt: Alle Schritte nach dem Aufruf bis zum Verlassen der Funktion werden atomar durchgeführt, d.h. damit diese Schritte *nicht* mit anderen Schritten (von anderen Prozessen) durchmischt werden, wird der gesamte Funktionsaufruf als *ein* Berechnungsschritt (gemäß Definition 1.5.1) definiert.

Im folgenden wird keine explizite Trennung zwischen Registeradressen und dem Wert von Registern verwendet. Wie vorher wird der Registername verwendet, um auf den Wert zuzugreifen (was gerade dem atomaren Lesen entspricht). Für das Beschreiben eines Registers wird wie bisher die Zuweisung $x :=$ Wert verwendet.

Zunächst können die bekannten Operationen des Lesens und Schreiben in diesem Muster als Funktionen definiert werden:

- *read*(r) liest den Wert des Registers mit Adresse r und gibt den gelesenen Wert zurück, d.h.

```
function read(r : Register) returns : Wert
  return(r) :
end function
```

- *write*(r, v) erhält ein Register und einen Wert und schreibt den Wert ins Register, d.h.

```
function write(r : Register, v : Wert)
  r := v
end function
```

  Da die *write*-Operation kein Ergebnis zurückliefert, entfällt der **`returns`**-Teil.

Beachte, dass bisher Lesen und Schreiben nicht explizit durch Funktionsaufrufe in den Algorithmen notiert wurden, z.B. könnte man dies nun mit den Funktionen *read* und *write* durchfüh-

ren. Z.B. statt **if** $x = 0$ **then** ... könnte man **if** *read*$(x) = 0$ **then** ... und statt $x := 10$ könnte man *write*$(x, 10)$ schreiben. Aufgrund der Übersichtlichkeit wird aber auch weiterhin darauf verzichtet. Die Funktionsnotation zeigt aber z.B. deutlich auf, dass die Zuweisung $x := y$, wobei x und y Register sind, nicht atomar ist, sondern zwei atomare Operationen enthält: *write*$(x,$ *read*$(y))$.

Nun werden einige neue atomare Operationen definiert:

- Die *test-and-set*-Operation *testet* den Wert eines Registers (was dem Lesen entspricht) und setzt gleichzeitig einen neuen Wert (was dem Schreiben entspricht). Genauer ist der Aufruf *test-and-set*(r, v), wobei r ein Register ist und v ein Wert ist. Der Wert v wird dem Register zugewiesen und der *alte* Wert des Registers wird von der Funktion zurückgegeben, d.h.

  function *test-and-set*(r : **Register**, v : **Wert**) **returns** : **Wert**
  $temp := r;$
  $r := v;$
  return$(temp);$
  end function

  In manchen anderen Definitionen bzw. in manchen Implementierungen der test-and-set-Operation, darf der Wert v nur den Wert 1 annehmen (und damit kann auf das zweite Argument der Funktion *test-and-set* verzichtet werden). Der Speicherplatz ist dann ein Bit (welches nur die Werte 0 oder 1 annehmen kann).

- *swap*(r, l) erhält ein Register und ein lokales Register und tauscht die Werte beider Register (atomar) aus, d.h.

  function *swap*(r : **Register**, l : **Lokales Register**)
  $temp := r;$
  $r := l;$
  $l := temp;$
  end function

  Die *swap*-Operation wird manchmal auch als *fetch-and-store*-Operation bezeichnet.

- *fetch-and-add*(r, v) erhält ein Register r und einen Wert v. Der Wert von r wird um den Wert v erhöht. Der alte Wert des Registers r wird zurückgegeben, d.h.

  function *fetch-and-add*(r : **Register**, v : **Wert**) **returns** : **Wert**
  $temp := r;$
  $r := temp + v;$
  return$(temp);$
  end function

  Auch von dieser Operation gibt es Varianten, die auf den Wert als Argument verzichten und stets genau 1 hinzu addieren (diese Operation wird auch manchmal als *fetch-and-increment* bezeichnet).

- *read-modify-write*(r, f) erhält ein Register und eine Funktion. Die Funktion wird auf den Wert des Registers angewendet und anschließend der entstandene Wert als neuer Wert

des Registers übernommen. Die Funktion liefert den alten Wert des Registers zurück. D.h.

```
function read-modify-write(r : Register, f : Funktion) returns : Wert
  temp := r;
  r := f(temp);
  return(temp);
end function
```

Beachte, dass die Operationen *read*, *write*, *test-and-set* und *fetch-and-add* auch alle mittels *read-modify-write* ausgedrückt werden können, indem man passende Funktionen als Argument für *read-modify-write* verwendet. Beachte auch, dass die Funktion f Seiteneffekt-frei sein sollte. Insbesondere wäre es problematisch, wenn man für die Funktion eine atomare Operation (z.B. *fetch-and-increment*) einsetzen würde, da dann zwei atomare Operationen zu einer neuen atomaren Operation per Definition gemacht würde. Deswegen sind Funktionen, die den Speicher manipulieren, als Argument für *read-modify-write* verboten.

- *compare-and-swap*(r, old, new) erwartet ein Register r und zwei Werte old und new. Wenn der Wert des Registers gleich ist zu old, dann wird r auf den Wert new gesetzt und True wird als Ergebnis der Funktion zurückgegeben. Andernfalls wird der Wert von r nicht verändert und False zurückgegeben, d.h.

```
function compare-and-swap(r : Register, old : Wert, new : Wert)
                                                  returns : Wert
  if r = old then
     r := new;
     return(True);
  else
     return(False);
end function
```

- *sticky-write*(r, v) erwartet ein Register und einen Wert. Der Initialwert von r ist dabei undefiniert (geschrieben als $\perp$). Wenn der Wert des Registers r undefiniert (also $\perp$) oder gleich zu v ist, dann wird der Wert auf v gesetzt. Die Operation war in diesem Fall erfolgreich und liefert True zurück. Andernfalls wird False zurückgegeben. Die *sticky-bit-write*-Operation beschreibt den Spezialfall in dem r nur die Wert 0, 1 und $\perp$ annehmen kann.

- *move*(r_1, r_2) erwartet zwei Register und setzt den Wert des ersten Registers auf den Wert des zweiten Registers, d.h.

```
function move(r1 : Register, r2 : Register)
  temp := r2;
  r1 := temp;
end function
```

Die Zuweisung $x := y$ für zwei gemeinsame Variablen (die im Modell des atomaren Lesens und Schreibens verboten ist), ist daher gerade eine *move*-Operation.

- *shared-swap*(r_1, r_2) erwartet zwei Register und tauscht die Werte der Register aus, d.h.

```
function shared-swap(r1 : Register, r2 : Register)
  temp1 := r1;
  temp2 := r2;
  r1 := temp2;
  r2 := temp1;
end function
```

Die Implementierung solcher atomarer Operationen ist nicht-trivial. Im folgenden wird angenommen, dass die Hardware solche Operationen zur Verfügung stellt und atomar durchführt.

Es bleibt zu definieren, wie genau die Objekte, d.h. die Datenstrukturen aussehen (z.B. ist es eher unsinnig ein Register mit allen obigen Operationen als Datenstruktur anzunehmen). Daher werden nun einige solcher Objekte definiert, d.h. (nebenläufige) Datenstrukturen mit zugehörigen atomaren Operationen:

Atomares Register: Ein gemeinsames Speicherregister, das (atomare) *read*- und *write*-Operationen unterstützt.

Test-and-set-Objekt: Ein gemeinsames Speicherregister, das (atomare) *write*- und *test-and-set*-Operationen unterstützt. Hierbei wird oft die Operation *reset* hinzugefügt, die gerade dem Schreiben einer 0 entspricht.

Test-and-set-Bit: Ein spezielles Test-and-set-Objekt: Nur 0 und 1 sind als Werte des Registers erlaubt, wobei die *test-and-set*-Operation nur eine 1 schreiben darf und die *write*-Operation nur eine 0 schreiben darf (also nur ein *reset* unterstützt). Fast alle Hardwarearchitekturen unterstützen Test-and-set-Bits.

Test-and-test-and-set-Objekt Ein Test-and-set-Objekt, das zusätzlich die atomare *read*-Operation unterstützt.

Fetch-and-increment-Objekt Ein gemeinsames Speicherregister, das die *fetch-and-increment*-, die *write*- und die *read*-Operation unterstützt.

Fetch-and-add-Objekt: Ein gemeinsames Speicherregister, das die *fetch-and-add*-, die *write*- und die *read*-Operation unterstützt.

Swap-Objekt: Ein gemeinsames Speicherregister, das die *swap*-Operation zwischen dem Register und jedem beliebigen lokalen Register unterstützt.

Read-Modify-Write-Objekt: Ein gemeinsames Speicherregister, das die *read-modify-write*-, die *write*- und die *read*-Operation unterstützt.

Compare-and-swap-Objekt: Ein gemeinsames Speicherregister, das die *compare-and-swap*-, die *write*- und die *read*-Operation unterstützt.

Sticky-Bit: Ein gemeinsames Speicherregister, das die *read* und die sticky-bit-write-Operation unterstützt. Ein solches Bit kann drei Werte haben: 0, 1 oder $\perp$.

Move-Objekt : Mehrere gemeinsame Speicherregister, die untereinander paarweise die *move*-Operation unterstützen (zusätzlich *write* und *read*).

Shared-swap-Objekt : Mehrere gemeinsame Speicherregister, die untereinander paarweise die *shared-swap*-Operation unterstützen (zusätzlich *write*f und *read*).

Diese Aufzählung von Objekten ist nicht vollständig, es gibt weitere Objekte, die in der Literatur zu finden sind.

2.5.1 Mutual-Exclusion mit Test-and-set-Bits

In diesem Abschnitt werden Mutual-Exclusion-Algorithmen vorgestellt, die Test-and-set-Bits als Objekte verwenden. Da diese Objekte nur 0 und 1 als Werte und die atomaren Operationen *reset* (Setzen auf 0) und *test-and-set* mit dem Wert 1 unterstützen, wird *test-and-set(r)* anstelle von *test-and-set(r,1)* verwendet.

Test-and-set-Bits werden oft auch als *Lock* (Sperre) bezeichnet. Die beiden Operationen *test-and-set* und *reset* werden dann mit *lock* und *unlock* bezeichnet. Wenn ein Prozess bemerkt, dass ein Lock gesetzt ist, wird oft so verfahren, dass der Prozess in einer Schleife darauf wartet, dass er den Lock setzen darf, d.h. er führt eine Schleife der Form **`while`** *test-and-set*(lock)=1 **`do skip`**; aus. D.h. der Prozess „kreiselt“ (engl. spin) um den Lock. Deshalb werden test-and-set-Bits auch oft als *Spinlocks* bezeichnet.

Mit dieser Technik ist ein Mutual-Exclusion-Algorithmus einfach zu implementieren. Abbildung 2.12 zeigt den Code für Prozess i für diesen einfachen Algorithmus.

Initial: Test-and-set-Bit x hat Wert 0

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  await (test-and-set(x)=0);
(3)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Abbildung 2.12: Einfacher Mutual-Exclusion Algorithmus mit Test-and-set-Bit

Der Algorithmus garantiert wechselseitigen Ausschluss und ist Deadlock-frei. Beides ist einfach zu sehen: Nehme an, zwei Prozesse i und j sind gleichzeitig im kritischen Abschnitt. Sei Prozess i der erste im kritischen Abschnitt. Dann hat Prozess i beim letzten Ausführen von (2) den Wert von x auf 1 gesetzt. Anschließend muss Prozess j vor dem Eintreten in den kritischen Abschnitt Zeile (2) ausgeführt haben, so dass *test-and-set(x)* den Wert 0 geliefert hat. Das ist aber nur der Fall, wenn x den Wert 0 hat, was unmöglich ist, da x erst wieder von Prozess i auf den Wert 0 gesetzt werden kann (mittels *reset*), nachdem Prozess i den kritischen Abschnitt verlassen hat. D.h. wechselseitiger Ausschluss ist garantiert. Zur Deadlock-Freiheit: Da x den Wert 1 immer nur für endlich viele Schritte besitzt (nur solange ein Prozess im kritischen Abschnitt ist und noch nicht Zeile (4) ausgeführt hat), wird immer ein Prozess die **`await`**-Bedingung nach endlich vielen Schritten passieren. Der einfache Algorithmus hat jedoch auch Nachteile: Er erfüllt die Starvation-Freiheit nicht, da ein wartender Prozess von einem anderen Prozess immer wieder „überholt“ werden kann. Ein anderer Nachteil ist, dass die *test-and-set*-Operationen beim Warten ständig den Wert von x neu setzt (immer wieder auf 1), auch dann, wenn 1 das Ergebnis ist. Der Nachteil dabei ist, dass das Verändern des gemeinsamen Speichers aus Hardware-Sicht aufwändig ist. Z.B. müssen die Caches von verschiedenen Prozessoren stets neu aktualisiert werden.

Eine (eher praktische, weniger theoretische) Verbesserung des Algorithmus kann erzielt werden, indem man Test-and-test-and-set-Objekte verwendet und dadurch versucht das Warten mit einer *read*-Anweisung anstelle einer *test-and-set*-Operation durchzuführen. In Abbildung 2.13 ist diese Variante als Programm für Prozess i gezeigt. Sie hat den Vorteil, dass wartende Prozesse nur selten *test-and-set*-Operationen durchführen, da sie mittels *read*-Operationen aktiv warten. Genau wie der vorherige Algorithmus garantiert dieser Algorithmus wechselseitigen Ausschluss und ist Deadlock-frei aber nicht Starvation-frei.

Initial: Test-and-test-and-set-Bit x hat Wert 0

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  await (x=0);
(3)  while (test-and-set(x)=1) do
(4)    await (x=0);
(5)  Kritischer Abschnitt
(4)  reset(x);
end loop
```

Abbildung 2.13: Mutual-Exclusion Algorithmus mit Test-and-test-and-set-Bit

Aus praktischer Sicht ist dieser Algorithmus bei vielen Prozessen immer noch eher schlecht, da sobald ein Prozess ein *reset* durchführt, alle anderen wartenden Prozesse eine *test-and-set*-Operation durchführen. Praktisch wird gerne so genanntes „exponential backoff“ durchgeführt. Die Idee dabei ist, dass ein Prozess – nachdem sein Versuch den Lock zu setzen, scheitert – eine Zeit lang wartet. Diese Zeit kann dynamisch angepasst werden. Beim exponentiellen Verfahren wächst diese Zeit exponentiell mit jedem Fehlversuch, wobei es sinnvoll ist, obere Grenzen für die Wartezeit zu verwenden. Die meisten Programmbibliotheken stellen eine **`pause`** (oder **`delay`**) Operation zur Verfügung, die einen einzelnen Thread für eine Zahl an Millisekunden warten lässt. Damit lassen sich leicht solche Algorithmen implementieren. Abbildung 2.14 zeigt das Programm des $i.$ Prozesses eines solchen Algorithmus. Hierbei wird nur dann die Wartezeit erhöht, wenn der Lock einmal frei war und der wartende Prozess ihn nicht setzen konnte. Diese Variante scheint in der Praxis sinnvoller im Gegensatz zur Erhöhung der Wartezeit im **`await`**-Statement.

Als nächstes wird ein schneller Algorithmus für das Mutual-Exclusion Problem erörtert, der Starvation-Freiheit zusichert. Er stammt von R. Alur und G. Taubenfeld. In Abbildung 2.15 ist der Programmcode des $i.$ Prozesses angegeben.

Die Funktionsweise des Algorithmus bzw. der Ablauf des Programms für Prozess i lässt sich wie folgt erläutern: Im Initialisierungscode signalisiert der Prozess durch Setzen des Eintrags waiting[i], dass er in die Wartephase eintritt (Zeile (2)). Prozess i hat zwei Möglichkeiten aus dem Warten herauszukommen: Entweder er setzt den Lock auf das Test-and-Set-Bit lock (dann erhält die lokale Variable key den Wert 0) oder ein *anderer* Prozess setzt den Eintrag von Prozess i (also waiting[i]) auf False. Vor dem Eintritt in den kritischen Abschnitt signalisiert Prozess i,

Initial: Test-and-test-and-set-Bit x hat Wert 0, delay: lokale Variable, minDelay, maxDelay: Konstanten

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  delay := minDelay;
(3)  repeat
(4)    delay := min(2*delay,maxDelay)
(5)    while (x=1) do pause(delay);
(6)  until (test-and-set(x)=0)
(7)  Kritischer Abschnitt
(8)  reset(x);
end loop
```

Abbildung 2.14: Mutual-Exclusion Algorithmus mit „exponential backoff“

dass er den Wartebereich verlassen hat, indem er waiting[i] auf False setzt. Der etwas längere Code im Abschlusscode hat – kurz gesprochen – die Aufgabe den Lock falls möglich nicht zurückzusetzen, sondern an den nächsten wartenden Prozess zu übergeben. Hierfür kommt zunächst die gemeinsame Variable turn ins Spiel (lturn ist nur eine lokale Hilfsvariable): turn bestimmt den nächsten Prozess, der – falls er wartend ist – in den kritischen Abschnitt eintreten darf. Wenn turn den Wert i besitzt, dann wird der Wert um 1 erhöht (da Prozess i beim Verlassen des kritischen Abschnitts ist). Die modulo-Operation dient dabei dazu, dass turn von n auf 1 gesetzt wird (Zeilen (8)-(10)). In den Zeilen (11)-(15) wird zunächst geprüft, ob der durch turn ermittelte Prozess am Warten ist. Trifft dies zu, so wird sein waiting-Eintrag auf False gesetzt, d.h. der wartende Prozess darf in den kritischen Abschnitt eintreten, der Lock wird direkt (ohne reset) übergeben. Ist der durch turn ermittelte Prozess nicht am Warten, so wird der Lock zurückgesetzt, so dass alle wartenden Prozesse wieder auf ihn zugreifen können und turn wird erneut erhöht, da der zu turn zugehörige Prozess nicht wartet.

Es lässt sich leicht nachprüfen, dass gilt

Theorem 2.5.1. *Der Algorithmus aus Abbildung 2.15 garantiert wechselseitigen Ausschluss und ist Deadlock-frei.*

Der Algorithmus erfüllt jedoch stärkere Eigenschaften. Er ist Starvation-frei und erfüllt die so genannte n-Fairness, welche definiert ist als:

Definition 2.5.2. *Ein Mutual-Exclusion Algorithmus erfüllt die* r-Fairness*, genau dann, wenn stets gilt:*

Ein wartender Prozess hat die Möglichkeit den kritischen Abschnitt zu betreten bevor alle anderen Prozesse gemeinsam den kritischen Abschnitt $r+1$*-mal betreten können.*

Im Gegensatz zum r-bounded waiting zählt die r-Fairness, wie oft alle Prozesse gemeinsam (summiert) den kritischen Abschnitt betreten können, bevor ein wartender Prozesse den kritischen Abschnitt betreten darf. Allerdings sind die Begriffe eng verwandt:

Initial:
turn: atomares Register, Wert am Anfang zwischen $1 \ldots n$;
lock: Test-and-set-Bit Wert, Wert am Anfang 0;
waiting: Feld (Index: $1 \ldots n$) von atom. Registern, Wert für $i = 1, \ldots, n$: False;
lturn, key: lokale Register, Wert am Anfang egal

Programm des i. Prozesses

```
loop forever
(1)   restlicher Code
(2)   waiting[i]:=True;
(3)   key := 1;
(4)   while (waiting[i] and key=1) do
(5)      key := test-and-set(lock);
(6)   waiting[i]:=False;
(7)   Kritischer Abschnitt
(8)   if turn=i
(9)    then lturn := 1 + (turn mod n);
(10)   else lturn := turn
(11)  if waiting[lturn]
(12)   then turn := lturn;
(13)        waiting[lturn] := False;
(14)   else turn := 1+(lturn mod n);
(15)        reset(lock);
end loop
```

Abbildung 2.15: Starvation-freier Algorithmus mit Test-and-set-Bit

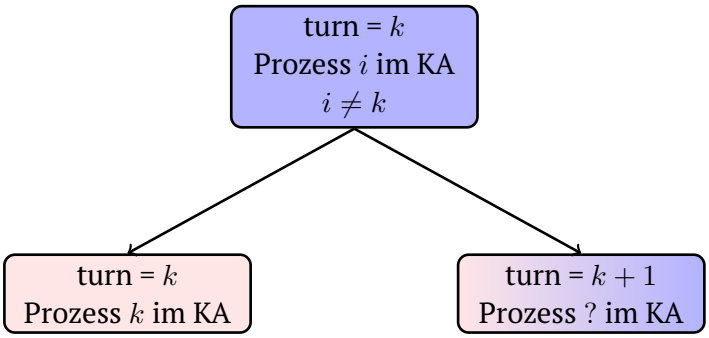
Satz 2.5.3. *Die Eigenschaft r-Fairness impliziert die Eigenschaft r-bounded waiting. Für n Prozesse gilt: Die Eigenschaft r-bounded waiting impliziert die Eigenschaft $((n-1) \cdot r)$-Fairness.*

Beweis. Die erste Implikation ist offensichtlich: Wenn in der Summe nur r Prozesse in den kritischen Abschnitt gelangen dürfen, dann kann auch jeder einzelne Prozess nur r mal in den kritischen Abschnitt. Die zweite Implikation lässt sich analog berechnen: Wenn alle anderen $(n-1)$ Prozesse höchstens r-mal jeweils den kritischen Abschnitt betreten dürfen, dann gibt es in der Summe höchstens $((n-1) \cdot r)$ Eintritte in den kritischen Abschnitt. □

Theorem 2.5.4. *Der Algorithmus aus Abbildung 2.15 ist Starvation-frei und garantiert für n Prozesse die Eigenschaft n-Fairness.*

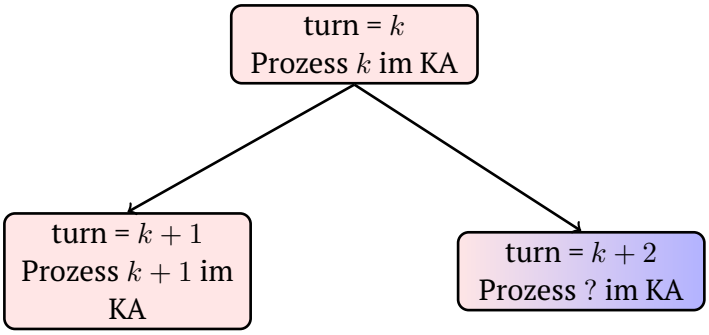
Beweis. Zunächst folgt der Beweis der n-Fairness anhand einer Betrachtung von zwei Fällen:

- Wenn Prozess i den kritischen Abschnitt verlässt, und turn $\neq i$ gilt, dann kann es für den nachfolgenden Prozess im kritischen Abschnitt zwei Möglichkeiten geben:



Entweder Prozess k betritt den kritischen Abschnitt und turn wird vorher nicht verändert, oder ein beliebiger Prozess betritt den kritischen Abschnitt, aber turn wird um 1 erhöht.

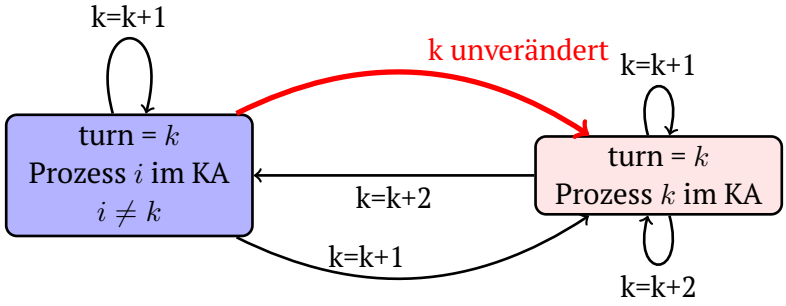
- Wenn Prozess k den kritischen Abschnitt verlässt und turn $= k$ gilt, kann es auch zwei Möglichkeiten für den nächsten Prozess im kritischen Abschnitt geben:



Entweder wird turn um 1 auf $k+1$ erhöht und Prozess $k+1$ ist anschließend im kritischen Abschnitt, oder turn wird zweimal erhöht, da Prozess $k+1$ nicht wartet. Anschließend ist ein beliebiger Prozess im kritischen Abschnitt.

Die Farbgebung der Knoten ist nicht wahllos getroffen: Zustände, in denen der Wert von turn gleich dem Prozess im kritischen Abschnitt ist, sind rot, andere sind blau. Bei den gemischten Zuständen können beide Fälle zutreffen.

Vereint man die Fälle, so kann man folgendes Schaubild erstellen:



Sei nun Prozess l ein wartender Prozess, d.h. es gilt waiting[l]=True. Dann zeigt das Schaubild, dass nach n-maligem Verlassen des kritischen Abschnitts irgendwann $turn = l$ gegolten haben muss. Hierbei ist wichtig, dass immer wenn der rote Pfeil gewählt wurde, er danach erst wieder gewählt werden kann, wenn turn um 2 erhöht wurde. Das ist wichtig, da über den roten Pfeil der kritische Abschnitt verlassen werden kann *ohne* turn zu erhöhen. Da $turn = l$ irgendwann galt und waiting[l]=True gilt, muss Prozess l spätestens nach n-maligem Ausführen des Ab-

schlusscodes in den kritischen Abschnitt gelangen. Beachte: n-maliges Verlassen (statt $n-1$) kann nötig sein, wenn am Anfang der rote Pfeil gewählt wird.

Starvation-Freiheit folgt aus der Deadlock-Freiheit und der n-Fairness. □

2.5.2 Ein Mutual-Exclusion Algorithmus mit RMW-Objekt

Im folgenden wird ein weiterer Algorithmus für das Mutual-Exclusion-Problem vorgestellt – der *Ticket-Algorithmus*. Der Algorithmus stammt von M.J. Fischer, N.A. Lynch, J.E. Burns, A. Borodin aus dem Jahr 1989 und benutzt ein Read-Modify-Write-Objekt. Die Idee des Algorithmus ist ähnlich wie beim Bakery-Algorithmus: Im Doorway zieht ein Prozess ein Ticket (eine Zahl) und erhöht die Zahl für den nächsten Prozess. Mithilfe des Read-Modify-Write Objektes kann dieser Schritt atomar erfolgen. Im Gegensatz zum Bakery-Algorithmus sind die Zahlen für die Tickets jedoch nicht beliebig groß, sondern werden nur modulo n erhöht.

Das Programm für den i. Prozess des Ticket-Algorithmus zeigt Abbildung 2.16.

Initial:
(ticket,valid): Read-Modify-Write Objekt für ein Paar von Zahlen im Bereich $1 \ldots n$ wobei initial ticket = valid
(ticket$_i$, valid$_i$) für $i = 1, \ldots, n$: lokales Register

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
     // erhöhe ticket (lese alte Werte):
(2)  (ticket_i,valid_i) := read-modify-write((ticket,valid), inc-fst)
(3)  while ticket_i ≠ valid_i do
(4)      valid_i := valid;
(5)  Kritischer Abschnitt
(6)  read-modify-write((ticket,valid), inc-snd) // erhöhe valid
end loop
```

Funktionen

```
function inc-fst((a,b))
  return (1+(a mod n), b)
end function

function inc-snd((a,b))
  return (a, 1+(b mod n))
end function
```

Abbildung 2.16: Ticket-Algorithmus

Beachte, dass am Anfang ticket=valid gelten muss, damit der allererste Prozess den kritischen Abschnitt betreten kann. Mutual-Exclusion gilt, da valid immer erst beim Ausführen des Abschlusscodes erhöht wird. Deadlock-Freiheit ist ebenfalls leicht einzusehen. Zusätzlich erfüllt der Algorithmus die FIFO-Eigenschaft, d.h. die Reihenfolge des Eintretens in den kritischen Abschnitt entspricht der Reihenfolge des Ticket-Ziehens.

Theorem 2.5.5. *Der Ticket-Algorithmus garantiert wechselseitigen Ausschluss, ist Starvation-frei und hat die FIFO-Eigenschaft.*

Bezüglich des Wertebereichs des RMW-Objektes benutzt der Ticket-Algorithmus n^2 Werte (jeweils n für ticket und n für valid). Dies ist der beste bekannte Algorithmus bezüglich dieser Größe, wenn man die (starke) FIFO-Eigenschaft fordert.

2.5.3 Der MCS-Algorithmus mit Queue

Als letzten Algorithmus in diesem Kapitel wird der MCS-Algorithmus vorgestellt. Er wurde von J.M. Mellor-Crummey und M.L. Scott im Jahr 1991 veröffentlicht. Die Idee dabei ist, dass die wartenden Prozesse sich in eine Warteschlange einreihen. Die Konstruktion und Dekonstruktion der Warteschlange wird dabei mithilfe eines Compare-and-Swap Objektes bewerkstelligt, welches zusätzlich die Swap-Operation unterstützen muss. Der Algorithmus besitzt als weitere Besonderheit, dass der Abschlusscode unseren Annahmen nicht ganz entspricht, denn er ist nicht „wait-free“.

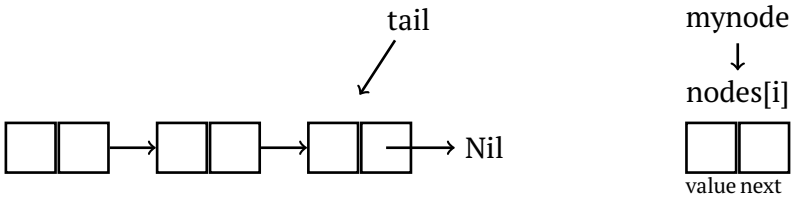
Der Algorithmus verwendet *Zeiger* um eine verzeigerte Liste zu erstellen. Hierbei wird folgende Notation für einen Zeiger p verwendet:

- ***p** als Indirektions-Operation, d.h. ***p** liefert das Objekt, auf welches der Zeiger zeigt.
- **&**p gibt die Speicheradresse eines Objekts und kann verwendet werden, um damit Zeiger auf das Objekt zu erstellen.

Der Algorithmus baut verzeigerte Listen auf, wobei ein Listenelement zwei Komponenten besitzt (und mithilfe eines Records dargestellt werden kann): Den Wert des Elements und einen Zeiger auf das nächste Element. Sei e ein Listenelement, dann sei e.value der Wert und e.next der Zeiger auf das nächste Element. Zeiger im MCS-Algorithmus zeigen entweder auf Listenelemente oder auf Nil (leere Liste, d.h für das Listende zeigt ein Zeiger auf Nil).

In Abbildung 2.17 ist das Programm des $i.$ Prozesses und eine Übersicht über die Variablen dargestellt. Die einzelnen Phasen des Algorithmus werden detailliert erläutert. Der Zeiger tail zeigt immer auf das letzte Listenelement. Wenn die Liste leer ist (z.B. am Anfang), so zeigt tail auf Nil. Die Feldeinträge nodes[i] sind Listenelemente jeweils für den $i.$ Prozess. Will ein Prozess in den kritischen Abschnitt, so hängt er sein Listenelement in die Liste ein.

Im Initialisierungscode wird zunächst (Zeile (2)) der next-Zeiger des nodes[i]-Eintrages auf Nil gesetzt. Dieses Nil wird später das neue Listenende markieren. Im nächsten Schritt wird der lokale Zeiger prev erstellt, der genau wie mynode auf nodes[i] zeigt. In Zeile (4) wird nun der tail-Zeiger mit dem prev Zeiger ausgetauscht. Die swap-Operation sichert hierbei zu, dass dies atomar in einem Schritt geschieht. Ab diesem Schritt zeigt tail auf nodes[i] – das neue letzte Listenelement, wobei das eigentliche Anhängen an die Liste noch fehlt. Diese Schritte lassen sich mit den folgenden Box-and-Pointer-Diagrammen illustrieren:



Typen:

- Record Element: zwei Attribute: value:Bool, und next: Zeiger auf ein Element

Gemeinsame Variablen:

- nodes[i]: Feldeintrag: Inhalt ein Record vom Typ Element
- tail: Swap und Compare-and-Swap Objekt vom Typ: Zeiger auf ein Element, Wert am Anfang: Nil

Lokale Variablen:

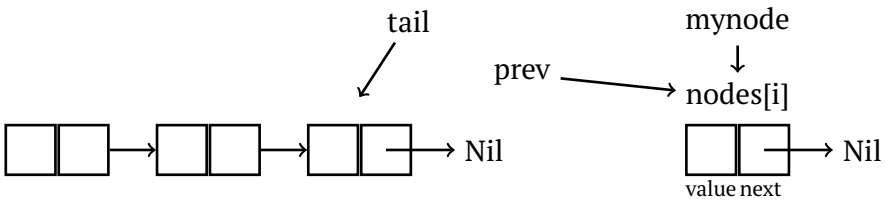
- mynode: Zeiger auf ein Element, am Anfang auf nodes[i]
- prev, succ: Zeiger auf Elemente

Programm des i. Prozesses

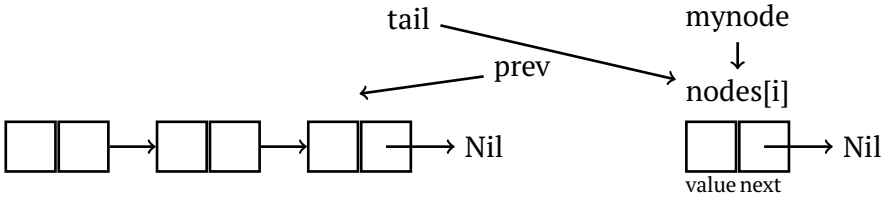
```
loop forever
(1)   restlicher Code
(2)   *mynode.next := Nil;
(3)   prev := mynode:
(4)   swap(tail,prev);
(5)   if prev ≠ Nil then
(6)     *mynode.value := 1;
(7)     *prev.next := mynode;
(8)     await *mynode.value = 0;
(9)   Kritischer Abschnitt
(10)  if mynode.next = Nil then
(11)    if compare-and-swap(tail,mynode,Nil) = False then
(12)      await *mynode.next ≠ Nil;
(13)      succ := *mynode.next;
(14)      *succ.value := 0;
(15)  else
(16)    succ := *mynode.next;
(17)    *succ.value := 0;
end loop
```

Abbildung 2.17: Der MCS-Algorithmus

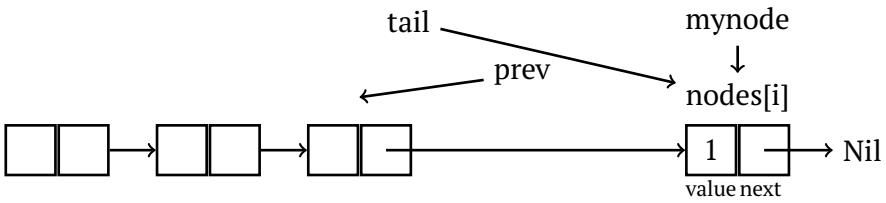
Nach Ausführen von Zeilen (2) und (3):



Nach Ausführen von Zeile (4)

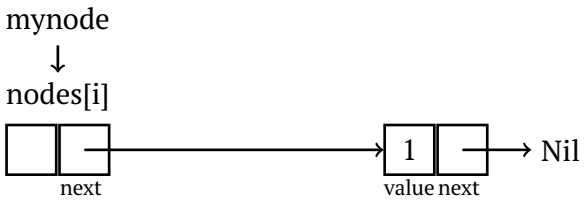


In Zeile (5) wird geprüft, ob es Elemente in der Liste gibt (dann muss gewartet werden). Wenn der Zeiger prev auf Nil zeigt, dann ist klar, dass tail vorher auf Nil zeigte, die Liste also leer war. In diesem Fall kann der kritische Abschnitt betreten werden. Ist dies nicht der Fall, so werden Zeile (6)-(8) ausgeführt: der Wert-Eintrag von nodes[i] wird auf 1 gesetzt, das Element nodes[i] an die Liste angehängt (durch Setzen des next-Zeigers des vorherigen Elements), und anschließend wird gewartet, bis der Wert-Eintrag von nodes[i] auf 0 gesetzt wird. In der Illustration resultiert dies in der Situation:

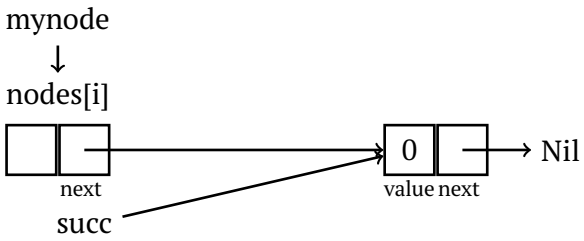


Im Abschlusscode wird zunächst überprüft, ob der next-Eintrag auf Nil oder ein Element zeigt. Wenn der Zeiger auf ein Element zeigt, dann hat sich in der Zwischenzeit ein weiterer Prozess eingehängt. Dessen value-Wert wird dann in Zeilen (16) und (17) auf 0 gesetzt, damit er das Warten verlassen kann.

Das lässt sich wie folgt illustrieren: Beim Abfragen in Zeile (10)

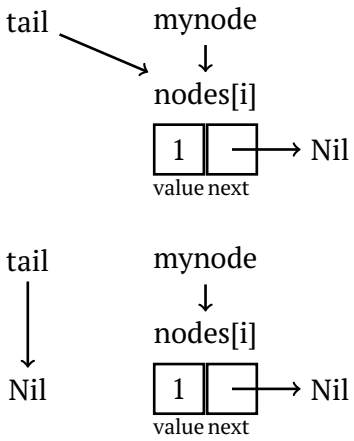


Da ein Nachfolger existiert, wird dessen Wert gesetzt:

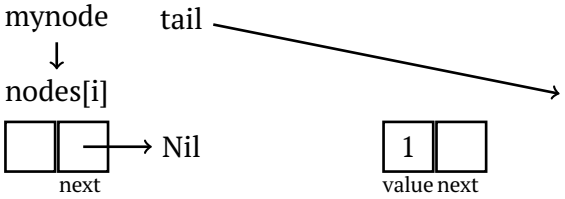


Beachte, dass der Prozess im Abschlusscode sich nicht explizit von der Liste abhängen muss, da er an erster Stelle der Liste sein muss (d.h. die alten Zeigereinträge sind irrelevant).

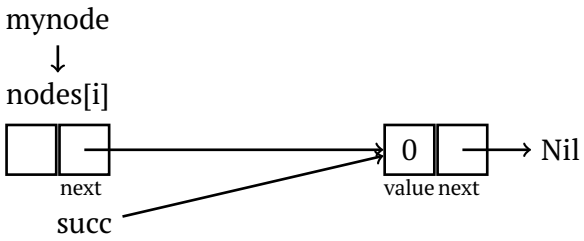
Ist die **`if`**-Bedingung in Zeile (10) erfüllt, so können zwei Fälle zutreffen: Entweder ist der Prozess im Abschlusscode wirklich das letzte Listenelement und es befindet sich kein weiterer Prozess im Initialisierungscode, oder weitere Prozesse sind im Initialisierungscode, aber sie haben den next-Zeiger des Prozesses im Abschlusscode noch nicht verändert (dies geschieht in Zeile (7)). Die Unterscheidung in diese beiden Fälle geschieht durch eine *compare-and-swap*-Operation: Zeigen beide Zeiger tail und mynode auf das gleiche Element (nämlich nodes[i], wenn Prozess i im Abschlusscode ist), dann hat kein anderer Prozess die swap-Operation in Zeile (4) ausgeführt. In diesem Fall wird durch die *compare-and-swap*-Operation der tail-Zeiger auf Nil gesetzt, und der Prozess im Abschlusscode hängt sich dadurch ab; die Liste ist leer. Dieser Fall lässt sich durch die beiden Box-and-Pointer-Diagramme illustrieren:



Im anderen Fall schlägt die *compare-and-swap*-Operation in Zeile (11) fehl, d.h. der Zeiger tail wird nicht verändert. Nun wartet der Prozess im Abschlusscode bis sein eigener next-Zeiger durch den Prozess im Initialisierungscode verändert wird: Einen solchen Prozess muss es geben, da der tail-Zeiger verändert wurde und nicht mehr gleich zu mynode ist. Ist dies geschehen, so wird genau wie vorher fortgefahren: Der value-Eintrag des Nachfolgers wird von 1 auf 0 gesetzt. Die Wartesituation lässt sich durch folgendes Bild illustrieren: Zunächst, zeigt tail nicht mehr auf das aktuelle Element nodes[i], aber der next-Zeiger zeigt noch auf Nil.



Nun wird gewartet, bis der next-Zeiger umgehängt wird, anschließend wird der value-Eintrag des Nachfolgers auf 0 gesetzt:



Es lässt sich zeigen, dass gilt:

Theorem 2.5.6. *Der MCS-Algorithmus garantiert wechselseitigen Ausschluss, ist Starvation-frei und erfüllt die FIFO-Eigenschaft.*

Der MCS-Algorithmus besitzt als weitere Eigenschaft, dass die einzelnen Prozesse immer auf unabhängigen Speicherplätzen ihre Warte-Operationen ausführen, d.h. es werden keine Speichereinträge von mehreren Prozessen wiederholend abgefragt. Dies wirkt sich in Implementierungen positiv aus, da Wertänderungen dieser Speicherplätze nur selten allen Prozessen mitgeteilt werden müssen. Dies ist vor allem in solchen Situationen nützlich, in denen viele Prozesse erzeugt werden. Hier zeigt sich, dass sich der MCS-Algorithmus sehr gut verhält, da nicht auf der Sperrvariablen (hier tail) gewartet wird.

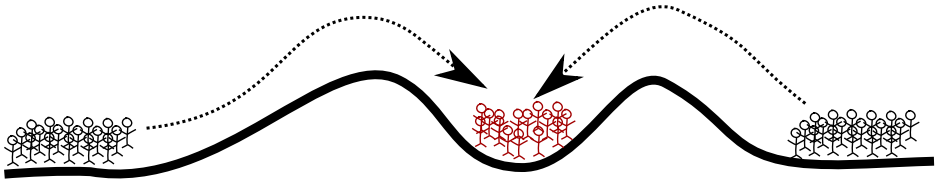
2.6 Konsensus und die Herlihy-Hierarchie

In den letzten Abschnitten wurden verschiedene Speicheroperationen und entsprechende nebenläufige Objekte eingeführt und benutzt. Es stellt sich die Frage, ob man die verschiedenen Objekte bezüglich ihrer *Ausdruckskraft* vergleichen kann. Intuitiv sind Objekte, die ausschließlich *read* und *write* als atomare Operationen bereitstellen, am schwächsten (z.B. ist das Mutual-Exclusion-Problem eher schwer damit lösbar), und RMW-Objekte sind stark, da man damit viele andere Objekte ausdrücken kann. Maurice Herlihy (Herlihy, 1991) hat zum Vergleich der Ausdruckskraft einen allgemeinen Ansatz entwickelt und anschließend eine Hierarchie der Objekte aufgestellt. In diesem Abschnitt werden Herlihys Resultate kurz dargestellt.

2.6.1 Prozessmodell mit Abstürzen

Zunächst verwendet Herlihys Ansatz ein etwas anderes Modell, welches für diesen Abschnitt übernommen wird: Prozesse können jederzeit *abstürzen*: Jeder Prozess kann an jedem Befehl für immer hängen bleiben, aber einzelne Befehle werden atomar also ganz oder gar nicht ausgeführt. Beachte, dass für dieses Modell keiner der vorgestellten Mutual-Exclusion-Algorithmen funktioniert, da Prozesse z.B. auch im kritischen Abschnitt oder im Abschlusscode abstürzen und damit einen Deadlock herbeiführen können.

Ein schönes Beispiel in diesem Modell ist das sogenannte „coordinated attack“-Problem oder auch „Zwei Generäle-Problem“: Die Situation des Problems ist die folgende: Zwei Divisionen möchten eine feindliche Division angreifen. Die feindliche Division befindet sich im Tal zwischen zwei Hügeln. Die angreifenden Division befinden sich links und rechts hinter den Hügeln.



Die angreifenden Divisionen werden sicher verlieren, wenn sie nicht im gleichen Moment angreifen. Daher müssen sie sich absprechen, um im selben Moment angreifen zu können. Zur Kommunikation stehen den Generälen nur Boten zur Verfügung, die sie zur gegenüberliegenden Division schicken können. Der jeweilige Bote muss dabei das feindliche Tal durchlaufen und kann daher unter Umständen vom Feind gefasst werden, d.h. es ist nicht sichergestellt, ob der Bote beim anderen General tatsächlich ankommt. Der rechte General schickt einen Boten los, um dem linken General zu informieren, dass der Angriff um 12 Uhr mittags erfolgen soll. Da nicht sicher ist, ob der Bote angekommen ist, wird der rechte General darauf warten, dass der linke General einen Boten zurückschickt, der ihm bestätigt, dass die Botschaft angekommen ist. Der linke General kann sich aber nicht sicher sein, dass sein Bote wirklich angekommen ist, und kann daher nicht sicher sein, dass der Angriff wirklich stattfindet. Als möglichen Ausweg schickt der rechte General anschließend erneut einen Boten, um den Empfang zu bestätigen. Aber auch er kann nun nicht sicher sein, dass die Botschaft wirklich ankommt. Dieses Hin- und Herschicken findet daher kein Ende und es wird nicht zum Angriff kommen. Tatsächlich kann man relativ leicht nachweisen, dass das Problem des koordinierten Angriffs in diesem Modell nicht lösbar ist. Man beachte, dass die Schwierigkeit / Unlösbarkeit nur dadurch entsteht das Boten ausfallen können. Übertragen auf das Prozessmodell wäre ein solcher Bote gerade ein Prozess, der u.U. abstürzen kann.

2.6.2 Das Konsensus-Problem

Herlihy hat zur Untersuchung der Ausdruckskraft der nebenläufigen Objekte das sogenannte *Konsensus-Problem* betrachtet. Gegeben seien n Prozesse. Jeder Prozess i erhält einen Eingabewert $x_i \in \{0, 1\}$. Das Konsensus-Problem besteht darin, die Prozesse so zu programmieren, dass alle Prozesse sich für einen gemeinsamen Wert $d \in \{0, 1\}$ entscheiden (dieser Wert wird *Entscheidungswert* genannt). Für den Entscheidungswert muss dabei gelten:

- *Übereinstimmung*: Alle nicht-abgestürzten Prozesse entscheiden sich für den gleichen Wert d.
- *Gültigkeit*: $d \in \{x_1, \ldots, x_n\}$, d.h. d entspricht einem der Eingabewerte.

Gäbe es keine abstürzenden Prozesse, so wäre es einfach das Konsensus-Problem zu lösen: Alle Prozesse teilen sich (über gemeinsamen Speicher) untereinander alle ihre Eingabewerte mit, anschließend berechnet jeder Prozess $f(x_1, \ldots, x_n) = d$ wobei f eine feste Funktion ist, die alle Prozesse kennen, z.B. $f(x_1, \ldots, x_n) = x_1$. Da Prozesse jedoch abstürzenden dürfen, ist das Problem nicht mehr einfach zu lösen. Beachte, dass die Übereinstimmungsbedingung auch impliziert, dass der zu entwerfende Algorithmus *wait-free* sein muss, d.h. unabhängig davon ob die anderen Prozesse abstürzen oder nicht abstürzen muss gelten: Wenn Prozess i nicht abstürzt, so entscheidet er sich nach endlich vielen Schritten für den Entscheidungswert d, oder umgekehrt: Prozess i gerät nicht in Gefahr, in einer Endlosschleife hängen zu bleiben.

Mithilfe eines dreiwertigen RMW-Objekts ist es sehr einfach das Konsensus-Problem zu lösen. In Abbildung 2.18 ist der entsprechende Code zu finden. Das RMW-Objekt kann die Werte $\perp$, 0

oder 1 annehmen und hat initial den Wert $\perp$. Jeder Prozess versucht seinen Eingabewert in das RMW-Objekt zu schreiben, allerdings nur, wenn dieses den Wert $\perp$ besitzt, anderenfalls bleibt der Wert unverändert. Dies führt dazu, dass der erste erfolgreiche Prozess den Wert setzt, und alle anderen Prozesse den Wert übernehmen. Insbesondere heisst das auch, dass für alle nicht abstürzenden Prozesse der Wert d_i identisch ist. Abstürzende Prozesse sind dabei kein Problem, da sie den Wert und den Zugriff auf das RMW-Objekt nicht beeinflussen. D.h. der Algorithmus aus Abbildung 2.18 stellt eine wait-freie Lösung für das Konsensus-Problem für beliebig viele Prozesse dar.

Objekte und Initialisierung:
x: Read-Modify-Write Objekt mit den möglichen Werten $\perp, 0, 1$, initial $\perp$
x_i: Eingabewert von Prozess i
d_i: Entscheidungswert, den Prozess i trifft.

Programm des i. Prozesses

```
(1)  d_i := read-modify-write(x,f_i);
(2)  if d_i = ⊥ then
       d_i := x_i;
```

Funktion f_i des i. Prozesses

```
function f_i(v)
  if v=⊥ then return x_i
  else return v
end function
```

Abbildung 2.18: Konsensus mit einem dreiwertigen RMW-Objekt

Ein wichtiges prominentes Resultat (auch bekannt als „FLP impossibility result“, benannt nach den Autoren Fischer, Lynch und Paterson, (Fischer et al., 1985)) ist jedoch, dass mit atomarem *read* und *write* das Konsensus-Problem – selbst für 2 Prozesse und nur einen Absturz – nicht lösbar ist:

Theorem 2.6.1. *Es gibt keinen Konsensus-Algorithmus für atomares read und write, der einen Absturz tolerieren kann.*

Beweis. Wir skizzieren den Beweis, wobei wir im Wesentlichen die Darstellung und Argumente aus (Herlihy & Shavit, 2008) verwenden. Wir nehmen dabei nicht an, dass Prozesse abstürzen, aber dass es einen wait-freien Konsensusalgorithmus für 2 Prozesse gibt, der nur atomare Register verwendet und widerlegen seine Existenz.

Ähnlich zum Beweis von Theorem 2.4.3 verwenden wir Berechnungsbäume für die Ausführung von zwei Prozessen, die eine Konsensusberechnung durchführen, wobei die Knoten Zustände der Prozesse und des Speichers repräsentieren und die Kanten *wesentliche Berechnungsschritte* der Prozesse P_1, P_2 repräsentieren. Ein *wesentlicher Schritt* ist eine Operation eines gemeinsamen Speicherobjekts. Da wir hier nur mit atomaren Registern argumentieren, sind das Lese- und Schreiboperationen. Eine Kante zum linken Kind repräsentiert einen wesentlichen Schritt von Prozess P_1 und eine Kante zum rechten Kind repräsentiert einen wesentlichen Schritt vom Prozess P_2. Ein Blatt ist ein Zustand, indem beide Prozesse sich für einen (gemeinsamen)

Entscheidungswert entschieden haben. Ein Anfangszustand ist ein Zustand, bevor irgendein Prozess einen wesentlichen Schritt gemacht hat. Da der Konsensusalgorithmus wait-free sein muss, ist der Berechnungsbaum tatsächlich ein endlicher Baum. Da Blätter einen eindeutigen Entscheidungswert haben, markieren wir diese mit diesem Wert, der entweder 0 oder 1 ist. Ein Knoten ist *bivalent*, wenn der Entscheidungswert noch nicht feststeht (d.h. von ihm aus sind Blätter mit Markierungen 0 als auch 1 erreichbar). Ansonsten nennen wir einen Knoten *univalent* bzw. *1-valent*, wenn der Entscheidungswert 1 feststeht und *0-valent*, wenn 0 als Entscheidungswert feststeht.

Behauptung 1: *Jeder 2-Prozess Konsensusalgorithmus hat einen bivalenten Anfangszustand.*
Beweis von Behauptung 1: Nehme an P_1 hat Eingabewert $x_1 = 0$ und P_2 hat Eingabewert $x_2 = 1$. Wenn nur P_1 Schritte macht (linkester Pfad im Baum), dann muss P_1 als Entscheidungswwert 0 berechnen, umgekehrt: Wenn P_1 nie einen Schritt macht, dann muss P_2 als Entscheidungswert 1 berechnen.

Ein Knoten ist *kritisch*, wenn er bivalent ist und jeder Nachfolger ist univalent.

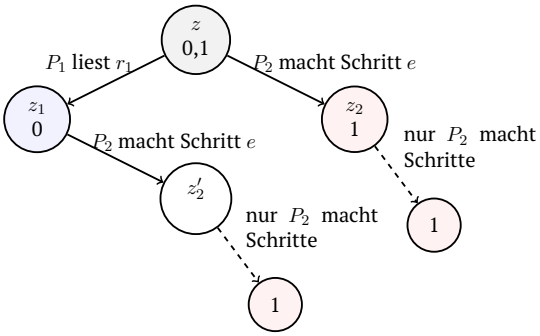
Behauptung 2: *Jeder Berechnungsbaum zu einem Konsensusalgorithmus, der mit einem bivalenten Anfangszustand startet, hat einen kritischen Knoten.*
Beweis von Behauptung 2: Angenommen das gilt nicht. Starte mit einem bivalenten Anfangszustand. Solange es Schritte gibt, die zu einem bivalenten Zustand führen, führe diese Schritte aus. Wenn das unendlich lange möglich ist, dann ist der Algorithmus nicht wait-free, was einen Widerspruch darstellt.

Nehme nun an, es gibt einen Konsensusalgorithmus für 2 Prozesse. Wir führen den Algorithmus bis zu einem kritischen bivalenten Zustand z aus. Die Kinder dieses Zustands sind univalent und zwar einmal 0-valent und einmal 1-valent. O.b.d.A. sei das linke Kind z_1 0-valent und das rechte Kind z_2 1-valent. Wir betrachten die Schritte von P_1 und P_2 zu deren Kindern (siehe Abbildung 2.19)

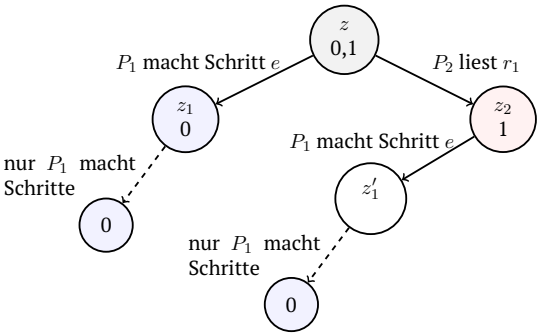
- Wenn P_1 eine Leseoperation durchführt und P_2 eine Lese- oder Schreiboperation e, dann kann P_2 die Operation e auch von z_1 aus ausführen. Der erhaltene Zustand ist aus Sicht von P_2 identisch zu z_2. Daher kann P_2 den kompletten rechtesten Pfad bis zum Blatt von dort ausführen. Dieses Blatt muss 1-valent sein. Dann kann z_1 aber nicht 0-valent sein. Widerspruch.
- Wenn P_2 eine Leseoperation durchführt und P_1 eine Lese- oder Schreiboperation e, dann kann P_1 die Operation e auch von z_2 aus ausführen. Der erhaltene Zustand ist aus Sicht von P_1 identisch zu z_1. Daher kann P_1 den kompletten linkesten Pfad bis zum Blatt von dort ausführen. Dieses Blatt muss 0-valent sein. Dann kann z_2 aber nicht 1-valent sein. Widerspruch.
- Wenn P_1 und P_2 jeweils Schreiboperationen in verschiedene Register ausführen, dann kann P_2 von z_1 und P_1 von z_2 die Schreiboperation ebenso durchführen und der erreichte Zustand z' ist jeweils der selbe. Dieser kann dann aber nicht univalent sein! Widerspruch.
- Wenn P_1 eine Schreiboperation und P_2 eine Schreiboperation auf das gleiche Register durchführen, dann kann P_2 die Schreiboperation auch von z_1 aus durchführen und danach dieselben Schritte wie auf dem rechtesten Pfad. Dieser Pfad muss in einem Blatt enden, welches 1-valent ist. Dann kann z_1 nicht 0-valent sein. Widerspruch.

□

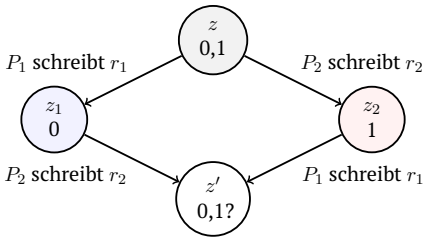
Ein ausführlicherer Beweis ist in der Literatur zu finden. Als erstes Fazit lässt sich feststellen,



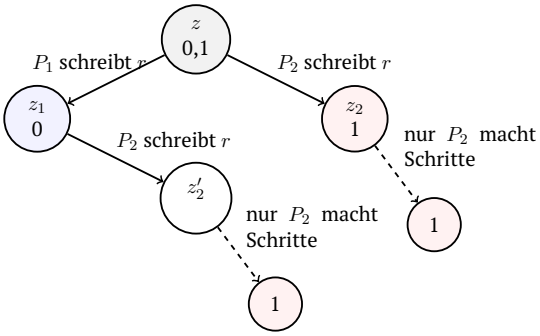
Fall 1: P_1 führt Leseoperation aus



Fall 2: P_3 führt Leseoperation aus



Fall 3: P_1 und P_2 schreiben in unterschiedliche Register



Fall 4: P_1 und P_2 schreiben in dasselbe Register

Abbildung 2.19: Fälle in der Beweisskizze von Theorem 2.6.1

dass das Konsensus-Problem für atomares *read* und *write* nicht lösbar ist, aber für ein drei-wertiges RMW-Objekt für jede Anzahl von Prozessen und Abstürzen einfach lösbar ist. Eine Konsequenz ist, dass ein RMW-Objekt offensichtlich nicht durch atomare *read* und *write* Operationen implementiert (bzw. simuliert) werden kann.

2.6.3 Die Konsensus-Zahl

Basierend auf dem Konsensus-Problem lässt sich die folgende *Konsensus-Zahl* für ein nebenläufiges Objekt definieren:

Definition 2.6.2 (Konsensus-Zahl)**.** *Für ein nebenläufiges Objekt vom Typ o ist die* Konsensus-Zahl $\mathbb{CN}(o)$ *die größte Zahl an Prozessen n für die man das Konsensus-Problem für n Prozesse lösen kann, indem man beliebig viele Objekte vom Typ o und beliebig viele atomare Register (mit read und write) verwendet. Ist die Anzahl unbeschränkt, so sei $\mathbb{CN}(o) = \infty$.*

Die folgende Tabelle zeigt einige Konsensus-Zahlen

$\mathbb{CN}(o)$	Objekt o
1	atomares Register mit *read* und *write*
2	test-and-set-Objekt, fetch-and-increment-Objekt, fetch-and-add-Objekt, swap-Objekt, read-modify-write-Bit
$\Theta(\sqrt{m})$	swap^m-Objekt
$2m-2$	m-Register mit m-facher Zuweisung($m > 1$)
∞	(drei-wertiges) RMW-Objekt, Compare-and-swap-Objekt, Sticky-Bit

Erwähnenswert ist zum einen, dass der Algorithmus aus Abbildung 2.18 leicht auf ein Compare-and-Swap-Objekt und auch auf ein Sticky-Bit angepasst werden kann und daher diese Objekte genau wie das drei-wertige RMW-Objekt die Konsensus-Zahl ∞ besitzt. Die Tabelle enthält auch einige Multi-Objekte: Ein swap^m-Objekt erlaubt es m-Speicherplätze in einem atomaren Schritt zu vertauschen, ein m-Register mit m-facher Zuweisung erlaubt die atomare (Mehrfach-) Zuweisung von m gemeinsamen Registern.

Ein wichtiges Resultat bezüglich dieser Hierarchie ist:

Theorem 2.6.3. *Wenn o_1 und o_2 zwei Objekte sind mit $\mathbb{CN}(o_1) < \mathbb{CN}(o_2)$, dann gilt für ein System mit $\mathbb{CN}(o_2)$ Prozessen:*

- *Es gibt keine wait-freie Implementierung von Objekt o_2 ausschließlich mit Objekten vom Typ o_1 und atomaren Registern.*
- *Es gibt eine wait-freie Implementierung von Objekt o_1 ausschließlich mit Objekten vom Typ o_2 und atomaren Registern.*

Skizze. Die erste Aussage folgt leicht durch einen Widerspruch: Nehme an ein Objekt vom Typ o_2 kann mittels Objekten vom Typ o_1 und atomaren Registern wait-frei implementiert werden. Sei $n = \mathbb{CN}(o_2)$. Dann gibt es einen Konsensusalgorithmus für n Prozesse mit Objekten vom Typ o_2 und atomaren Registern. Ersetze dort alle o_2-Objekte durch die Implementierung mit o_1-Objekten. Dies ergibt einen korrekten Konsensusalgorithmus für n Prozesse mit Objekten vom Typ o_1 und atomaren Registern. Daher gilt $\mathbb{CN}(o_1) = n$, was ein Widerspruch zur Annahme $\mathbb{CN}(o_1) < \mathbb{CN}(o_2)$ ist.

Die zweite Aussage ist nicht so einfach zu beweisen. Sie folgt aus dem nächsten Theorem 2.6.4, da dieses impliziert, dass man für $n = \mathbb{CN}(o_2)$ Prozesse jedes nebenläufige Objekt mit einer sequentiellen Beschreibung nur mit o_2-Objekten und atomaren Registern wait-frei implementieren kann. Damit kann man insbesondere auch o_1-Objekte für $n = \mathbb{CN}(o_2)$ Prozesse damit kodieren. ¸ □

Dies zeigt, dass Objekte mit höheren Konsensus-Zahlen ausdrucksstärker sind. Ein weiteres Resultat ist, dass Objekte mit einer Konsensus-Zahl von mindestens n sogenannte *universelle Objekte* sind:

Theorem 2.6.4. *Ein Objekt o mit $\mathbb{CN}(o) \geq n$ ist universell in einem System mit n Prozessen. D.h. jedes wait-freie Objekt (mit einer sequentiellen Beschreibung, wie die Objektbeschreibungen am Anfang von Abschnitt 2.5) kann durch Objekte vom Typ o und atomaren Registern in einem System mit n Prozessen implementiert werden.*

Wir verzichten auf den genauen Beweis des Theorems (dieser kann im Original in (Herlihy, 1991) oder z.B. in den Lehrbüchern (Taubenfeld, 2006) oder (Herlihy & Shavit, 2008) nachgelesen werden). Die wesentliche Idee des Beweises ist, dass heder Prozess die Zugriffe auf die nebenläufige Objekte in einer großen Tabelle speichert und jeder Prozess rechnet die Nachfolgezustände der Objekte solange aus bis sein Zugriff stattfindet. Dieses Ausrechnen selbst ist aufgrund der sequentiellen Beschreibung kein echtes Problem. Die beiden Hürden im Beweis sind, dass einerseits eine Reihenfolge festgelegt werden muss, wann welche Operationen welches Prozesses ausgeführt wird und andererseits muss dabei sichergestellt werden, dass diese Ausführung nicht beliebig lange verzögert werden kann (um eine wait-freie Implementierung zu erhalten). Für die erste Hürde wird der Konsensusalgorithmus eingesetzt, mit ihm wird bestimmt, welcher Prozess die nächste Operation ausführen darf. Die zweite Hürde wird durch geschicktes Abarbeiten aller Operationen durch alle Prozesse bewerkstelligt.

Die Universalität eines Objekts besagt, dass sequentiell beschreibbare Objekte (die wait-Freiheit garantieren), durch das Objekt kodierbar sind. D.h. die spezielle Eigenschaft der Konsensus-Zahl macht auch Aussagen über das Objekt im Generellen. Der Schluss aus dem Theorem ist es, dass Objekte mit einer Konsensus-Zahl von n (oder größer) zu bevorzugen sind, und dass eine Art solcher Objekte ausreichend ist.

2.7 Quellennachweis

Die Darstellung der Annahmen und Algorithmen folgt meistens (Taubenfeld, 2006). Teile (z.B. die Variablenbenennung mancher Algorithmen) folgen (Ben-Ari, 2006). Die Originalveröffentlichungen der vorgestellten Algorithmen, Korrektheitsbeweise und Komplexitätsresultate sind allesamt in (Taubenfeld, 2006) zu finden. Eine umfassende Darstellung des Konsensusproblems und Konsensusalgorithmen (aus der auch die Beweisskizze zu Theorem 2.6.1 stammt) ist in (Herlihy & Shavit, 2008).

3
Programmierprimitiven

Nachdem im vorherigen Kapitel primitive Operationen betrachtet und untersucht wurden, die von der Hardware bereit gestellt werden, werden in diesem Kapitel Programmierabstraktionen erörtert, also Primitiven, die von Programmierbibliotheken nebenläufiger Programmiersprachen bereit gestellt werden. Die Programmierkonstrukte ermöglichen im Allgemeinen eine komfortablere Programmierung. Gleichzeitig zur Darstellung dieser Konstrukte (und entsprechender Modellanpassungen) werden einige (klassische) Beispiele / Problemstellungen der nebenläufigen Programmierung erläutert und Lösungen mithilfe der Programmierabstraktionen entwickelt.

3.1 Erweiterungen des Prozessmodells

Zur Beschreibung der Eigenschaften der verschiedenen Programmierabstraktionen wird zunächst das Modell ein wenig geändert, indem Prozessen verschiedene Zustände (oder Phasen) zugeordnet werden. Die Kontrolle, in welcher Phase sich ein Prozess befindet, bzw. wie der Wechsel eines Zustands in einen anderen erfolgt, liegt im Allgemeinen beim Betriebssystem (bei Betriebssystemprozessen) oder bei der Laufzeitumgebung der Programmiersprachen.

Wenn genügend Prozessoren für alle laufenden Prozesse zur Verfügung stehen, so sind im Normalfall alle Prozesse im Zustand *laufend* (engl. *running*). Wenn zu wenig Ressourcen (z.B. Prozessoren) zur Verfügung stehen, können nicht alle Prozesse laufend sein. Ein Prozess der laufen will, aber aufgrund des Ressourcenmangels nicht laufen darf, wird als *bereit* (eng. *ready*) bezeichnet. Während des Ablaufs eines nebenläufigen Programms sollte stets mindestens ein Prozess laufend und beliebig viele weitere Prozesse bereit sein. Auch wenn kein eigentlicher Prozess vorhanden ist, der Arbeit verrichten möchte, so läuft normalerweise ein *Leerlaufprozess*.

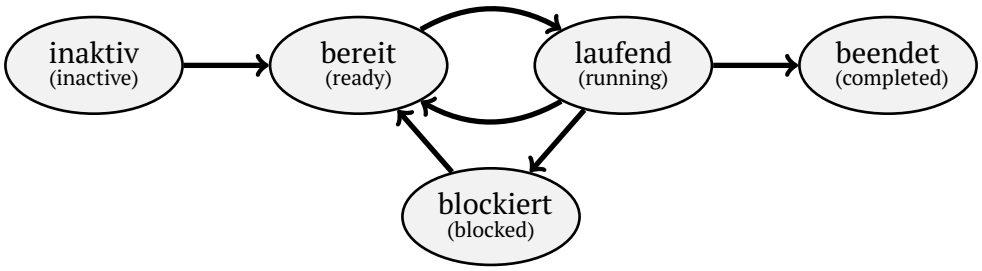
Der Scheduler ist dafür verantwortlich, laufende Prozesse anzuhalten (und damit in den Status „bereit“ zu versetzen), um anderen bereiten Prozessen das Laufen zu ermöglichen. Dieses Umschalten des Zustandes von einzelnen Prozess wird als „Context switch“ bezeichnet. Wie bereits vorher wird keine Annahme getroffen, wie sich der Scheduler verhält: Sämtliche „Interleavings“ sind möglich mit der einzigen Einschränkung, dass Fairness gelten muss, d.h. jeder bereite Prozess muss nach endlich vielen Berechnungsschritten in den Zustand „laufend“ versetzt werden. Um den Zustand von Prozessen kenntlich zu machen bzw. auch zu verändern, wird die Notation P.state verwendet, wobei P ein Prozess ist und das Attribut state den momentanen *Zustand* des Prozesses speichert. D.h. z.B. wenn der Scheduler einen laufenden Prozess p in den Zustand „bereit“ versetzt, dann verändert sich der Wert von p.state von running auf ready.

Für die Programmierkonstrukte, die in den folgenden Abschnitten dargestellt werden, wird

angenommen, dass es einen weiteren Zustand für Prozesse gibt: Der Zustand *blockiert* (engl. *blocked*). Ein blockierter Prozess ist nicht bereit, kann also vom Scheduler nicht in den Zustand „laufend“ gesetzt werden. Ein blockierter Prozess wird entblockiert (oder aufgeweckt), indem eine Operation ihn vom Zustand „blockiert“ in den Zustand „bereit“ versetzt. Diese Operation wird nicht durch den Scheduler durchgeführt, sondern durch eine Programmanweisung. Anschließend kann der (bereite) Prozess wieder vom Scheduler in den Zustand „laufend“ durch einen Context-Switch versetzt werden.

Man kann noch zwei weitere Zustände für Prozesse definieren: Ein Prozess ist am Anfang *inaktiv*, bis er irgendwann bereit ist. Wenn ein Prozess fertig ist, also keine weiteren Berechnungsschritte mehr durchführen will, so ist er *beendet* (engl. *completed*).

Insgesamt lassen sich die verschiedenen Zustände des Prozessstatus und deren mögliche Übergänge wie folgt in einem Diagramm zusammenfassen:



3.2 Semaphore

Der Begriff Semaphor stammt eigentlich aus dem Bahnverkehr, dort werden bzw. wurden sie verwendet als mechanische Signalgeber. In der Informatik ist ein Semaphor ein abstrakter Datentyp mit (meistens) zwei Komponenten:

- Eine nicht-negative Ganzzahl V.
- Eine Menge von Prozessen M (bzw. deren Identifikationen).

Sei S ein Semaphor, so bezeichnet $S.V$ die nicht-negative Ganzzahl und $S.M$ die Menge von Prozessen mit $S.M$.

Sei **`newSem`**(k) eine Operation, die einen neuen Semaphor S erzeugt und dabei $S.V$ mit k und $S.M$ mit der leeren Menge initialisiert. Für einen Semaphor S sind die zwei *atomaren* Operationen **`wait`** und **`signal`** verfügbar. Atomar meint hier, dass die Programmiersprache (der Compiler, Interpreter, Laufzeitumgebung) zusichert, dass die entsprechenden Operationen in einem unteilbaren Berechnungsschritt durchgeführt werden. Der Compiler selbst kann dabei für die Implementierung des Datentyps Semaphor die bekannten Primitiven der Rechnerarchitektur benutzen.

Bemerkung 3.2.1. *In Dijkstra's Originalartikel heißt die Operation nicht* **`wait`**(S)*, sondern* **`P`**(S) *und die Operation* **`signal`**(S) *heißt* **`V`**(S)*. Die Bezeichnungen* **`P`** *und* **`V`** *stammen von niederländischen Begriffen: „V“ steht für „verhoog“ (niederl. “erhöhe“), und „P“ steht für „prolaag“, welches ein von Dijsktra eingeführtes Kunstwort ist, da „verlaag“ (niederl. „erniedrige“) auch mit dem Buchstaben „V“ beginnt. Oft werden anstelle von* **`wait`** *und* **`signal`** *auch die Begriffe* **`down`** *und* **`up`** *verwendet.*

Sei P der Prozess, der **wait** bzw. **signal** für den Semaphor S aufruft, dann lässt sich die Semantik der beiden Operationen durch folgenden Programmcode beschreiben: Zunächst die **wait**-Operation:

```
procedure wait(S)
 if S.V > 0 then
   S.V := S.V − 1;
 else
   S.M := S.M ∪ {P};
   P.state := blocked;
```

Die **wait**-Operation liest zunächst die V-Komponente. Ist diese echt größer als 0, so wird sie um 1 dekrementiert. Andernfalls ($S.V$ ist 0) wird der aufrufende Prozess P zur Menge $S.M$ von Prozessen hinzugefügt und sein Status auf „blockiert“ gesetzt.

Die Prozedur für die **signal**-Operation kann beschrieben werden durch:

```
procedure signal(S)
 if S.M = ∅ then
   S.V := S.V + 1;
 else
   wähle ein Element Q aus S.M;
   S.M := S.M \ {Q};
   Q.state := ready;
```

Die **signal**-Operation entblockiert einen blockierten Prozess (und setzt dessen Zustand auf „bereit“), wenn blockierte Prozesse vorhanden sind, die Menge der Prozesse also nicht leer ist. Enthält die Menge mehrere Prozesse, so wird ein beliebiger Prozess aus der Menge gewählt. Gibt es keine blockierten Prozesse, so wird der Zähler erhöht.

Beachte, dass man für solche (generellen) Semaphore die **newSem**-Operation auch so definieren könnte, dass stets mit $k = 0$ initialisiert wird und anschließend (vor der eigentlichen Benutzung des Semaphors) k-mal **signal** aufruft (dadurch wird der Zähler auf k gesetzt).

Im folgenden wird eine Invariante gezeigt, die für (generelle) Semaphore gilt, die zunächst folgende Definition erfordert: Für eine endliche Auswertungsfolge $\mathcal{P}$ und ein Semaphor S ist $\#_{wait}(S, \mathcal{P})$ die Anzahl von **wait**(S)-Operationen in $\mathcal{P}$ und $\#_{signal}(S, \mathcal{P})$ ist die Anzahl von **signal**(S)-Operationen in $\mathcal{P}$.

Lemma 3.2.2. *Sei S ein genereller Semaphor, der mit der Zahl $k \geq 0$ initialisiert wurde. Dann gilt nach jeder endlichen Auswertungsfolge $\mathcal{P}$:*

- $S.V \geq 0$
- $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$

Beweis. Die erste Aussage ist trivial, da $S.V$ mit $k \geq 0$ initialisiert wird, **signal** den Wert $S.V$ erhöht oder unverändert lässt und **wait** den Wert $S.V$ nur dann um 1 erniedrigt, wenn er größer als 0 ist. Die zweite Aussage lässt sich per Induktion über die Länge von $\mathcal{P}$ zeigen: Ist $\mathcal{P}$ die leere Sequenz, so gilt offensichtlich $S.V = k$, da **newSem** so programmiert wurde, und des Weiteren muss gelten $S.M = \emptyset$, also $|S.M| = 0$. Für den Induktionsschritt sei $\mathcal{P}_1$ eine Sequenz der Länge

$n-1$. Als Induktionshypothese wird verwendet, dass die Aussage nach Ausführung von $\mathcal{P}_1$ gilt, d.h. $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}_1) - \#_{wait}(S, \mathcal{P}_1)$. Es ist zu zeigen, dass die Aussage für $\mathcal{P}_1 e$ gilt, wobei e eine weitere Operation sei.

- Hat die Operation e nichts mit dem Semaphor S zu tun, dann gilt die Aussage.
- Die Operation e ist eine **wait**(S)-Operation. Wenn $S.V$ vorher den Wert 0 hatte, dann wird $S.V$ nicht verändert, aber $|S.M|$ wächst um 1. Da dann gilt $\#_{wait}(S, \mathcal{P}_1 e) = 1 + \#_{wait}(S, \mathcal{P}_1)$ stimmt die Aussage. Wenn $S.V$ vorher größer 0 wahr, so wird $S.V$ um eins erniedrigt und die Gleichung stimmt wiederum.
- Die Operation e ist eine **signal**(S)-Operation. Dann wird entweder $S.V$ um 1 erhöht, oder $|S.M|$ um eins erniedrigt. Da dann gilt $\#_{signal}(S, \mathcal{P}_1 e) = 1 + \#_{signal}(S, \mathcal{P}_1)$, folgt die Aussage. □

Man unterscheidet generelle Semaphore, wie sie bisher dargestellt wurden, von *binären Semaphore*. Für binäre Semaphore darf das Wert-Attribut $S.V$ nur die Werte 0 und 1 annehmen. Die **newSem**-Operation kann den Semaphor mit Wert 0 oder Wert 1 initialisieren. Die **wait**-Operation funktioniert genauso wie vorher, aber die **signal**-Operation muss geändert werden (da für $S.V = 1$ der Wert nicht weiter erhöht werden kann). Genauer ist eine **signal**-Operation nicht erlaubt, wenn $S.V = 1$ gilt, d.h. die Operation führt zu einem Fehler oder undefinierten Zustand. Der Code für die **signal**-Operation eines binären Semaphors lautet somit:

```
procedure signal(S)
 if S.V = 1 then
   undefined
 else if S.M = ∅ then
        S.V := 1
      else
       wähle ein Element Q aus S.M;
       S.M := S.M \ {Q};
       Q.state := ready;
```

Binäre Semaphore werden auch oft als „mutex“ bezeichnet. Für binäre Semaphore gilt Lemma 3.2.2 nicht generell (da zu viele **signal**-Operationen in einen undefinierten Zustand führen können). Gibt es in der Folge $\mathcal{P}$ jedoch je eine **wait**-Operation bevor eine **signal**-Operation durchgeführt wird, so gilt das Lemma weiterhin. In diesem Fall gilt als weitere Invariante $S.V \leq 1$.

3.2.1 Mutual-Exclusion mithilfe von Semaphore

Mithilfe von binären Semaphore lässt sich das Mutual-Exclusion Problem sehr leicht lösen. In Abbildung 3.1 ist der Programmcode für den $i.$ von n Prozessen dargestellt.

Theorem 3.2.3. *Der Algorithmus aus Abbildung 3.1 garantiert wechselseitigen Ausschluss und ist Deadlock-frei.*

Beweis. Sei $\#_{KA}(\mathcal{P})$ die Anzahl von Prozessen im kritischen Abschnitt nach Ausführung der endlichen Sequenz $\mathcal{P}$.

Initial: S sei ein binärer Semaphor, initialisiert mit 1

Programm des i. Prozesses

```
loop forever
(1)  restlicher Code
(2)  wait(S)
(3)  Kritischer Abschnitt
(4)  signal(S)
end loop
```

Abbildung 3.1: Einfacher Mutual-Exclusion Algorithmus mit binärem Semaphor

Da der Algorithmus stets **wait** vor **signal** ausführt, gelten die Invarianten aus Lemma 3.2.2 und der undefinierte Zustand kann nicht erreicht werden. Offensichtlich gilt $\#_{KA}(\mathcal{P}) = \#_{wait}(S,\mathcal{P}) - \#_{signal}(S,\mathcal{P}) - |S.M|$: Alle Prozesse, die ein **wait** und **signal** durchgeführt haben, haben den kritischen Abschnitt wieder verlassen, von den restlichen Prozessen, die **wait** durchgeführt haben, aber noch kein **signal** müssen noch die wartenden Prozesse in $S.M$ abgezogen werden ($S.M$ meint hier die Menge nach Ausführung von $\mathcal{P}$). Mit der zweiten Invariante aus Lemma 3.2.2 ergibt das

$$\begin{aligned}\#_{KA}(\mathcal{P}) + S.V \quad &= \quad \#_{wait}(S,\mathcal{P}) - \#_{signal}(S,\mathcal{P}) - |S.M| \\ &\quad +k + |S.M| + \#_{signal}(S,\mathcal{P}) - \#_{wait}(S,\mathcal{P}) \quad = k\end{aligned}$$

Da der Semaphor mit $k = 1$ initialisiert wird, zeigt dies $\#_{KA}(\mathcal{P}) = 1 - S.V$, d.h. höchstens ein Prozess kann nach Ausführung einer beliebigen Sequenz im kritischen Abschnitt sein.

Für die Deadlock-Freiheit nehme zunächst das Gegenteil an. Sei $\mathcal{P}$ eine unendlich lange Auswertungsfolge, so dass mindestens ein Prozess in den kritischen Abschnitt will, aber keiner den kritischen Abschnitt betritt. Dann gibt es einen endlichen Präfix $\mathcal{P}_1$ von $\mathcal{P}$, so dass kein Prozess im kritischen Abschnitt ist ($\#_{KA}(\mathcal{P}_1) = 0$) und für jeden längeren Präfix von $\mathcal{P}$ dies auch so bleibt. Zusätzlich muss ein Prozess durch **wait** für immer blockiert sein (d.h. $S.V = 0$ gilt ab $\mathcal{P}_1$). Das ist allerdings unmöglich, da die gerade bewiesene Gleichung $\#_{KA}(\mathcal{P}) = 1 - S.V$ verletzt ist. □

Beachte, dass der Algorithmus für beliebig viele Prozesse keine Starvation-Freiheit garantiert. Der folgende Ablauf deutet dies für 3 Prozesse an, wobei Prozess 3 verhungert.

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	restlicher Code	restlicher Code	restlicher Code
1	$\emptyset$	**wait**(S)	restlicher Code	restlicher Code
0	$\emptyset$	Kritischer Abschnitt	**wait**(S)	restlicher Code
0	$\{2\}$	Kritischer Abschnitt	(blockiert)	**wait**(S)
0	$\{2,3\}$	Kritischer Abschnitt	(blockiert)	(blockiert)
0	$\{2,3\}$	**signal(**S**)**	(blockiert)	(blockiert)
0	$\{3\}$	restlicher Code	Kritischer Abschnitt	(blockiert)
0	$\{1,3\}$	**wait**(S)	Kritischer Abschnitt	(blockiert)
0	$\{1,3\}$	(blockiert)	**signal**(S)	(blockiert)
0	$\{3\}$	Kritischer Abschnitt	restlicher Code	(blockiert)
0	$\{3\}$	Kritischer Abschnitt	**wait**(S)	(blockiert)
0	$\{2,3\}$	Kritischer Abschnitt	(blockiert)	(blockiert)
...				

Man kann allerdings nachweisen, dass der Algorithmus bei maximal 2 nebenläufigen Prozessen Starvation-frei ist.

3.2.2 Weitere Varianten von Semaphore

Die bisher definierten Semaphore werden im Allgemeinen als *schwache Semaphore* (engl. weak semaphores) bezeichnet, da die Auswahl des zu entblockierenden Prozesses bei einer **signal**-Operation beliebig ist. Im Gegensatz dazu gibt es so genannte *starke Semaphore* (engl. strong semaphores), die die wartenden (blockierten) Prozesse in first-in-first-out Reihenfolge abarbeiten. Für deren Implementierung wird anstelle der Menge $S.M$ von Prozessen eine Warteschlange (bzw. einfach eine Liste) $S.L$ verwendet. Bei der Initialisierung des Semaphors wird die leere Liste erzeugt. Die Implementierung der Operationen wird dann verändert in:

```
procedure wait(S)
 if S.V > 0 then
   S.V := S.V − 1;
 else
   S.L := append(S.L, P);
   P.state := blocked;
```

```
procedure signal(S)
 if isEmpty(S.L) then
   S.V := S.V + 1;
 else
   Q := head(S.L);
   S.L := tail(S.L);
   Q.state := ready;
```

Hierbei fügt *append* ein Element an eine Liste hinten an, *head* liefert das erste Listenelement einer Liste, *tail* liefert die Restliste ohne das erste Element und *isEmpty* prüft, ob eine Liste leer ist.

Verwendet man starke Semaphore im Mutual-Exclusion-Algorithmus aus Abbildung 3.1, so garantiert der Algorithmus Starvation-Freiheit und die FIFO-Eigenschaft.

Eine weitere Klasse von Semaphore sind so genannte *unfaire Semaphore*. Diese sind noch schwächer als schwache Semaphore, da überhaupt keine Annahme darüber getroffen wird, wann ein blockierter Prozess entblockiert wird. Genauer: Diese Semaphore lassen durch busy-wait implementieren, und verfügen nicht über die $S.M$ Komponente. Die Operationen können z.B. mit einem fetch-and-add-Objekt implementiert werden:

procedure wait(S**)**
await $S.V > 0$;
$S.V := S.V - 1$;

procedure signal(S**)**
$S.V := S.V + 1$;

Benutzt man einen solche unfairen Semaphor im vorherigen Mutual-Exclusion-Algorithmus, so ist dieser selbst für zwei Prozesse nicht mehr Starvation-frei.

3.3 Semaphore in Java

Das Package `java.util.concurrent` stellt die Klasse `Semaphore` bereit. Der Konstruktor für Semaphore-Objekte erwartet dabei eine Ganzzahl. Mit dieser Zahl wird der Semaphor initialisiert. Im Gegensatz zur abstrakten Darstellung von generellen Semaphore ist es in Java erlaubt den Semaphor mit einem negativen Wert zu initialisieren. In diesem Fall müssen zunächst **`signal`**-Operationen ausgeführt werden, um den Wert auf 0 oder einen positiven Wert zu erhöhen. In Java haben **`wait`** und **`signal`** andere Namen: Die Methode **`acquire`** entspricht der **`wait`**-Operation und anstelle von **`signal`** gibt es die Methode **`release`**.

Über den Konstruktor kann noch ein zweites Argument beim Erzeugen eines Semaphors übergeben werden. Dieses Argument ist ein boolescher Wert. Ist der Wert **`True`**, so verhält sich der Semaphor wie ein starker Semaphor (in Java als fairer Semaphor bezeichnet). Andernfalls wird ein Semaphor ohne explizite Menge von wartenden Prozessen erzeugt, die sich mehr oder weniger wie ein unfairer (busy-wait) Semaphor verhält.

Ein kleines Beispiel (übernommen aus (Ben-Ari, 2006)) ist:

```
import java.util.concurrent.Semaphore;
class CountSem extends Thread {
  static volatile int n = 0; // globale atomare Variable
  static Semaphore s = new Semaphore(1);

  public void run() {
    int temp;
    for (int i = 0; i < 10; i++) {
      try {
        s.acquire();
      }
      catch (InterruptedException e) {}
      temp = n;
      n = temp + 1;
      s.release();
    }
  }
```

```
  public static void main(String[] args) {
      CountSem p = new CountSem();
      CountSem q = new CountSem();
      p.start(); // startet Thread p
      q.start(); // startet Thread q
      try {
        p.join();// wartet auf Terminierung von Thread p
        q.join();// wartet auf Terminierung von Thread q
      }
      catch (InterruptedException e) { }
      System.out.println("The value of n is " + n);
    }
}
```

Die Klasse **CountSem** wird von Thread abgeleitet. Es wird eine Klassenvariable n definiert (der Counter). Das Schlüsselwort volatile sorgt dafür, dass die Variable zwischen allen Threads aktualisiert wird, wenn sie verändert wird. (Im Grunde wird damit ein atomares Register definiert). Als zweites Attribut besitzt die Klasse einen Semaphor s, der mit 1 initialisiert wird. Die **run**-Methode muss für jeden Thread definiert werden. Innerhalb der **run**-Methode wird 10-mal der Zähler n erhöht, wobei der Zugriff auf n durch den Semaphor geschützt wird (da **acquire** zu einer Exception führen kann, muss diese mit dem **try**...**catch**-Block abgefangen werden). In der **main**-Methode werden zwei Objekte der Klasse CountSem erzeugt und gestartet. Schließlich wird unter Verwendung der vordefinierten **join**-Methode darauf gewartet, dass beide Threads beendet sind und anschließend der Wert von n ausgedruckt.

3.4 Anwendungsbeispiele für Semaphore

In diesem Abschnitt werden einige klassische Problemstellungen der nebenläufigen Programmierung betrachtet und Lösungen (d.h. Implementierungen) dieser Probleme unter Zuhilfenahme von Semaphore erörtert.

3.4.1 Koordination der Reihenfolge am Beispiel Mergesort

Will man nebenläufige oder parallele Programme schreiben, so kann man meistens nicht alle Schritte parallel durchführen. Oft müssen einzelne Schritte sequentiell ohne Interleaving durchgeführt werden. Dieses Problem wurde mit den Mutual-Exclusion-Algorithmen bereits gelöst. Allerdings erfordern manche Problemstellungen, dass ein einzelner Prozess erst dann los rechnen darf, wenn andere Prozesse die Berechnungen ihrerseits abgeschlossen haben. Betrache als Beispiel den Mergesort. Es sei daran erinnert, dass der Mergesort-Algorithmus ein rekursiver Algorithmus ist, der grob wie folgt funktioniert (als sequentieller Algorithmus):

- Solange noch mehr als ein Element sortiert werden muss:
  - Teile die Eingabefolge in zwei gleich große Hälften
  - Sortiere beide Hälften durch rekursiven Aufruf des Algorithmus
  - Mische die beiden sortierten Hälften in eine sortierte Gesamtfolge

Will man diesen Algorithmus parallelisieren, so ist relativ offensichtlich, dass man die beiden rekursiven Aufrufe parallel (oder nebenläufig) durchführen kann. Ein ähnliches Vorgehen zur Parallelisierung ist bei fast allen Divide&Conquer-Algorithmen möglich. Es ist jedoch auch offensichtlich, dass der Schritt des Mischens der sortierten Teilfolgen erst dann durchgeführt werden kann, wenn das (rekursive) Sortieren der beiden Hälften beendet ist. Der mischende Prozess muss somit warten, bis die rekursiven Sortierprozesse ihre Arbeit beendet haben. Mittels zweier binärer Semaphore kann genau diese Koordination durchgeführt werden. Abbildung 3.2 zeigt die Implementierungen für den merge-Prozess (der mischende Prozess) und für den Code für die beiden rekursiven Prozesse, welche die beiden Teilhälften sortieren. Der genaue Code zum Mischen und Teilen der Folgen wurde hierbei ausgespart, da hier nur die Koordination mithilfe der Semaphore interessiert.

Initial: left, right: Binäre Semaphore mit 0 initialisiert

Prozess für linke Hälfte:	Prozess für rechte Hälfte:
(1) sortiere linke Hälfte;	(1) sortiere rechte Hälfte;
(2) **signal**(left);	(2) **signal**(right);

merge-Prozess:
(1) **wait**(left);
(2) **wait**(right);
(3) mische die beiden Hälften;

Abbildung 3.2: Koordination des Merge-Sort mithilfe von Semaphore

Die Funktionsweise der Implementierung ist einfach zu beschreiben: Da die beiden binäre Semaphore initial mit 0 belegt sind, wird der merge-Prozess an beiden Semaphoren blockiert. Sobald beide Sortierprozesse für die beiden Hälften ihre **signal**-Operation durchgeführt haben, wird der merge-Prozess wieder bereit und kann mischen.

Es sei noch anzumerken, dass diese Implementierung des Mergesorts zwei Semaphore pro Rekursionsschritt benötigt, also bei n zu sortierenden Elementen $O(\log\ n)$ binäre Semaphore benötigt.

3.4.2 Erzeuger-Verbraucher Probleme

Das Erzeuger-Verbraucher-Problem kennzeichnet sich dadurch, dass man die einzelnen Prozesse zwei Gruppen zuordnen kann. Die Erzeuger produzieren Daten, welche von der anderen Gruppe – den Verbrauchern – weiter verarbeitet werden. Klassische solche Szenarien sind z.B. die Eingaben-produzierende Tastatur und das Betriebssystem, welches die Tastatureingaben weiterverarbeitet. Ein weiteres Beispiel ist der Webbrowser, der mit dem Webserver kommuniziert. In diesem Fall können beide Prozesse sowohl als Erzeuger sowie als Verbraucher auftreten. Um die Kommunikation zwischen Erzeuger und Verbraucher reibungslos sicherzustellen, wird oft ein so genannter Pufferspeicher verwendet, der im Wesentlichen eine Liste oder Queue darstellt, auf die der Erzeuger seine Daten ablegt und von der der Verbraucher die Daten herunter liest. Da die Kommunikation zwischen Erzeuger und Verbraucher nur selten synchron geschieht, ist es oft ratsam einen solchen Zwischenspeicher zu verwenden. Wenn der Erzeu-

ger allerdings stets viel schneller erzeugt, als der Verbraucher die Daten verbraucht, dann sollte die Größe dieses Speichers beschränkt sein, um den Erzeuger manchmal zu stoppen (ansonsten würde der Speicher voll laufen). Es gibt auch Anwendungen, die einen solchen Zwischenspeicher benötigen, da Daten immer nur in größeren Blöcken (oder im Gesamten) vom Verbraucher verarbeitet werden können. Lädt der Webbrowser beispielsweise eine zip-Datei aus dem Internet, so kann das Entpackprogramm erst nach dem vollständigen Download mit dem Entpacken anfangen. Die zip-Datei könnte also z.B. in einem Pufferspeicher zwischengelagert werden.

Die Schwierigkeit beim Erzeuger/Verbraucher-Problem besteht darin, den Zugriff auf den Pufferspeicher zu schützen. Je nach Art des verwendeten Pufferspeicher müssen verschiedene Anforderungen hierbei erfüllt werden. Man unterscheidet zwischen beliebig großen Puffern (engl. infinite buffer) und Puffern mit begrenztem Platz (engl. bounded buffer).

Es zunächst das Erzeuger/Verbraucher-Problem mit einem infinite buffer betrachtet. In diesem Fall muss zum Einen sichergestellt werden, dass auf den Puffer atomar lesend und schreibend zugegriffen wird. Zum Anderen muss implementiert werden, dass der Verbraucher-Prozess beim Lesen eines leeren Puffers wartet, bis der Puffer wieder mit Daten gefüllt ist. In Abbildung 3.3 ist eine Implementierung des Erzeuger/Verbraucher-Problems mit beliebig großem Pufferspeicher dargestellt. Sie benutzt zwei Semaphore: Der binäre Semaphor mutex sichert hierbei zu, dass Lesen und Schreiben stets atomar durchgeführt wird. Der Semaphor **`notEmpty`** ist ein genereller Semaphor, er wird benutzt, um die Größe des Pufferspeichers zu speichern und den Verbraucher im Falle eines leeren Puffers zu blockieren.

Initial: notEmpty: Genereller Semaphor, initialisiert mit 0
mutex: Binärer Semaphor, initialisiert mit 1
l: Liste

Erzeuger (erzeugt e):

```
(1) erzeuge e;
(2) wait(mutex);
(3) l := append(l,e);
(4) signal(notEmpty);
(5) signal(mutex);
```

Verbraucher (verbraucht e):

```
(1) wait(notEmpty);
(2) wait(mutex);
(3) e := head(l);
(4) l := tail(l);
(5) signal(mutex);
(6) verbrauche e;
```

Abbildung 3.3: Semaphore-Implementierung von Erzeuger/Verbraucher mit infinite buffer

Mittels mutex wird sichergestellt, dass immer nur ein Prozess auf die Liste zugreifen kann. Der Verbraucher muss zunächst am Semaphor notEmpty prüfen, ob Elemente im Puffer vorhanden sind. Ist dies der Fall, so erniedrigt die **`wait`**-Operation den Wert notEmpty.V um 1 und der Verbraucher darf sich das erste Element der Liste nehmen. Ist der Wert von notEmpty.V gleich 0, dann wird der Verbraucher blockiert: Er muss warten, bis ein Erzeuger-Prozess eine **`signal`**-Operation für den Semaphor notEmpty durchführt. Der Erzeuger hängt sein erzeugtes Element hinten an die Liste an (dies ist immer möglich, da der Puffer beliebig groß werden darf) und signalisiert anschließend, dass die Liste nicht leer ist.

Unter der Annahme, dass die Liste am Anfang leer ist, lässt sich leicht nachvollziehen, dass die Gleichung notEmpty.V = length(l) (wobei length die Länge der Liste berechnet) stets erfüllt ist,

also eine Invariante der Pufferspeicher-Implementierung ist.

Nun wird ein Pufferspeicher begrenzter Größe betrachtet. In diesem Fall muss neben den bisherigen Anforderungen (Lesen und Schreiben auf den Puffer atomar, und Schutz bei Zugriff eines Verbrauchers auf den leeren Puffer) zusätzlich sichergestellt werden, dass ein Erzeuger nicht in einen vollen Puffer schreiben kann und stattdessen warten muss, bis der Puffer nicht mehr voll ist. Abbildung 3.4 zeigt eine Implementierung für diesen Fall. Es wird ein zusätzlicher genereller Semaphor notFull benutzt, der initial mit der maximalen Größe des Puffers N erstellt wird.

Initial: notEmpty: Genereller Semaphor, initialisiert mit 0
notFull: Genereller Semaphor, initialisiert mit N
mutex: Binärer Semaphor, initialisiert mit 1
l: Liste

Erzeuger (erzeugt e):

```
(1) erzeuge e;
(2) wait(notFull);
(3) wait(mutex);
(4) l := append(l,e);
(5) signal(notEmpty);
(6) signal(mutex);
```

Verbraucher (verbraucht e):

```
(1) wait(notEmpty);
(2) wait(mutex);
(3) e := head(l);
(4) l := tail(l);
(5) signal(notFull);
(6) signal(mutex);
(7) verbrauche e;
```

Abbildung 3.4: Semaphore-Implementierung von Erzeuger/Verbraucher mit bounded buffer

Gegenüber der Implementierung des infinite buffers wurde nur wenig geändert: Der zusätzliche Semaphor notFull überwacht den Aufruf des Erzeugers. Hat notFull.V den Wert 0, so ist der Puffer vollständig belegt und der Erzeuger wird blockiert. Der Verbraucher signalisiert, dass er ein Element aus dem Puffer entfernt hat, durch Ausführen der **signal**(notFull)-Operation.

Als Invariante der beiden Semaphore lässt sich leicht zeigen, dass stets gilt notEmpty.V + notFull.V = N. Eine solche Datenstruktur aus zwei Semaphore, die quasi wie ein Semaphor „in beide Richtungen“ wirkt, wird im Allgemeinen als „Split-Semaphor“ bezeichnet.

3.4.3 Die speisenden Philosophen

Das Problem der speisenden Philosophen ist wohl das bekannteste klassische Problem aus der nebenläufigen Programmierung. Die Formulierung mithilfe der Philosophen stammt von Tony Hoare.

Für dieses Problem sitzen mehrere Philosophen um einen runden Tisch herum. Jeder Philosoph hat einen Teller, in der Mitte des Tisches befindet sich eine große Schüssel mit Spaghetti. Jeder Philosoph symbolisiert einen Prozess. Ein Philosoph denkt und isst abwechselnd. Er führt dies unendlich lange durch. Die Problematik ergibt sich daraus, dass ein einzelner Philosoph zum Essen zwei Gabeln benötigt (eine in der linken Hand, eine in der rechten Hand), aber auf dem Tisch stets nur eine Gabel zwischen zwei Philosophen liegt. Abbildung 3.5 illustriert die Situation für fünf Philosophen.

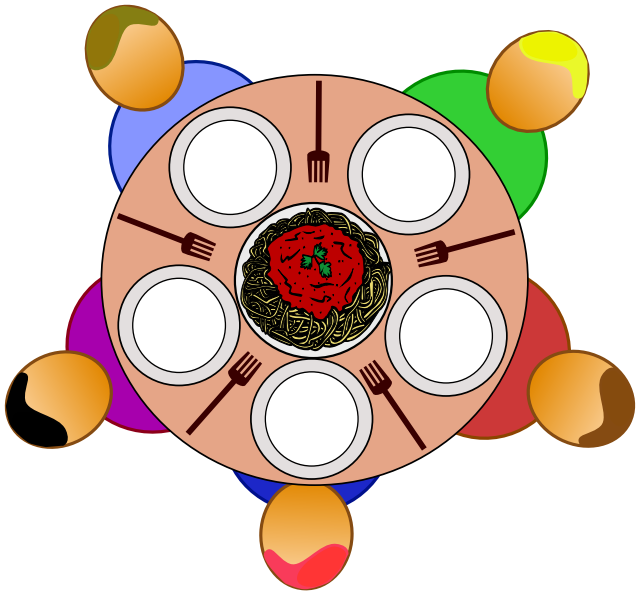


Abbildung 3.5: Fünf speisende Philosophen

Eine wichtige Annahme beim Problem der speisenden Philosophen ist, dass ein Philosoph nur eine Gabel zu einer Zeit nehmen kann, oder umgekehrt formuliert, er kann nicht beide Gabeln gleichzeitig aufnehmen.

Lösungen des Problems sollten die Deadlock-Freiheit garantieren, d.h. zumindest irgendein Philosoph kann unendlich oft zwischen Denken und Essen wechseln. Als zweite Anforderungen sollte eine Art Starvation-Freiheit garantiert werden, wobei Starvation hier wirklich wörtlich gemeint sein kann: Kein Philosoph verhungert, d.h. nach endlich vielen Schritten isst jeder Philosoph.

Für die Modellierung des Problems seien die N Philosophen von 1 bis N durchnummeriert. Die Gabeln sind durch binäre Semaphore modelliert (am Anfang mit 1 initialisiert). Für den Philosoph mit Nummer i seien gabel[i] seine linke Gabel und gabel[$i+1$] seine rechte Gabel (die Addition erfolge hierbei implizit modulo N).

Die Lösung des Philosophen-Problems besteht nun darin ein Programm für den Philosophen i anzugeben. Ein erster Versuch ist in Abbildung 3.6 dargestellt. Jeder Philosoph versucht zunächst seine linke und anschließend seine rechte Gabel zu nehmen (mittels der **`wait`**-Operation). Nachdem der Philosoph gegessen hat, legt er seine beiden Gabeln nacheinander ab (mittels der **`signal`**-Operation). Dieser Ansatz funktioniert allerdings *nicht*, wie der folgende Beispiel-Ablauf für drei Philosophen zeigt:

gabeln: Feld (Dimension $1 \ldots N$) von binären Semaphore, initial alle mit 1 initialisiert

Philosoph i:
loop forever
(1) Philosoph denkt;
(2) **wait**(gabel[i]); // linke Gabel
(3) **wait**(gabel[i+1]);// rechte Gabel
(4) Philosoph isst
(5) **signal**(gabel[$i+1$]);
(6) **signal**(gabel[i]);
end loop

Abbildung 3.6: Versuch zur Lösung des Philosophen-Problems

Philosoph 1	Philosoph 2	Philosoph 3
wait(gabeln[1]) „hat linke Gabel“		
	wait(gabeln[2]) „hat linke Gabel“	
		wait(gabeln[3]) „hat linke Gabel“
wait(gabeln[2]) blockiert		
	wait(gabeln[3]) blockiert	
		wait(gabeln[1]) blockiert

Alle Philosophen schaffen es, nacheinander ihre linke Gabel aufzunehmen. Anschließend liegen keine Gabeln mehr auf dem Tisch. Da nun alle Philosophen versuchen, ihre rechte Gabel aufzunehmen, werden diese nacheinander blockiert. Es ist ein Deadlock entstanden, da kein Philosoph mehr entblockiert werden kann.

Eine Möglichkeit solche Deadlocks zu verhindern, ist das Benutzen eines zusätzlichen binären Semaphors mutex, der den Zugriff auf die Gabeln exklusiv regelt. In Abbildung 3.7 ist der entsprechende Code für den $i.$ Philosophen dargestellt. Bevor ein Philosoph die Gabeln anfassen darf, muss er durch den Semaphor mutex hindurch kommen. Anschließend nimmt er die Gabeln, isst, und legt die Gabeln ab. Danach gibt er den Semaphor mutex frei.

Es ist leicht einzusehen, dass diese Lösung Deadlock-frei ist, da stets nur ein Philosoph Zugriff auf *alle* Gabeln hat. Der Algorithmus ist allerdings nur dann Starvation-frei, wenn mutex ein starker Semaphor ist, die die FIFO-Eigenschaft erfüllt. Ist mutex ein schwacher Semaphor, kann ein Philosoph verhungern, da er beim blockiert sein stets von anderen Philosophen „überholt“ werden kann. Ein weiteres Problem bei diesem Lösungsvorschlag besteht darin, dass höchstens ein Philosoph gleichzeitig essen kann. Umgekehrt bedeutet dies auch, dass $N-2$ Gabeln stets auf dem Tisch liegen. Überträgt man dies auf reale Probleme, so findet hier Ressourcenverschwendung statt: Nur um Korrektheit der Implementierung sicherzustellen, werden $N-2$ Ressourcen „blockiert“, obwohl sie nicht verwendet werden. Aus diesem Grund ist, diese Lösung im Allgemeinen nicht akzeptabel. Betrachtet man jedoch den Fall, dass nur zwei Philosophen am Tisch sitzen, so kann man folgern: Die Ressourcen werden gut genutzt und selbst bei der Verwendung eines schwachen Semaphors für mutex gilt Starvation-Freiheit.

gabeln: Feld (Dimension $1 \ldots N$) von binären Semaphore, initial alle mit 1 initialisiert
mutex: Binärer Semaphor, mit 1 initialisiert

Philosoph i:
loop forever
(1) Philosoph denkt;
(2) **wait**(mutex);
(3) **wait**(gabel[i]); // linke Gabel
(4) **wait**(gabel[i+1]);// rechte Gabel
(5) Philosoph isst
(6) **signal**(gabel[i+1]);
(7) **signal**(gabel[i]);
(8) **signal**(mutex);
end loop

Abbildung 3.7: 2. Versuch zur Lösung des Philosophen-Problems

Die nächste Lösung versucht die „guten“ Eigenschaften der vorherigen Lösung für 2 Prozesse auf N Prozesse zu erweitern. Es sollen möglichst viele Gabeln zur gleichen Zeit benutzt werden und Starvation-Freiheit soll auch dann noch gelten, wenn nur schwache Semaphore zur Verfügung stehen. Der Code in Abbildung 3.8 zeigt den nächsten Versuch. Die Idee dabei ist, dass es einen Wächter gibt, der höchstens $N-1$ Philosophen den gleichzeitigen Zugriff auf die Gabeln (bzw. den Tisch) ermöglicht. Selbst wenn alle $N-1$ Prozesse zunächst die linke Gabel aufnehmen, muss es stets einen Prozess geben der auch seine rechte Gabel anschließend aufnehmen kann. Die zugehörige rechte Gabel ist genau die linke Gabel desjenigen Philosophen, der nicht in den Raum darf. Mit dieser Begründung ist leicht zu zeigen, dass kein Deadlock auftreten kann. Der Wächter ist mithilfe des generellen Semaphors raum implementiert. Dieser Semaphor wird mit dem Wert $N-1$ initialisiert, so dass der N. Prozess, der den Raum betreten möchte, blockiert wird.

Selbst wenn raum ein schwacher Semaphor ist, bleibt diese Lösung Starvation-frei: Es kann höchstens einer der N Prozesse gleichzeitig blockiert sein. Dieser eine Prozess muss bei der nächsten **signal**(raum)-Operation (die selbst auch stattfinden muss) entblockiert werden. Innerhalb des Raumes können pro Gabel höchstens zwei Prozesse den Zugriff fordern (für einen ist dies die linke, für den anderen die rechte Gabel). Sobald ein solcher Prozess blockiert wird (die Gabel hat bereits der andere Prozess), ist sicher, dass dieser Prozess die Gabel nach endlich vielen Schritten erhalten wird (Erinnerung: binäre, schwache Semaphore garantieren bei 2 Prozessen die Starvation-Freiheit). Hierbei muss man nur beachten, dass kein „globaler Deadlock“ (also eine zyklische Abhängigkeit der angeforderten Ressourcen) auftreten kann. Genau dies wurde bereits durch die Verwendung von raum ausgeschlossen.

Im Folgend wird eine weitere Möglichkeit aufgezeigt, den globalen Deadlock unmöglich zu machen. Hierfür muss man beachten, dass der Deadlock (alle haben die linke Gabel) dadurch auftritt, dass die Ressourcen zyklisch voneinander abhängen. Die Lösung in Abbildung 3.9 durchbricht diese Abhängigkeit, indem der N. Prozess umgekehrt zu allen anderen Prozessen vorgeht: Er versucht zunächst seine *rechte* Gabel und anschließend im nächsten Schritt erst die *linke* Gabel aufzunehmen. Man kann zeigen, dass auch für diesen Algorithmus weder Dead-

Initial alle Gabeln mit 1 initialisiert
raum: genereller Semaphor, mit $N-1$ initialisiert

Philosoph i
```
loop forever
(1) Philosoph denkt;
(2) wait(raum);
(3) wait(gabel[i]); // linke Gabel
(4) wait(gabel[i+1]);// rechte Gabel
(5) Philosoph isst
(6) signal(gabel[i+1]);
(7) signal(gabel[i]);
(8) signal(raum);
end loop
```

Abbildung 3.8: Lösung des Philosophen-Problems mit Wächter

locks noch Starvation möglich sind. Die Deadlock-Freiheit ist eine direkte Konsequenz des sogenannten „Theorems der Totalen Ordnung“, welches später betrachtet wird (allerdings erst im anschließenden Kapitel).

Initial alle Gabeln mit 1 initialisiert

Philosoph i, $i < N$:
```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i]); // linke Gabel
(3) wait(gabel[i+1]);// rechte Gabel
(4) Philosoph isst
(5) signal(gabel[i]+1);
(6) signal(gabel[i]);
end loop
```

Philosoph N
```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i+1]); // rechte Gabel
(3) wait(gabel[i]);// linke Gabel
(4) Philosoph isst
(5) signal(gabel[i]);
(6) signal(gabel[i+1]);
end loop
```

Abbildung 3.9: Lösung des Philosophen-Problems mit asymmetrischem N.-Prozess

3.4.4 Das Sleeping-Barber-Problem

Das Sleeping-Barber-Problem besteht aus der in Abbildung 3.10 dargestellten Situation eines Friseurladens.

Der Laden hat $N-1$ Warteplätze, einen Frisierstuhl und einen Friseur. Der Friseur und die Kunden sind als Prozesse zu modellieren, wobei folgende Bedingungen gelten müssen: Wenn kein Kunde im Laden ist, dann schläft der Friseur. Der erste Kunde im Laden weckt den Friseur und lässt sich frisieren. Sind mehrere Kunden im Laden, so nimmt ein neuer Kunde auf einem der Warteplätze Platz. Sind alle Warteplätze belegt, so verlässt der Kunde den Laden sofort wie-

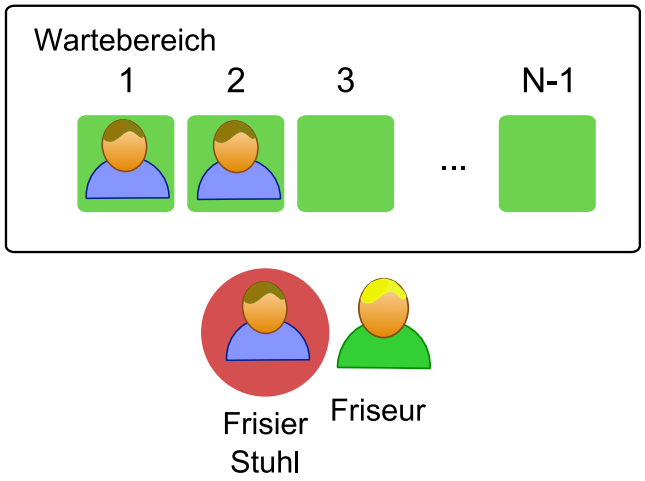


Abbildung 3.10: Situation des Sleeping-Barber-Problem

der. Nachdem der Friseur einem Kunden eine neue Frisur erschaffen hat, wartet der Friseur bis der Kunde den Laden verlassen hat, anschließend nimmt der Friseur den nächsten wartenden Kunden dran.

Eine Modellierung und Lösung des Sleeping-Barber-Problems mithilfe von Semaphore ist in Abbildung 3.11 dargestellt. Das atomare Register wartend zählt die Anzahl der wartenden Kunden (am Anfang 0), der generelle Semaphor kunden ist ähnlich, auch er zählt die Anzahl der gesamten Kunden und wird benutzt, um den Friseur zu wecken. Der binäre Semaphor mutex dient dem Zweck, Speicheränderungen (z.B. am Register wartend) geschützt (also unteilbar) durchzuführen. Der Semaphor synch wird benutzt, um den Friseur darauf warten zu lassen, bis der Kunde den Laden verlässt, bevor der Friseur den nächsten Kunden bedient. Der Semaphor friseur dient schließlich dazu, immer einen der wartenden Kunden dem Friseur zu übergeben, sobald der Frisierstuhl wieder frei ist.

Das Programm eines Kunden kann wie folgt erläutert werden: In Zeile (1) wird der Semaphor mutex belegt, um sicheren Zugriff auf die Variable wartend zu erlangen (diese Ressource wird dann entweder in Zeile (5) oder in Zeile (9) wieder freigegeben). In Zeile (2) prüft der Kunde, ob der Laden zu voll ist. Ist dies der Fall, springt er in Zeile (9) und verlässt den Laden (bzw. auch den Algorithmus) sofort wieder. Andernfalls wird zunächst die Anzahl der wartenden Kunden erhöht und zudem auch der generelle Semaphor kunden mittels der **`signal`**-Operation erhöht. Wenn der aktuelle Kunde der erste Kunde im Laden ist, dann wird der Friseur am Semaphor kunden warten und wird durch die **`signal`**-Operation des Kunden geweckt (entblockiert). In Zeile (5) wird der mutex wieder freigegeben (damit können weitere Kunden den Laden betreten). In Zeile (6) wartet der Kunde a, Semaphor friseur, bis der Friseur ihn bedient (der Friseur wird ihm das mit einer **`signal`**-Operation mitteilen). Nachdem der Kunde in Zeile (7) seine Frisur erhalten hat, verlässt er in Zeile (8) den Laden, wobei er dies wiederum dem Friseur durch eine **`signal`**-Operation für den Semaphor synch signalisiert.

Aus der Sicht des Friseurs: Der Friseur läuft im Gegensatz zum Kunden in einer Endlosschleife (er frisiert die ganze Zeit, während ein Kunde sich mit einer Frisur begnügen muss). Zunächst wartet der Friseur am Semaphor kunden. Wenn keine Kunden da sind, dann wird der Friseur

Initial: wartend: atomares Register, am Anfang 0
kunden: genereller Semaphor, am Anfang 0
mutex: binärer Semaphor, am Anfang 1
synch,friseur: binärer Semaphor am Anfang 0

Friseur:

```
loop forever
(1)  wait(kunden);//schlafe, solange keine Kunden da sind
(2)  wait(mutex);
(3)  wartend := wartend -1;
(4)  signal(friseur);//nehme nächsten Kunden
(5)  signal(mutex);
(6)  schneide Haare;
(7)  wait(synch);//warte, bis Kunde Laden verlässt
end loop
```

Kunde:

```
(1)  wait(mutex);
(2)  if wartend < N then
(3)    wartend := wartend + 1;
(4)    signal(kunden);//Wecke Friseur (bzw. erhöhe Kunden)
(5)    signal(mutex);
(6)    wait(friseur);//Warte bis Friseur bereit
(7)    erhalte Frisur;
(8)    signal(synch);//verlasse Laden
(9)  else signal(mutex);//gehe sofort
```

Abbildung 3.11: Lösung des Sleeping Barber Problems mit Semaphore

blockiert, d.h. er schläft. Nachdem der Friseur geweckt wurde, benutzt er in den Zeilen (2) bis (5) den Semaphor mutex, um die Variable wartend atomar zu ändern: In Zeile (3) erniedrigt er die Anzahl der wartenden Kunden, da er in Zeile (4) den nächsten Kunden dran nimmt (dies signalisiert er über den Semaphor friseur). In Zeile (6) erschafft er die neue Frisur des Kunden, um schließlich in Zeile (7) darauf zu warten (am Semaphor synch), dass der gerade bediente Kunde den Laden verlässt.

3.4.5 Das Cigarette Smoker's Problem

Beim Cigarette Smoker's Problem gilt es das folgende Problem als nebenläufiges Programm korrekt zu modellieren: Es gibt drei Raucher und einen Agenten. Zum Rauchen einer Zigarette werden drei Zutaten benötigt: Tabak, Papier und ein Streichholz. Jeder der drei Raucher hat allerdings nur genau eine der Zutaten (davon hat er unendlich viel). Der Agent hat alle drei Zutaten. Der Agent wählt stets zwei der Zutaten zufällig aus und legt sie auf den Tisch. Die Raucher versuchen nun eine Zigarette unter Benutzung der Zutaten auf dem Tisch zu rauchen. Hierbei müssen Sie die Zutaten nacheinander vom Tisch nehmen. Der Agent legt erst dann wieder Zutaten auf den Tisch, wenn die vorherigen verbraucht wurden. Die Grafik in Abbildung 3.12 stellt die Situation symbolisch dar.

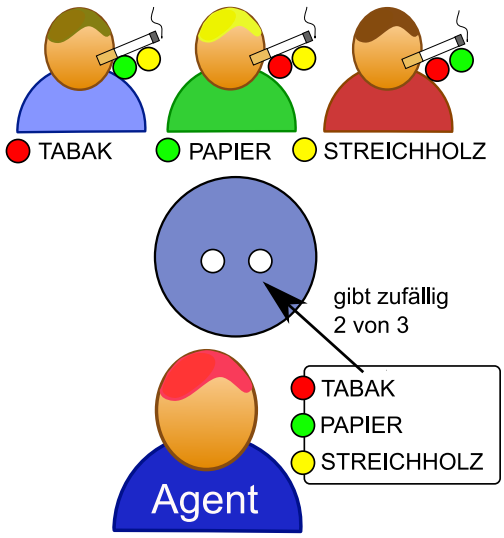


Abbildung 3.12: Situation beim Cigarette Smoker's Problem

Das Programm des Agenten ist durch die Problemstellung gegeben und in Abbildung 3.13 dargestellt, wobei vier Semaphore verwendet werden: S[1], S[2], S[3] sind binäre Semaphore für Tabak, Papier und Streichholz. Initial sind die Semaphore mit 0 initialisiert. Ist einer dieser Semaphore mit 1 belegt, so bedeutet dies, dass die entsprechende Zutat auf dem Tisch liegt. Der Agent benutzt einen weiteren Semaphor agent, der angibt, ob der Agent wieder neue Zutaten auf den Tisch legen soll (d.h. der Agent wartet mittels **`wait`** an diesem Semaphor). Der Semaphor agent ist initial mit 1 belegt, da der Agent am Anfang sofort Zutaten auf den Tisch legen soll. Das Programm des Agenten besteht darin, zunächst in Zeile (1) zwei Zahlen aus der

Menge $\{1, 2, 3\}$ auszuwählen. Diese beide Zahlen geben später an, welche zwei der drei Zutaten vom Agent auf den Tisch gelegt werden. Zunächst wartet der Agent jedoch in Zeile (2), bis er das Signal erhält, Zutaten auf den Tisch zu legen. In Zeilen (3) und (4) legt er die Zutaten auf den Tisch, indem er für die beiden entsprechenden Semaphore eine **`signal`**-Operation ausführt.

Eine Lösung des Cigarette Smoker's Problems besteht nun in der Angabe des Codes für die Raucher, wobei kein (globaler) Deadlock auftreten soll: Der passende Raucher nimmt die Zutaten und raucht.

Initial: wait: binärer Semaphor mit 1 belegt
S[1], S[2] und S[3]: binäre Semaphore für Tabak, Papier und Streichholz mit 0 belegt

```
loop forever
 (1)  wähle i und j zufällig aus {1, 2, 3};
 (2)  wait(agent);
 (3)  signal(S[i]);
 (4)  signal(S[j]);
end loop
```

Abbildung 3.13: Programm des Agenten

Die Schwierigkeit des Problems ist offensichtlich: Nimmt ein Raucher eine passende Zutat, aber die andere Zutat auf dem Tisch passt nicht, so ist ein Deadlock mehr oder weniger unvermeidbar. In Abbildung 3.14 ist ein Code für die Raucher angegeben, der naiv vorgeht, aber leider falsch ist.

Raucher mit Tabak:
```
loop forever
 (1)  wait(S[2]);
 (2)  wait(S[3]);
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Raucher mit Papier:
```
loop forever
 (1)  wait(S[1]);
 (2)  wait(S[3]);
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Raucher mit Streichholz:
```
loop forever
 (1)  wait(S[1]);
 (2)  wait(S[2]);
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Abbildung 3.14: Naiver Versuch zum Cigarette Smoker's Problem

Hierfür betrachte die folgende Ausführung (unterstrichene Befehle werden als nächstes durchgeführt):

Agent	Raucher m. Tabak	Raucher m. Papier	Raucher m. Streichholz	S[1]	S[2]	S[3]	agent
wählt i = 1, j = 2	**wait**(S[2])	**wait**(S[1])	**wait**(S[1])	0	0	0	1
wait(agent)	**wait**(S[2])	**wait**(S[1])	**wait**(S[1])	0	0	0	1
signal(S[1])	**wait**(S[2])	**wait**(S[1])	**wait**(S[1])	0	0	0	0
signal(S[2])	**wait**(S[2])	**wait**(S[1])	**wait**(S[1])	1	0	0	0
wait(agent)	**wait**(S[2])	**wait**(S[1])	**wait**(S[1])	1	1	0	0
wait(agent)	**wait**(S[3])	**wait**(S[1])	**wait**(S[1])	1	0	0	0
wait(agent)	**wait**(S[3])	**wait**(S[3])	**wait**(S[1])	0	0	0	0
blockiert	**wait**(S[3])	**wait**(S[3])	**wait**(S[1])	0	0	0	0
blockiert	blockiert	**wait**(S[3])	**wait**(S[1])	0	0	0	0
blockiert	blockiert	blockiert	**wait**(S[1])	0	0	0	0
blockiert	blockiert	blockiert	blockiert	0	0	0	0

Die Ausführung zeigt, dass ein Deadlock möglich ist: Alle Prozesse sind blockiert. Tatsächlich sind Lösungen für das Cigarette Smoker's Problem nicht einfach. Eine Möglichkeit besteht darin, zu den bestehenden vier Prozessen noch 3 weitere Prozesse hinzu zu fügen, die als Helfer verwendet werden. Zu den bestehenden Semaphore werden noch sechs weitere binäre Semaphore R[1], ..., R[6] hinzugefügt (initialisiert mit 0), sowie ein atomares Register t (initial mit Wert 0 belegt) und ein binärer Semaphor mutex (initial 1), der zum Schutz des Registers t verwendet wird. Abbildung 3.15 zeigt die neue Implementierung: Sie beinhaltet neben den Programmen für die drei Raucher, auch drei Programme für die drei Helfer.

Die Lösung lässt sich wie folgt erläutern: Die Raucher greifen nicht mehr selbst direkt auf den Tisch zu. Sie werden durch die Helfer entblockiert. Für jede der Zutaten gibt es einen Helfer. Jeder Helfer wartet auf seine entsprechende Ressource. Sobald er die Zutat in Form des Semaphors hält, modifiziert er den Wert des atomaren Registers t, wobei er dies atomar durchführt: Hierfür belegt er den Semaphor mutex und stellt damit sicher, dass nur er den Wert von t ändern kann. Hierbei findet noch eine kleine Zahlenspielerei statt: Der Helfer für den Tabak erhöht t um 1, der Helfer für das Papier erhöht t um 2 und der Helfer für das Streichholz erhöht t um 4. Der Anfangswert von t ist 0. Anschließend prüft der Helfer in Zeile (4), ob er den Wert von t *als Erster* erhöht. Ist dies der Fall, so gibt der Helfer den mutex frei und tut ansonsten nichts. Ist der Helfer der zweite der t erhöht hat, dann ist die andere Zutat bereits vom Tisch. In diesem Fall (Zeile (5)) weckt er den entsprechenden Raucher über den Semaphor R[t]. Der Trick besteht also darin, dass derjenige Helfer, der die zweite Zutat erhält (und damit t als zweites erhöht), dafür zuständig ist, den entsprechenden Raucher zu wecken. Hier kommt die Zahlenspielerei hinzu: Mit einer Tabelle kann man nachrechnen, dass das „Richtige" geschieht:

Zusätzliche Objekte: R[i], i = 1,...6, binäre Semaphore (initial 0),
mutex: binärer Semaphor (initial 1),
t: atomares Register (initial 0)

Helfer (Tabak)

```
loop forever
 (1)  wait(S[1]);
 (2)  wait(mutex);
 (3)  t := t+1;
 (4)  if t ≠ 1 then
 (5)   signal(R[t]);
 (4)  signal(mutex);
end loop
```

Helfer (Papier)

```
loop forever
 (1)  wait(S[2]);
 (2)  wait(mutex);
 (3)  t := t+2;
 (4)  if t ≠ 2 then
 (5)   signal(R[t]);
 (4)  signal(mutex);
end loop
```

Helfer (Streichholz)

```
loop forever
 (1)  wait(S[3]);
 (2)  wait(mutex);
 (3)  t := t+4;
 (4)  if t ≠ 4 then
 (5)   signal(R[t]);
 (4)  signal(mutex);
end loop
```

Raucher mit Tabak

```
loop forever
 (1)  wait(R[6]);
 (2)  t := 0;
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Raucher mit Papier

```
loop forever
 (1)  wait(R[5]);
 (2)  t := 0;
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Raucher mit Streichholz

```
loop forever
 (1)  wait(R[3]);
 (2)  t := 0;
 (3)  „rauche“;
 (4)  signal(agent);
end loop
```

Abbildung 3.15: Lösung des Cigarette Smoker's Problem mit Helfern

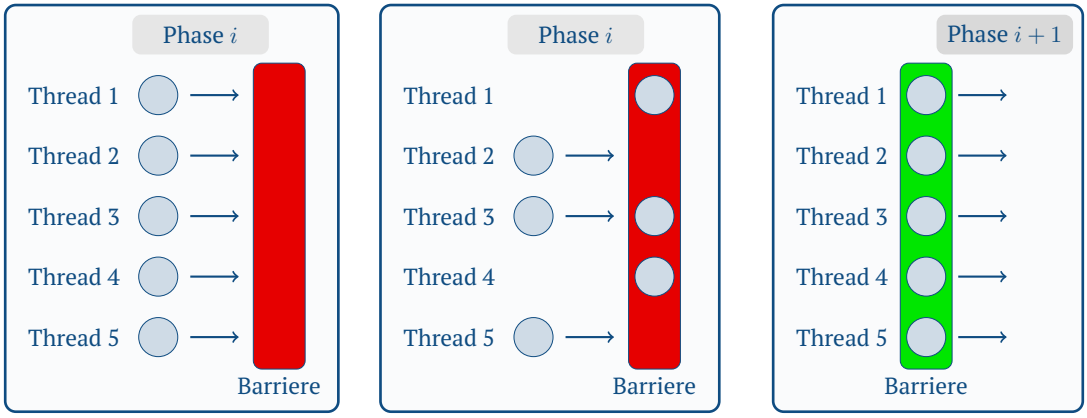
Zutaten auf dem Tisch	erster Helfer	zweiter (=weckender) Helfer	Wert von t	geweckter Raucher
Tabak & Papier	(Tabak)	(Papier)	1+2 = 3	R[3] (=Streichh.)
Tabak & Papier	(Papier)	(Tabak)	2+1 = 3	R[3] (=Streichh.)
Tabak & Streichh.	(Tabak)	(Streichh.)	1+4 = 5	R[5] (=Papier)
Tabak & Streichh.	(Streichh.)	(Tabak)	4+1 = 5	R[5] (=Papier)
Papier & Streichh.	(Papier)	(Streichh.)	2+4 = 6	R[6] (=Tabak)
Papier & Streichh.	(Streichh.)	(Papier)	4+2 = 6	R[6] (=Tabak)

Der Algorithmus für die Raucher ist einfach: Ein Raucher wartet bis er vom Helfer geweckt wird. Anschließend setzt er den Wert von t für die nächste Runde zurück und raucht. Wenn er das Rauchen beendet hat, signalisiert er dem Agenten, dass die nächste Runde beginnen kann.

3.4.6 Barrieren

Als weiteres Beispiel wird die Konstruktion von so genannten „Barriers“ also Barrieren betrachtet. Diese wurden schon bei der Implementierung des parallelisierten Mergesorts verwendet. Barrieren dienen dazu, eine Menge von Prozessen an einem bestimmten Punkt zu synchronisieren, d.h. es wird gewartet bis alle (oder eine bestimmte Menge von) Prozesse die Barriere überschritten haben, bevor fortgesetzt wird. Im Mergesort hat der Merge-Prozess gewartet, bis die beiden Sortierprozesse für die Teilhälften ihr Sortieren beendet haben. Als Barriere dienten die beiden Semaphore left und right. Im Folgenden wird aber davon ausgegangen, dass sämtliche Prozesse aufeinander Warten (diese Situation lag beim Mergesort nicht vor, da beispielsweise der Sortierer der linken Hälfte nicht auf den Sortierer der rechten Hälfte gewartet hat).

Das Problem der Barrieren-Koordination lässt sich auch bildlich darstellen.



Die Prozesse laufen bis zu einem Punkt ihrer Berechnung. Nachdem Sie diesen Punkt erreicht haben, muss gewartet werden, bis sämtliche Prozesse an der Barriere angelangt sind. Erst dann dürfen die Prozesse ihre Berechnung fortsetzen.

Für zwei Prozesse lässt sich das Barrieren-Problem mittels zweier binärer Semaphore relativ leicht lösen. Der Code für beide Prozesse ist in Abbildung 3.16 dargestellt. Sobald ein Prozess mit seiner Berechnung vor der Barriere fertig ist, signalisiert er dies auf „seinem“ Semaphor. Anschließend wartet er an dem „anderen“ Semaphor darauf, dass der andere Prozess fertig wird und dies signalisiert.

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

Programm für Prozess 1:

(1) Berechnung vor der Barriere;
(2) **signal**(p1ready);
(3) **wait**(p2ready);
(4) Berechnung nach der Barriere;

Programm für Prozess 2:

(1) Berechnung vor der Barriere;
(2) **signal**(p2ready);
(3) **wait**(p1ready);
(4) Berechnung nach der Barriere;

Abbildung 3.16: Barriere mit Semaphore für zwei Prozesse

Für n Prozesse kann man mit zwei binären Semaphore und einem atomaren Register (zum Zählen der Prozesse) auskommen. Abbildung 3.17 zeigt die Implementierung.

Sobald Prozesse an der Barriere eintreffen, erhöhen sie den Zähler counter der wartenden Prozesse und (mit der Ausnahme des $n.$ Prozesses) warten anschließend am Semaphor verlassen in Zeile (7). Für den $n.$ Prozess wird die **if**-Bedingung in Zeile (4) falsch. Deshalb gibt er den Semaphor verlassen frei (ein Prozess wird entblockiert). Beim Verlassen der Barriere (ab Zeile (8)) erniedrigen die Prozesse nacheinander den Zähler counter. Der letzte Prozess, der die Barriere verlässt, führt die **signal**(ankommen) Operation in Zeile (11) aus. Damit ist die Barriere wieder in seinen Ursprungszustand zurück versetzt und die nächste „Phase“ kann beginnen. Wichtig hierbei ist, dass zwischen dem Betreten des $n.$ Prozesses und dem Verlassen des letzten Prozesses keine weiteren Prozesse wieder in die Barriere eintreten können. Dies wird allerdings dadurch sichergestellt, dass der $n.$ Prozess am Anfang eine **wait**(ankommen)-Operation (in Zeile (2)) durchführt (alle nächsten Prozesse werden hier blockiert) und der ankommen-Semaphor erst vom aller letzten Prozess, der die Barriere verlässt, freigegeben wird (in Zeile (11)).

Im folgenden werden Barrieren auch als abstrakter Datentyp (z.B. als 4-Tupel (n,ankommen,verlassen,counter)) verwendet, welcher die folgenden Operationen unterstützt:

- `newBarrier`(k): Erzeugt eine Barriere für k Prozesse, d.h. es werden zwei Semaphore für ankommen und verlassen und ein atomares Register counter erzeugt und entsprechend initialisiert und ein 4-Tupel (n,ankommen,verlassen,counter) wird zurück gegeben.

- `synchBarrier`(B): Synchronisieren an der Barriere, d.h. gerade die Zeilen (2) bis (12) aus Abbildung 3.17 werden ausgeführt. für das 4-Tupel B.

Initial: ankommen: binärer Semaphor mit 1 initialisiert
verlassen: binärer Semaphor mit n initialisiert
counter: atomares Register mit 0 initialisiert

Programm für Prozess i:

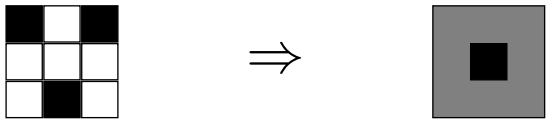
```
(1)   Berechnung vor der Barriere;
(2)   wait(ankommen);
(3)   counter := counter + 1;
(4)   if counter < n then
(5)     signal(ankommen)
(6)   else signal(verlassen)
(7)   wait(verlassen);
(8)   counter := counter - 1;
(9)   if counter > 0 then
(10)    signal(verlassen)
(11)  else signal(ankommen)
(12)  Berechnung nach der Barriere;
```

Abbildung 3.17: Barriere mit Semaphore für n Prozesse

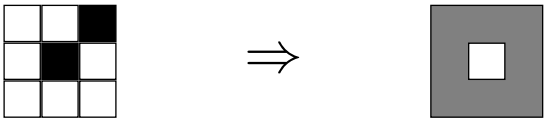
3.4.6.1 Eine Beispielanwendung: Conways Game of Life

Das Spielfeld von Conways Game of Life besteht aus einer zweidimensionalen $N \times N$-Matrix, wobei Felder entweder *bewohnt* oder *unbewohnt* sind. Im Game of Life wird diese Matrix von Generation zu Generation verändert. Für ein Feld wird der Wert in der nächsten Generation dabei in Abhängigkeit der Anzahl der Nachbarn ermittelt:

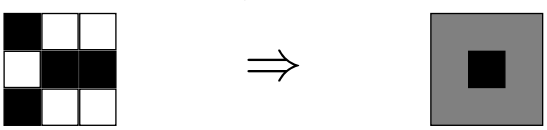
- Ist das Feld unbewohnt, so ist es nur dann in der nächsten Generation bewohnt, wenn es genau drei Nachbarn hatte. Zum Beispiel:



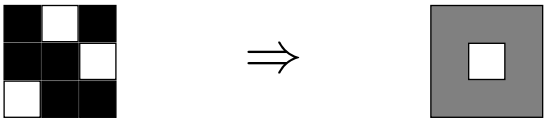
- Ist das Feld bewohnt, so gilt für die nächste Generation:
  - Hatte es keinen oder einen Nachbarn, so ist es anschließend aufgrund von Unterpopulation unbewohnt. Z.B.



  - Hatte das Feld zwei oder drei Nachbarn, so bleibt es bewohnt. Z.B.



  - Hatte das Feld vier oder mehr Nachbarn, so ist es anschließend aufgrund von Überpopulation unbewohnt.



Das Spielfeld sei durch ein zweidimensionales Array mit Booleschen Einträgen dargestellt. Außerdem sei Funktion **`naechsterWert`**$(i, j, array)$ bereits implementiert, die für das Feld in Zeile i und Spalte j berechnet, ob dieses in der nächsten Generation unbewohnt oder bewohnt ist: Die Funktion liest nacheinander alle Werte der Nachbarfelder des Feldes (i, j) sowie den Wert des Feldes (i, j) und berechnet anschließend **`True`** oder **`False`** entsprechend der obigen Regeln.

Ein sequentieller Algorithmus, der die k. Nachfolgegeneration berechnet, ist der folgende:

array: Initialisiertes $N \times N$ Array, dass das Spielfeld darstellt

Algorithmus:

```
for g:=1 to k do
 for i=1 to N do
  for j=1 to N do
   array2[i,j] := naechsterWert(i,j,array);
 for i=1 to N do
  for j=1 to N do
   array[i,j] := array[i,j];
```

Angenommen man möchte diesen Algorithmus nun parallelisieren (bzw. nebenläufig programmieren), so dass pro Feld ein Prozess verfügbar ist, der die Aktualisierung eines Feldes berechnet, so ist ein naiver Ansatz der folgende Algorithmus:

array: Initialisiertes $N \times N$ Array, dass das Spielfeld darstellt
$(N \times N)$ Prozesse: jeweils einen pro Spielfeld

Programm für Prozess (i,j):

```
for g:=1 to k
    v := naechsterWert(i,j,array);
    array[i,j] := v;
```

Leider ist der Algorithmus jedoch falsch, denn wenn Prozess (i, j) den Wert der l. Generation berechnet, ist nicht sichergestellt, dass er im Array wirklich nur die Werte der $l-1$. Generation für die Nachbarn liest: Andere Prozesse können langsamer oder schneller sein und das Feld an ihren Positionen zu früh bzw. zu spät aktualisieren. Die Prozesse müssen sich synchronisieren. Hierfür eignet sich eine Barriere: Dann können die Prozesse wie folgt programmiert werden:

array: Initialisiertes $N \times N$ Array, dass das Spielfeld darstellt
barrier: Barriere für $N \times N$ Prozesse
$(N \times N)$ Prozesse: jeweils einen pro Spielfeld

Programm für Prozess (i,j):

```
for g:=1 to k
    v := naechsterWert(i,j,array);
    synchBarrier(barrier);
    array[i,j] := v;
    synchBarrier(barrier);
```

Jetzt funktioniert die Berechnung: Bevor die Prozesse den neuen Wert in das Feld schreiben dürfen, müssen sie alle an der Barriere aufeinander Warten (erster **synchBarrier**-Aufruf) und anschließend nach dem Schreiben, warten sie erneut alle aufeinander (zweiter **synchBarrier**-Aufruf), damit sichergestellt ist, dass vor dem Berechnen der nächsten Generation wirklich alle ins Feld geschrieben haben.

3.4.7 Das Readers & Writers Problem

Das Readers & Writers Problem ist wie das Barrieren-Problem eher ein generelles Konzept. Hierbei werden die Prozesse in zwei Gruppen aufgeteilt. Eine Gruppe sind die Readers (also lesende Prozesse), während die andere Gruppe die Writers (also schreibende Prozesse) sind. Eine solche Aufteilung kann man z.B. auf Datenbanksysteme bzw. Zugriffe auf Datenbanken anwenden. Es gibt Prozesse, welche nur lesend auf die Daten zugreifen, während andere Prozesse die Einträge in der Datenbank ändern möchten. Der wesentliche Punkt hierbei ist, dass mehrere Prozesse gleichzeitig auf die Datenbank zugreifen können, solange alle nur die Daten lesen. Die schreibenden Prozesse benötigen hingegen exklusiven Zugriff, um Konsistenz der Daten zu gewährleisten. Das Readers & Writers Problem hat somit als Ziel, möglichst parallelen Lesezugriff zu ermöglichen und den Schreibzugriff nur sequentiell zu zulassen.

Man kann für dieses Problem zwei unterschiedliche Strategien verfolgen, je nachdem welche der beiden Gruppen von Prozessen höhere Priorität beim Zugriff auf die Daten haben soll. Gibt man den Lesern eine höhere Priorität als den Schreibern, so müssen die schreibenden Prozesse warten (sie werden blockiert) solange noch lesende Prozesse vorhanden sind bzw. eintreffen. Bei umgekehrter Priorität werden schreibende Prozesse bevorzugt, sie erhalten den Vorrang gegenüber lesenden Prozessen.

Je nach Anwendung kann eine Prioritätswahl vorteilhafter sein als eine andere. Betrachtet man als Beispiel ein Flugreservierungssystem, so ist es aus Sicht des Betreibers wohl günstiger den Schreibern den Vorrang zu geben, gegenüber solchen Prozessen, die nur nach möglichen Flügen suchen, aber (noch) nicht Flüge buchen wollen: Zum Einen verdient der Betreiber mit gebuchten Flügen mehr Geld, zum Anderen wollen die lesenden Anfragen möglichst schnell über aktuelle Flüge (also nicht ausgebuchte Flüge) informiert werden.

Im Folgenden werden Lösungen mit Semaphore für beide Prioritätsvarianten erläutert.

Abbildung 3.18 enthält den Code für Reader- und Writer-Prozesse, wobei Reader-Prozesse höhere Priorität haben als Writer-Prozesse.

Initial: countR: atomares Register, am Anfang 0
mutex, mutexR, w: binäre Semaphore, am Anfang 1

Programm für Reader:

```
(1)   wait(mutexR);
(2)   countR := countR + 1;
(3)   if countR = 1 then
(4)     wait(w);
(5)   signal(mutexR);
(6)   Kritischer Abschnitt
(7)   wait(mutexR);
(8)   countR := countR - 1;
(9)   if countR = 0 then
(10)    signal(w);
(11)  signal(mutexR);
```

Programm für Writer:

```
(1)  wait(mutex);
(2)  wait(w);
(3)  signal(mutex);
(4)  Kritischer Abschnitt;
(5)  signal(w)
```

Abbildung 3.18: Readers & Writers mit Priorität für Readers

Die Implementierung benutzt vier Datenstrukturen: Ein atomares Register countR, welches die Anzahl der momentanen Reader-Prozesse speichert. Folglich hat countR als anfänglichen Wert den Wert 0. Der Semaphor mutexR wird benutzt, um den Zugriff auf countR (und auf den Semaphor w) zu schützen, wobei der initiale Wert 1 ist. Des Weiteren wird der Semaphor w benutzt, um den Zugriff von schreibenden Prozessen zu verhindern, falls lesende Prozesse vorhanden sind, oder schon ein anderer schreibender Prozess aktiv ist. Der Semaphor w ist mit 1 initialisiert. Der Semaphor mutex dient dazu, dass höchstens ein Schreiber an w warten kann.

Das Programm für die schreibenden Prozesse ist einfach: In Zeile (2) prüft der Prozess, ob der Semaphor w belegt ist. Ist dies der Fall, wird der Prozess blockiert und wartet auf die Freigabe. Andernfalls darf der schreibende Prozess den kritischen Abschnitt betreten (Zeile (4)). Nach Verlassen des kritischen Abschnitts signalisiert dies der schreibende Prozess durch eine **`signal`**-Operation auf dem Semaphor w. Wie später gezeigt wird , kann der Semaphor w entweder durch einen lesenden Prozess oder durch schreibende Prozesse belegt werden. Schon jetzt ist offensichtlich, dass stets höchstens ein schreibender Prozess den kritischen Abschnitt betreten kann. Das Belegen des Semaphors mutex in Zeile (1) und Freigeben in Zeile (3) dient dazu, dass höchstens ein schreibender Prozess am Semaphor w wartet. Dadurch erhalten Leser den Vorzug vor Schreibern.

Der Code eines lesenden Prozesses lässt sich wie folgt kommentieren: Zunächst belegt dieser den Semaphor mutexR, um exklusiv auf das Register countR zugreifen zu können. In Zeile (2) wird die gespeicherte Anzahl der lesenden Prozesse um eins erhöht. In den Zeile (3) und (4) erhält der erste lesende Prozess eine Sonderrolle (er hatte countR zuvor von 0 auf 1 erhöht): Dieser erste Prozess belegt den Semaphor w. Dadurch wartet er entweder darauf, dass alle schreibenden Prozesse den kritischen Abschnitt verlassen und anschließend (oder gleich) sorgt er durch das Belegen von w dafür, dass keine weiteren schreibenden Prozesse in den kritischen Abschnitt eintreten können, sondern an w blockiert werden. An dieser Stelle können andere lesende Prozesse nicht stören: Der erste lesende Prozess hat zu dem Zeitpunkt als er in Zeile (4) ist, den Semaphor mutexR immer noch belegt, dadurch können andere lesende Prozesse nicht über Zeile (1) hinweg kommen.

In Zeile (5) gibt der lesende Prozess den Semaphor mutexR frei. Dadurch können weitere Leser in den kritischen Abschnitt eintreten. Nach dem Verlassen des kritischen Abschnitts benutzt ein Reader-Prozess erneut den Semaphor mutexR, um den Wert von countR exklusiv zu ändern. In Zeile (8) erniedrigt er den Wert um 1, die Zeilen (9) und (10) spielen wiederum eine Sonderrolle: Sie sind nur relevant für den letzten Leser. Dieser sorgt dafür, dass ein eventuell wartender Writer-Prozess entblockiert wird (mittels der **`signal`**(w)-Operation in Zeile (10)).

Dieser Algorithmus gibt den Reader-Prozessen dadurch Priorität vor den Writer-Prozessen, da solange lesende Prozesse im kritischen Abschnitt sind, stets weitere lesende Prozesse ebenfalls in den kritischen Abschnitt eintreten können. Wartende Writer-Prozesse müssen abwarten, bis der letzte Leser den Semaphor w entblockiert.

In Abbildung 3.19 ist eine weitere Implementierung für das Readers & Writers-Problem dargestellt, welche Writer-Prozessen den Vorzug vor Reader-Prozessen gibt.

Gegenüber dem vorherigen Algorithmus wird nun zusätzlich ein atomares Register countW benutzt, welches die Anzahl der wartenden Writer-Prozesse zählt und dessen Zugriff durch den Semaphor mutexW geschützt wird. Der Zähler countR für die Anzahl der Reader-Prozesse wird wie zuvor durch den Semaphor mutexR geschützt. Wie vorher wird der Semaphor w benutzt, um Writer-Prozesse am Eintritt in den kritischen Abschnitt zu hindern, wenn bereits ein weiterer

Initial: countR, countW: atomare Register, am Anfang 0
mutexR, mutexW, mutex, w, r: binäre Semaphore am Anfang 1

Programm für Reader:

```
(1)   wait(mutex);
(2)   wait(r);
(3)   wait(mutexR);
(4)   countR := countR + 1;
(5)   if countR = 1 then wait(w);
(6)   signal(mutexR);
(7)   signal(r);
(8)   signal(mutex);
(9)   Kritischer Abschnitt;
(10)  wait(mutexR);
(11)  countR := countR - 1;
(12)  if countR = 0 then signal(w);
(13)  signal(mutexR);
```

Programm für Writer:

```
(1)   wait(mutexW);
(2)   countW := countW + 1;
(3)   if countW = 1 then wait(r);
(4)   signal(mutexW);
(5)   wait(w);
(6)   Kritischer Abschnitt;
(7)   signal(w)
(8)   wait(mutexW);
(9)   countW := countW - 1;
(10)  if countW = 0 then signal(r);
(11)  signal(mutexW);
```

Abbildung 3.19: Readers & Writers mit Priorität für Writers

schreibender oder auch lesende Prozesse im kritischen Abschnitt sind. Neu ist der Semaphor r, der dazu dient, lesende Prozesse zu blockieren. Der Semaphor mutex erhält nun eine andere Aufgabe als zuvor. Diese wird später erläutert.

Das Programm für Writer-Prozesse lässt sich wie folgt erläutern: Zeilen (5) bis (7) sind genau wie vorher: Nur wenn der schreibende Prozess den Semaphor w hält, darf er in den kritischen Abschnitt. In den Zeile (1) bis (4) wird der Zähler countW (die Anzahl der schreibenden Prozesse, die in den kritischen Abschnitt wollen) exklusiv durch Schutz mittels des Semaphors mutexW erhöht. Der erste schreibende Prozess erhält in der Zeile (3) eine Sonderrolle: Er wartet am Semaphor r zum Einen darauf, dass keine Leser im kritischen Abschnitt sind und zum Anderen (sobald er nicht mehr blockiert ist) sorgt er dafür, dass kein schreibender Prozess mehr in den kritischen Abschnitt gelangt, solange noch schreibende Prozesse vorhanden sind. In Zeilen (8)-(10) wird die Zahl der schreibenden Prozesse (exklusiv durch Schutz mit mutexW) erniedrigt. Der letzte schreibende Prozess gibt den Zugriff für lesende Prozesse mittels der **`signal`**(r)-Operation frei.

Für die Betrachtung des Reader-Programms kann man zunächst feststellen, dass bis auf Zeilen (1), (2), (7) und (8) die Funktionalität wie vorher ist. Die **`wait`**(r)- und **`signal`**(r)-Aufrufe dienen dazu, lesende Prozesse zu blockieren, falls schreibende Prozesse vorhanden sind. Die beiden darum geschachtelten Aufrufe für den Semaphor mutex dienen dazu, dass höchstens ein lesender Prozess am (blockierten) Semaphor r wartet. Dadurch erhalten schreibende Prozesse den Vorzug gegenüber lesenden Prozessen: Wenn der erste Writer-Prozess in seiner Zeile (3) die **`wait`**(r)-Operation durchführt, kann höchstens noch ein Leser bereits an r blockiert warten. Würden die Operationen für mutex weggelassen, könnten dort beliebig viele lesende Prozesse warten und mit dem ersten schreibenden Prozess konkurrieren. Des Weiteren lässt sich feststellen, dass solange schreibende Prozesse vorhanden sind, diese stets Vorzug vor den lesenden haben, da der Semaphor r erst freigegeben wird, wenn der letzte schreibende Prozess den kritischen Abschnitt verlässt.

3.5 Monitore

In diesem Abschnitt wird eine weitere Programmierabstraktion – so genannte *Monitore* – erläutert. Obwohl sich mit Semaphore viele Probleme der nebenläufigen Programmierung relativ elegant und einfach lösen lassen, haben Semaphore einige Nachteile. Zum Beispiel widersprechen sie dem Prinzip der strukturierten Programmierung: Sobald ein **`signal`**-Aufruf am Ende fehlt, kann ein (globaler) Deadlock des Programms auftreten, oder auch umgekehrt: fehlt ein **`wait`**-Aufruf vor dem Zugriff auf kritische Abschnitte, so können Race Conditions auftreten. Das Problem hierbei ist, dass es die Aufgabe des Programmierers ist, dafür zu sorgen, dass immer die entsprechenden **`wait`**- und **`signal`**-Operationen vorhanden sind. Deswegen wäre es vorteilhafter, wenn die Programmstruktur (Syntax) den Programmierer dazu zwingt, die entsprechenden Operationen richtig zu benutzen. Monitore setzen an diesem Punkt an. Sie verfolgen dabei das Paradigma der strukturierten Programmierung, wie es auch bei der Objektorientierten Programmierung zu finden ist: ein Monitor besteht aus Daten (Attribute) und Methoden, welche die Werte der Attribute lesen und verändern dürfen.

In Abbildung 3.20 ist eine Monitor-Definition für ein Bankkonto angegeben. Mit dem Schlüsselwort **`monitor`** wird eine Monitor-Definition eingeleitet. Anschließend werden die Attribute, danach die Methoden definiert.

```
monitor Konto {
  int Saldo;
  int Kontonummer;
  int KundenId

  abheben(int x) {
    Saldo := Saldo - x;
  }

  zubuchen(int x) {
    Saldo := Saldo + x;
  }
}
```

Abbildung 3.20: Monitor für ein Bankkonto

Der Konto-Monitor verfügt über Attribute für den Kontostand, die Kontonummer und eine Kundenidentifikationsnummer. Es existieren zwei Methoden: Eine, um einen Betrag dem Konto gut zu schreiben, und eine weitere Methode zum Abbuchen eines Betrages.

Wird eine Monitor-Methode aufgerufen (z.B. Konto.abheben(10)), so wird implizit sichergestellt, dass stets nur eine Methode zur gleichen Zeit atomar ausgeführt wird. Hierfür gibt es einen Monitor-Lock, den nur ein Prozess zu einer Zeit belegen kann. Beim Aufruf der Monitor-Methode versucht der Prozess den Monitor-Lock zu belegen. Ist dieser frei, so wird er belegt, die Methode ausgeführt und anschließend der Lock wieder freigegeben. Ist der Lock belegt, so reiht sich der Prozess in die Menge der wartenden Prozesse ein (und wird blockiert). Beim Freigeben des Locks wird einer der wartenden Prozesse entblockiert. Je nach Implementierung kann die Menge der wartenden Prozesse als FIFO-Queue oder auch als einfache Menge implementiert sein.

Die Grafik in Abbildung 3.21 stellt einen Monitor symbolisch dar. Am Eingang gibt es eine Queue von wartenden Prozessen und nur ein Prozess hat Zugriff auf die Monitorvariablen. Hierbei ist zu erwähnen, dass der Zugriff auf die Monitor-Attribute ausschließlich über Monitor-Methoden erlaubt ist (d.h. der direkte Zugriff ist verboten).

Allgemein lässt sich der wechselseitige Ausschluss mithilfe eines Monitors implementieren, indem alle Variablen des kritischen Abschnitts durch den Monitor überwacht werden, d.h. als Attribute des Monitors definiert werden. Abbildung 3.22 zeigt eine solche Implementierung: Der Monitor Mutex enthält die Variablen des kritischen Abschnitts und die Methode inKritischenAbschnitt (diese kann evtl. Parameter erhalten, um entsprechende Werte zu übergeben). Durch den Monitor ist sichergestellt, dass nur ein Prozess gleichzeitig im kritischen Abschnitt ist, da nur ein Methodenaufruf zu gleicher Zeit ausgeführt wird. Die Implementierung ist auch Deadlock-frei, da stets einer der nebenläufigen Aufrufe durchgeführt wird. Starvation-Freiheit ist nur garantiert, wenn der Monitor-Lock durch eine FIFO-Queue verwaltet wird, was nicht immer der Fall ist.

Die bisher definierten Monitore beheben zwar das Problem der Semaphore, da kein explizites **`signal`** und **`wait`** notwendig ist, um wechselseitigen Ausschluss zu Programmieren, al-

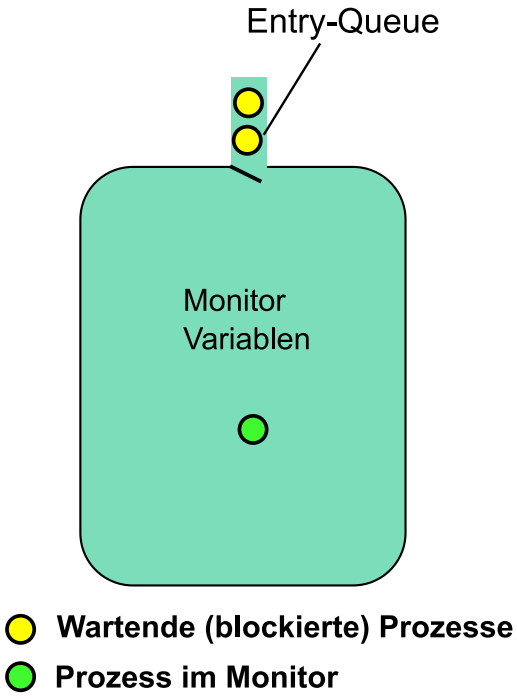


Abbildung 3.21: Schematische Darstellung eines einfachen Monitors

```
monitor Mutex {
   ...
   Variablen des kritischen Abschnitts
   ...
   inKritischenAbschnitt(...) {
      Kritischer Abschnitt;
   }
}

Prozess i:
 loop forever
 (1)   Restlicher Code
 (2)   Mutex.inKritischenAbschnitt()
 end loop
```

Abbildung 3.22: Mutual-Exclusion mit einem Monitor

lerdings fehlen noch einige wichtige Funktionalitäten, die Semaphore zur Verfügung stellen. Viele Lösungen zu den Problemen der nebenläufigen Programmierung benötigen das explizite Blockieren von Prozessen, bis eine Bedingung erfüllt ist. Ein Beispiel hierfür ist zum Beispiel der bounded Buffer für das Erzeuger/Verbraucher-Problem: Der Puffer muss so programmiert sein, dass ein Erzeuger blockiert wird, falls der Puffer voll ist und analog muss ein Verbraucher blockiert werden, falls der Puffer leer ist. Die bisherige Monitor-Definition erlaubt hierfür keine Implementierung. Deshalb stellen Monitore weitere Konstrukte zur Verfügung, um das explizite Blockieren und Entblockieren von Prozessen zu bewerkstelligen. Man unterscheidet hierbei zwei verschiedene Arten von Monitoren: Zum einen gibt es Monitore, die so genannte *Condition Variablen* zur Verfügung stellen, zum anderen gibt es Monitore mit so genannten *Condition Expressions*. Hier wird sich hauptsächlich mit den gebräuchlicheren Condition Variablen beschäftigt und später kurz auf Condition Expressions eingegangen.

3.5.1 Monitore mit Condition Variablen

Eine *Condition Variable* ist eine FIFO-Queue auf die im Monitor mithilfe der vordefinierten Operationen **`waitC`**, **`signalC`** und **`emtpy`** zugegriffen werden kann. Eine Condition Variable wird im Attribut-Definitionsblock mit dem Schlüsselwort **`condition`** definiert. Nach dem Schlüsselwort folgt der Name der zu definierenden Condition Variable. Dieser Name wird häufig so gewählt, dass er der Bedingung entspricht, auf deren Erfüllung gewartet wird. Hierbei ist zu beachten, dass es sich trotz allem nur um einen Variablennamen handelt, d.h. es gibt keine logische Verbindung zwischen dem Namen und der Bedingung.

Im Folgenden wird die Semantik der drei verfügbaren Operationen für Condition Variablen beschrieben. Schon jetzt sei erwähnt, dass Unterschiede zwischen den von Semaphore bekannten Operationen **`wait`** und **`signal`** und den Operationen **`signalC`** und **`waitC`** für Condition Variablen bestehen.

Die **`empty`**-Operation prüft, ob die Queue zur Condition Variable leer ist, d.h. ob es wartende Prozesse gibt. In Pseudo-Code lässt sich dies für die Condition Variable cond schreiben als:

```
empty(cond) {
   return(cond = empty);
}
```

Die **`waitC`**-Operation führt dazu, dass der aufrufende Prozess blockiert wird und sich in die Warteschlange der Condition Variable hinzufügt. Da die Operation innerhalb eines Monitors ausgeführt wird, hat der aufrufende Prozess zum Zeitpunkt des Aufrufs den Lock des Monitors belegt. Da der Prozess nun blockiert wird, gibt er durch die **`waitC`**-Operation den Lock frei.

Sei P der aufrufende Prozess, cond eine Condition Variable im Monitor monitor und sei monitor.lock der implizite Lock des Monitors (der Lock zur Entry-Queue). Die Semantik von **`waitQ`** lässt sich durch den folgenden Pseudo-Code beschreiben (der atomar ausgeführt wird):

```
waitC(cond) {
   cond := append(cond,P);    //Prozess wird zur Queue hinzugefügt
   P.state := blocked;        //Prozess blockiert
   monitor.lock := release;   //Monitor-Lock wird freigegeben
}
```

Die **signalC**-Operation entblockiert den ersten Prozess in der Warteschlange. Sind keine Prozesse vorhanden, so ist die **signalC**-Operation wirkungslos. In Pseudo-Code lässt sich dies schreiben als:

```
signalC(cond) {
  if cond ≠ empty then
    Q := head(cond);        //Erster Prozess der Queue
    cond := tail(cond);     //wird aus Queue entfernt
    Q.state := ready;       //und entblockiert
}
```

Allerdings entsteht hierbei ein Problem. Der Monitor sichert zu, dass stets nur ein Prozess in einer Monitor-Methode Berechnungen ausführen darf. Mit der bis hierhin spezifizierten Semantik, können aber beide Prozesse – der **signalC** aufrufende Prozess und der entblockierte Prozess – weiter rechnen. Das Problem wird durch die Annahme gelöst, dass der **signalC** aufrufende Prozess den Lock an den entblockierten Prozess abgibt. D.h. der **signalC** aufrufende Prozess wird blockiert, und muss erneut den Lock des Monitors erlangen. Für die meisten Algorithmen spielt dies keine so große Rolle, da **signalC** die letzte Operation einer Methode ist. Später wird jedoch genauer auf das Problem und verschiedene Lösungen dafür eingegangen.

Als Beispiel für die Verwendung von Monitoren wird die Implementierung eines Semaphors mithilfe von Monitoren betrachtet. Der Code ist in Abbildung 3.23 dargestellt.

```
monitor Semaphore {
  int s := k;
  condition notZero;

  wait() {
    if s = 0 then
      waitC(notZero)
    s := s - 1
  }

  signal() {
    s := s + 1
    signalC(notZero)
  }
}
```

Abbildung 3.23: Semaphore-Simulation mit einem Monitor

Die $S.V$-Komponente des Semaphors wird durch die Monitor-Variable s repräsentiert. Die Condition Variable notZero wird benutzt, um darauf zu warten, dass s einen Wert ungleich 0 erhält. Die implementierte wait-Operation testet, ob s den Wert 0 hat. Ist dies der Fall, so wartet der Prozess an der Condition Variable notZero. Wenn s einen Wert größer 0 hat *und* nach dem Entblockieren eines vorher blockierten Prozess wird der Wert s um 1 verringert. Das Erniedrigen ohne vorheriges Blockieren entspricht dem Verhalten eines Semaphors, der andere

Fall sieht zunächst falsch aus. Die Korrektheit sieht man jedoch beim Erläutern der signal-Implementierung: Hier wird zunächst stets der Wert von s erhöht und anschließend ein blockierter Prozess entblockiert. Beachte, dass **`signalC`** wirkungslos ist, wenn es keine blockierten Prozesse gibt. Da signal den Wert von s auch dann erhöht, wenn ein Prozess entblockiert wird, und der entblockierte Prozess (innerhalb von wait()) den Wert von s wieder erniedrigt, ergibt sich insgesamt, dass der Wert von s in diesem Fall gleich bleibt. Dadurch ist die Implementierung des Semaphors korrekt.

Vergleicht man die **`wait`**- und **`signal`**-Operationen von klassischen Semaphore und die **`waitC`**- und **`signalC`**-Operationen von Monitoren, so lassen sich die Unterschiede in der folgenden Tabelle festhalten:

Semaphore Sem	Monitore (Condition Variable cond)
`wait`(Sem) kann zum Blockieren führen, muss aber nicht (wenn $S.V > 0$ kein Blockieren, sondern Erniedrigen von $S.V$)	**`waitC`**(cond) blockiert den aufrufenden Prozess *stets*
`signal`(Sem) hat stets einen Effekt: Entblockieren eines Prozesses aus $S.M$ oder Erhöhen von Sem.V	**`signalC`**(cond) kann effektlos sein: Entweder wird ein Prozess der Queue cond entblockiert, oder – wenn cond leer ist – ist kein Effekt zu beobachten.

3.5.1.1 Monitorlösung für das Erzeuger / Verbraucher-Problem

In diesem Abschnitt betrachten wir eine Lösung für das Erzeuger/Verbraucher-Problem mithilfe eines Monitors. Der Code für einen Puffer mit begrenztem Platz (bounded buffer) ist in Abbildung 3.24 dargestellt.

Der Monitor hat als überwachte Objekte eine Liste buffer (vom Typ bufferType). Die maximale Größe des Puffers ist direkt als Zahl N kodiert. Es werden zwei Condition Variablen benutzt. An der Condition Variable notFull wird gewartet, falls der Puffer voll ist, und an der Condition Variable notEmpty wird gewartet, falls der Puffer leer ist. Die produce-Methode prüft zunächst, ob die Länge des Puffers gleich zur maximale Größe N ist. Ist dies der Fall, so ist der Puffer voll und der erzeugende Prozess wartet an der Condition Variable notFull darauf, dass der Puffer wieder nicht voll ist. Anschließend hängt der Erzeuger sein erzeugtes Element an die Liste an und ruft zuletzt **`signalC`**(notEmpty) auf, um zu signalisieren, dass der Puffer nicht leer ist. Dadurch wird ein wartender Verbraucher entblockiert (falls mindestens einer vorhanden ist). Die consume-Methode prüft zunächst, ob der Puffer leer ist. Ist dies der Fall, so wird der aufrufende Prozess blockiert und wartet an der Condition Variable notEmpty darauf, dass wieder Elemente in den Puffer eingefügt werden. Anschließend (oder sofort) wird das erste Element aus der Liste entfernt, eine **`signalC`**-Operation für die Condition Variable notFull ausgeführt, um einen wartenden Erzeuger zu wecken und schließlich der erste Wert des Puffers als Ergebnis zurück gegeben.

Die Programme für den Erzeuger und für den Verbraucher sind einfach zu erklären. Sie benutzen die entsprechenden Methoden des Monitors BoundedBuffer, um Elemente dem Puffer hinzu zu fügen bzw. vom Puffer zu entfernen.

```
monitor BoundedBuffer {
  bufferType buffer := empty;
  condition notEmpty;
  condition notFull;

  produce(v) {
    if length(buffer) = N then
      waitC(notFull);
    buffer := append(v,buffer);
    signalC(notEmpty);
  }

  consume() {
    if length(buffer) = 0 then
      waitC(notEmpty)
    w := head(buffer);
    buffer := tail(buffer);
    signalC(notFull)
    return(w);    }
}
```

Code des Erzeugers:

```
loop forever
(1) erzeuge e
(2) BoundedBuffer.produce(e)
end loop
```

Code des Verbrauchers:

```
loop forever
(1) e := BoundedBuffer.consume();
(2) verbrauche e;
end loop
```

Abbildung 3.24: Bounded Buffer als Monitor

3.5.2 Verschiedene Arten von Monitoren

Bevor weitere Monitor-Lösungen zu klassischen Problemen der nebenläufigen Programmierung präsentiert werden, wird das bereits erwähnte Problem nach einer **`signalC`**-Operation betrachtet: Hier ist zum Einen unklar, welcher Prozess den Monitor-Lock nach dem Entblockieren erhält. Zum Anderen ist nicht klar, wo (d.h. an welcher Queue) der Prozess, der den Monitor-Lock *nicht* erhält, warten muss.

Um das Problem genauer zu erläutern, wird das Modell von Monitoren verfeinert. Eine Condition Variable besteht nicht mehr aus nur *einer* Queue, sondern verwendet *drei* Warteschlangen: Die Condition-Queue (es gibt eine pro Condition-Variable), der Wait-Queue und der Signal-Queue (diese gibt es einmal pro Monitor). Diese werden wie folgt verwendet:

Condition-Queue: An der Condition-Queue warten die durch **`waitC`** blockierten Prozesse. Diese Queue ist jene Warteschlange, die wir eigentlich schon kennen und in der vorherigen Monitor-Spezifikation implizit benutzt haben.

Wait-Queue: Wird eine **`signalC`**-Operation ausgeführt und die Condition-Queue ist nicht leer, dann wird der erste wartende Prozess der Condition-Queue in die Wait-Queue eingefügt.

Signal-Queue: Der Prozess der **`signalC`** ausführt, wird in die Signal-Queue eingereiht, *falls* er einen Prozess der Condition-Queue entblockiert hat.

Für jede Condition Variable gibt es eine Condition Queue, und pro Monitor eine Wait-, eine Signal- und eine Entry-Queue. An der Entry-Queue warten jene Prozesse die den Monitor zum ersten Mal betreten wollen.

Abbildung 3.25 veranschaulicht den Monitor mit den drei globalen Queues und den Condition Queues.

Das so entstandene Modell geht nun davon aus, dass nach einer entblockierenden **`signalC`**-Operation zunächst keiner automatisch den Monitor-Lock erhält, sondern alle am Monitor Lock wartenden Prozesse diesen neu belegen können (das sind die Prozesse an der Entry-, der Wait- und der Signal-Queue). Man kann diesen drei unterschiedlichen Warteschlangen Prioritäten zuordnen, die angeben welche Prozesse an welche Queue zuerst den Monitor-Lock belegen dürfen. Die Prioritäten werden wie folgt bezeichnet:

- E für die Entry-Queue
- W für die Wait-Queue
- S für die Signal-Queue

Höhere Priorität bedeutet, dass die entsprechende Queue Vorzug vor der anderen Queue erhält. Vergleicht man die drei Prioritäten, so kann man 13 verschiedene Konstellationen untersuchen. In Abbildung 3.26 ist eine Tabelle mit allen 13 Möglichkeiten dargestellt.

Kombinationen in denen die Entry-Queue eine höhere Priorität hat als die Wait- und die Signal-Queue (Möglichkeiten 7, 12 und 13 in der Tabelle) sind wenig sinnvoll, da Prozesse, die den Monitor bereits betreten hatten, möglicherweise beliebig lange warten müssen, denn neu hinzukommende Prozesse können diese stets „überholen“. Auch die Möglichkeiten 8-11 sind wenig sinnvoll, da in diesen Fällen die Entry-Queue höhere Priorität hat als eine der beiden anderen Queues. Dadurch können erneut „alte“ Prozesse durch „neue“ Prozesse beliebig oft überholt werden.

Die klassische Definition von Monitoren lässt sich durch die 6. Möglichkeit beschreiben, d.h. $E < S < W$. Bei dieser Variante wird nach einer **`signalC`**-Operation sofort der Monitor-Lock

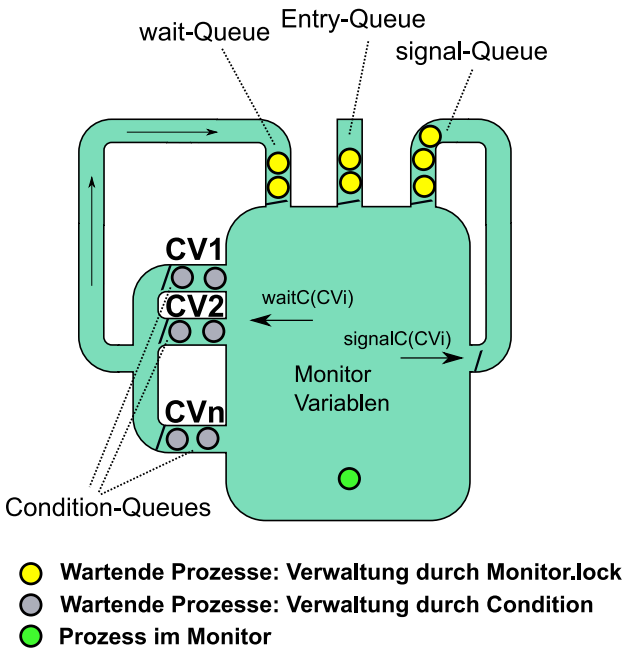


Abbildung 3.25: Illustration eines Monitors mit Wait- und Signal-Queue für die Condition Variablen

1	$E = W = S$	
2	$E = W < S$	Wait and Notify
3	$E = S < W$	Signal and Wait
4	$E < W = S$	
5	$E < W < S$	Signal and Continue
6	$E < S < W$	Signal and Urgent Wait, klassische Definition
7	$E > W = S$	nicht sinnvoll
8	$E = S > W$	nicht sinnvoll
9	$S > E > W$	nicht sinnvoll
10	$E = W > S$	nicht sinnvoll
11	$W > E > S$	nicht sinnvoll
12	$E > S > W$	nicht sinnvoll
13	$E > W > S$	nicht sinnvoll

Abbildung 3.26: 13 Möglichkeiten für die Verteilung der Prioritäten

an den entblockierten Prozess übergeben. Wenn dieser Prozess den Monitor verlässt, darf der signalisierende Prozess weiter rechnen. Neue Prozesse haben die niedrigste Priorität. Monitore mit dieser Strategie werden als *Signal and Urgent Wait*-Monitore bezeichnet. Alternativ sagt man auch, der Monitor erfüllt die *Immediate Resumption Requirement*.

Die Bezeichnung (soweit bekannt) für andere Kombinationen sind in der Tabelle zu finden. Erwähnenswert ist noch die zweite Konstellation, d.h. $E = W < S$. In diesem Fall darf der signalisierende Prozess weiter rechnen. Der entblockierte Prozess wird genauso behandelt wie ein neuer Prozess. Diese Strategie wird in Java verwendet (dies wird später noch genauer erläutert). Problematisch bei dieser Variante ist, dass zum Zeitpunkt zu dem der entblockierte Prozess weiter rechnen darf, die beim Entblockieren wahre Bedingung mittlerweile wieder falsch sein kann. Deshalb wird bei solchen Monitoren ein zirkuläres Prüfen der Bedingung durchgeführt. Wurde die Bedingung in der Zwischenzeit wieder falsch, so wartet der Prozess erneut. Betrachte als Beispiel eine Methode, die den Wert der Variablen s erst dann erniedrigt, wenn dieser ungleich zu Null ist. Die Condition Variable sIstNichtNull wird verwendet, um darauf zu warten, dass s ungleich Null wird. Das zirkuläre Prüfen der Bedingung lässt sich dann ausdrücken als:

```
while s = 0 do
  waitC(sIstNichtNull);
s := s - 1;
```

Das Fazit dieser verschiedenen Monitor-Varianten sollte sein, dass man je nach Monitor-Implementierung der jeweiligen Programmiersprachen leicht andere Programme schreiben muss. Man sollte sich daher bewusst sein, dass nicht alle Monitor-Implementierungen genau das gleiche Verhalten haben.

3.5.3 Monitore mit Condition Expressions

Wie bereits erwähnt, gibt es Monitore die keine Condition Variablen haben, sondern so genannte *Condition Expressions*. Hierbei handelt es sich um Boolesche Ausdrücke, auf deren Wahrheitsgehalt direkt mit der **waitCE** Operation getestet werden kann. Die Operation wird mit **waitCE** bezeichnet, um sie von den anderen wait-Operationen zu unterscheiden. Ist der Boolesche Ausdruck falsch, so wartet der Prozess (er wird blockiert und in eine entsprechende Warteschlange eingefügt).

Für Condition Expressions gibt es keine signal-Operation. Stattdessen werden wartende Prozesse automatisch vom Laufzeitsystem entblockiert, sobald der Boolesche Ausdruck wahr wird. Hierbei gibt es wie bei Condition Variablen mehrere Möglichkeiten, wann diese Prozesse den Monitor-Lock erhalten. Hierauf wird jedoch nicht weiter eingegangen. Zum Schluss der Betrachtung von Condition Expressions wird eine Implementierung für das Erzeuger/Verbraucher-Problem mit einem Bounded Buffer unter Verwendung von Condition Expressions vorgestellt. Hierbei wird davon ausgegangen, dass blockierte Prozesse sofort den Monitor-Lock erhalten, wenn der entsprechende Boolesche Ausdruck wahr wird.

Abbildung 3.27 zeigt die Implementierung des Puffers und den Code für den Erzeuger und den Verbraucher.

```
monitor BoundedBuffer {
  bufferType buffer := empty;

  produce(v) {
    waitCE(length(buffer) < N); // Puffer voll?
    buffer := append(v,buffer);
  }

  consume() {
    waitCE(length(buffer) > 0); // Puffer leer?
    w := head(buffer);
    buffer := tail(buffer);
    return(w);    }
}
```

Code des Erzeugers:

```
loop forever
(1) erzeuge e
(2) BoundedBuffer.produce(e)
end loop
```

Code des Verbrauchers:

```
loop forever
(1) e := BoundedBuffer.consume();
(2) verbrauche e;
end loop
```

Abbildung 3.27: Bounded Buffer als Monitor mit Condition Expressions

3.6 Einige Anwendungsbeispiele mit Monitoren

In diesem Abschnitt werden Implementierungen für klassische Probleme der nebenläufigen Programmierung mit Monitoren erläutert. Dabei werden Monitore mit Condition Variablen und der „Immediate Resumption Requirement“ verwendet, d.h. es werden die Prioritäten $E < S < W$ angenommen.

3.6.1 Das Readers & Writers Problem

Wie bereits in Abschnitt 3.4.7 erläutert besteht das Readers & Writers-Problem darin, möglichst vielen Prozessen parallelen Lesezugriff zu gewähren, während schreibende Prozesse exklusiven Zugriff auf den kritischen Abschnitt (die Datenbank o.ä.) benötigen. In Abbildung 3.28 ist eine Monitor-Implementierung für das Readers & Writers-Problem dargestellt. Der Monitor benutzt zwei Variablen vom Typ int, um die Anzahl der lesenden und der schreibenden Prozesse zu zählen (Variablen countR und countW). Außerdem werden zwei Condition Variablen benutzt: okToWrite und okToRead. An der ersten Condition Variablen okToWrite werden Writer-Prozesse warten, während an der zweiten Condition Variablen okToRead lesende Prozesse darauf warten, mit dem Lesen zu beginnen.

Der Monitor beinhaltet vier Methoden: Jeweils eine Methode zum Starten des Lese- bzw. Schreibzugriffs und jeweils eine Methode nach dem Beenden des Lese- bzw. Schreibzugriffs. Beachte, dass das eigentliche Lesen nicht innerhalb des Monitors geschehen darf, da dann die Monitoreigenschaften dazu führen würden, dass maximal ein Prozess gleichzeitig lesend zugreifen darf. Eine Erläuterung für die beiden Methoden für Writer-Prozesse ist: In der Methode startWrite() prüft ein Writer-Prozess zunächst, ob beide Zähler countR und countW den Wert 0 haben. Ist dies der Fall, so wird der Zähler countW auf 1 gesetzt und anschließend darf der Writer-Prozess den kritischen Abschnitt betreten. In den anderen Fällen muss der Writer-Prozess an der Condition Variablen okToWrite warten. Nach Verlassen des kritischen Abschnitts ruft ein Writer-Prozess die Methode endWrite() auf. Innerhalb dieser Methode wird zunächst die Zählvariable countW um 1 erniedrigt. Anschließend wird die **`empty`**-Methode verwendet, um festzustellen, ob es wartende Lese-Prozesse gibt. Ist dies nicht der Fall, so darf der nächste Writer-Prozess den kritischen Abschnitt betreten. Anderenfalls wird ein Reader-Prozess entblockiert.

Nun werden die Methoden für Reader-Prozesse betrachtet: In der Methode startRead() testet ein lesender Prozess zunächst, ob ein schreibender Prozess bereits im kritischen Abschnitt ist (dann ist countW $\neq 0$) oder ob es wartende Writer-Prozesse gibt. In beiden Fällen wartet der lesende Prozess an der Condition Variable okToRead. Anschließend (oder sofort, wenn nicht gewartet werden musste) wird der Zähler countR inkrementiert. Mithilfe des letzten Befehls der startRead()-Methode entblockiert ein Reader, der gleich in den kritischen Abschnitt treten darf, einen weiteren Reader. Dies setzt sich kaskadierend fort, so dass alle wartenden Leseprozesse gleichzeitig in den kritischen Abschnitt eintreten können. Die endRead()-Methode wird nach dem Verlassen des kritischen Abschnitts aufgerufen. Der Zähler countR wird dekrementiert und der letzte verlassende Prozess entblockiert einen eventuell vorhandenen Writer-Prozess.

Monitor-Definition:

```
monitor RW {
  int countR, countW := 0;
  condition okToWrite, okToRead;

  startRead() {
    if countW ≠ 0 or not(empty(okToWrite))
    then waitC(okToRead);
    countR := countR + 1;
    signalC(okToRead);
  }

  endRead() {
    countR := countR - 1;
    if countR = 0 then signalC(okToWrite);
  }

  startWrite() {
    if countR ≠ 0 or countW ≠ 0
    then waitC(okToWrite);
    countW := countW + 1;
  }

  endWrite() {
    countW := countW - 1;
    if empty(okToRead)
    then signalC(okToWrite);
    else signalC(okToRead);
  }
}
```

Programm des Readers:

```
(1)  RW.startRead();
(2)  Lese;
(3)  RW.endRead();
```

Programm des Writers:

```
(1)  RW.startWrite();
(2)  Schreibe;
(3)  RW.endWrite();
```

Abbildung 3.28: Readers & Writers mit Monitor

3.6.2 Die speisenden Philosophen

Eine mögliche Lösung für das Problem der speisenden Philosophen unter Zuhilfenahme von Monitoren ist in Abbildung 3.29 dargestellt. Da keine Semaphore benutzt werden sollen (bzw. können), werden die Gabeln nicht explizit dargestellt. Stattdessen erhält jeder Philosoph einen Eintrag im forks-Feld. Der Eintrag forks[i] gibt an wieviele der maximal zwei Gabeln momentan dem Philosophen mit Nummer i zur Verfügung stehen. Ein Philosoph kann nur dann essen, wenn beide seiner Gabeln verfügbar sind, d.h. forks[i]=2 erfüllt ist.

Die Operationen $i+1$ und $i-1$ sind dabei „modulo N“ gemeints, sodass $i-1$ für $i=1$ die Zahl N und $i+1$ für $i=1$ die Zahl 1 liefert.

```
monitor Forks {
  constant anzahlPhil := N;
  int array [1,...,anzahlPhil] forks := [2,...,2];
  condition array [1,..., anzahlPhil] okToEat;

  takeForks(i) {
    if forks[i] ≠ 2 then waitC(okToEat[i]);
    forks[i+1] := forks[i+1]-1;
    forks[i-1] := forks[i-1]-1;}

  releaseForks(i) {
    forks[i+1] := forks[i+1]+1;
    forks[i-1] := forks[i-1]+1;
    if forks[i+1] = 2 then signalC(okToEat[i+1]);
    if forks[i-1] = 2 then signalC(okToEat[i-1]);}

}
```

Programm des Philosophen i:

```
loop forever
(1)  Denke;
(2)  Forks.takeForks(i);
(3)  Esse;
(4)  Forks.releaseForks(i);
end loop
```

Abbildung 3.29: Monitor-Implementierung für die speisenden Philosophen

Ein Philosoph ruft die Monitor-Methode takeForks(i) auf, bevor er essen kann. Innerhalb der Methode wird zunächst getestet, ob beide Gabeln vorhanden sind. Ist dies der Fall, so schnappt sich der Philosoph die Gabeln und erniedrigt die Gabel-Einträge seines linken und seines rechten Nachbarn. Sind nicht genügend Gabeln vorhanden, so blockiert der Philosoph an der Condition Variable okToEat[i].

Nachdem der Philosoph das Essen beendet hat, ruft er die releaseForks-Methode auf. Zunächst legt er die Gabeln ab und erhöht die fork-Einträge seiner Nachbarn (diese haben jetzt jeweils wieder eine Gabel mehr). In den letzten beiden **if**-Abfragen schaut der Philosoph nach, ob seinen Nachbarn nun zwei Gabeln zur Verfügung stehen. Ist dies der Fall, so wird der entsprechende Philosoph entblockiert. Beachte, dass von der Immediate Resumption Requirement

ausgegangen wird, d.h. wenn der erste der beiden Nachbarn entblockiert wird, so wird dieser die forks-Einträge modifizieren, bevor der signalisierende Prozesse die forks-Einträge seines anderen Nachbarn abfragt.

Man kann nachweisen, dass diese Implementierung tatsächlich Deadlock-frei ist. Starvation-Freiheit gilt hingegen nicht.

3.6.3 Das Sleeping Barber-Problem

Das Sleeping Barber-Problem wurde bereits in Abschnitt 3.4.4 beschrieben. In Abbildung 3.30 ist eine Implementierung für das Problem mithilfe eines Monitors angegeben.

```
monitor Barbershop {
  int wartend:= 0;
  condition kundenVorhanden;
  condition friseurBereit;
  condition ladenVerlassen;

  // Methoden für den Friseur

  nehmeKunden() {
  if wartend = 0 then
    waitC(kundenVorhanden);
  wartend := wartend - 1;
  signalC(friseurBereit)}

  beendeKunden() {
  waitC(ladenVerlassen)}

  // Methoden für den Kunden

  bekommeFrisur() {
  if wartend < N then
    wartend := wartend + 1;
    signalC(kundenVorhanden)
    waitC(friseurBereit)
    return(True);
  else return(False);}

  verlasseLaden(){
  signalC(ladenVerlassen)}
  }
```

Friseur:

```
loop forever
  Barbershop.nehmeKunden();
  Frisiere;
  Barbershop.beendeKunden();
```

Kunde:

```
if Barbershop.bekommeFrisur()
then
  erhalte Frisur;
  Barbershop.verlasseLaden();
```

Abbildung 3.30: Sleeping Barber mit Monitor

Der Friseur läuft in einer Endlosschleife und ruft zunächst die Monitor-Methode nehmeKunden() auf. Hierbei prüft er anhand der Zählvariablen wartend, ob wartende Kunden vorhanden

sind. Ist dies nicht der Fall, so wartet der Friseur an der Condition Variable kundenVorhanden. Sobald er entblockiert wird, dekrementiert der Friseur die Anzahl der wartenden Kunden und signalisiert einem Kunden, dass er bereit ist, ihn dran zu nehmen (durch eine `signalC`-Operation auf der Condition Variable friseurBereit). Nachdem der Friseur den Kunden frisiert hat, ruft er die Methode beendeKunden() auf. Hierbei wartet er an der Condition Variable ladenVerlassen, bis der gerade frisierte Kunde ihm signalisiert, dass er den Laden verlassen hat.

Die Kundenprozesse sind für diese Lösung durchnummeriert, so dass jeder Kunde eine eindeutig Zahl erhält. Der Kunde ruft zunächst die Monitor-Methode bekommeFrisur auf. Zunächst prüft der Kunde, ob schon n Kunden warten. Ist dies der Fall, so verlässt er den Laden sofort. Andernfalls erhöht er die Anzahl der wartenden Kunden, signalisiert dem Friseur, dass Kunden vorhanden sind und wartet an der Condition Variable friseurBereit darauf, dass der Friseur ihn dran nimmt. Je nachdem, ob alle Warteplätze belegt waren oder nicht, liefert die Methode bekommeFrisur() False oder True zurück. Nur wenn der Wert True zurück geliefert wird, erhält der Kunde seine neue Frisur und ruft anschließend die Methode verlasseLaden auf, um dem Friseur zu signalisieren (an der Condition Variable ladenVerlassen), dass er den Laden verlassen hat.

3.6.4 Barrieren

Barrieren werden für Algorithmen benutzt, die in mehreren Phasen ablaufen, und bei denen es notwendig ist, dass vor dem Eintreten aller Prozesse in die nächste Phase zuvor gewartet wird, bis alle Prozesse die vorherige Phase beendet haben. In Abbildung 3.31 ist eine Implementierung für eine Barriere mittels einem Monitor angegeben. Diese benutzt eine neue Operation auf Condition Variablen, die bisher nicht definiert war:

Die Operation **`signalAllC`**(cond) entblockiert sämtliche Prozesse, die an der Condition Queue der Condition Variable cond warten und fügt sie in die Wait-Queue ein. Dieser Schritt geschieht atomar.

Die Monitor-Implementierung verwendet eine Condition Variable alleAngekommen, eine Variable angekommen zum Zählen der an der Barriere eingetroffenen Prozesse und eine Variable maxProzesse, die die Anzahl der zu synchronisierenden Prozesse angibt.

Ein Prozess führt am Ende seiner Phase einen Aufruf der Methode barrier() aus. Innerhalb dieser Methode wird zunächst angekommen inkrementiert. Anschließend warten die Prozesse an der Condition Variable alleAngekommen, bis auf den letzten Prozess: Dieser Prozess setzt angekommen auf 0 und ruft die **`signalAllC`**-Operation auf der Condition Variable alleAngekommen aus, wodurch alle wartenden Prozesse in die nächste Phase eintreten dürfen.

3.7 Monitore in Java

In diesem Abschnitt wird kurz erläutert, wie Monitore in Java implementiert sind und welche Besonderheiten bei der Benutzung von Monitoren in Java zu beachten sind.

Es gibt in Java kein eigenes Konstrukt, um Monitore zu definieren. Vielmehr kann jedes Objekt wie ein Monitor benutzt werden, d.h. zu jedem Objekt ist ein Lock assoziiert. Um exklusiven Zugriff auf ein Objekt zu garantieren, müssen die entsprechenden Methoden mit dem Schlüsselwort `synchronized` modifiziert werden. Semantisch bedeutet dies, dass eine mit

```
monitor Barrier {
  int angekommen:= 0;
  int maxProzesse := N;
  condition alleAngekommen;

  barrier (){
   angekommen := angekommen + 1;
   if angekommen < maxProzesse then
     waitC(alleAngekommen);
   else
     angekommen := 0;
     signalAllC(alleAngekommen);
   }
  }

  Prozess i:
   loop forever
    Code vor der Barriere
    Barrier.barrier()
    Code nach der Barriere
   end loop
```

Abbildung 3.31: Barrierenimplementierung mit Monitor

`synchronized` gekennzeichnete Methode stets von ausschließlich einem Prozess gleichzeitig ausgeführt werden kann. Der Lock, der verwendet wird, um den exklusiven Zugriff abzusichern, gehört zum Objekt. Wenn mehrere Methoden einer Klasse (d.h. auch eines Objekts) synchronized sind, dann wird der Zugriff über einen einzigen Lock (zugehörig zum Objekt) gesteuert. D.h. für die Menge sämtlicher mit synchronized gekennzeichneter Methoden eines Objekts gilt: Nur ein Prozess führt eine dieser Methoden gleichzeitig aus. Um das klassische Monitor-Verhalten zu implementieren, muss die Klasse somit so programmiert sein, dass *alle* Methoden mit synchronized versehen sind.

Der folgende Code soll dies für eine Klasse mit zwei Methoden andeuten.

```
class MonitoredClass {
 ...Attribute ...

  synchronized method1 {...}


  synchronized method2 {...}
}
```

Da kein direkter Zugriff auf die Attribute für Monitore erlaubt ist, sollten die Attribute der Klasse alle mittels `private` gekennzeichnet sein. Wenn method1 im Rumpf method2 aufruft, so wird in Java der Lock des Objekts (des „Monitors“) beibehalten, der Lock wird erst nach Beenden des ersten Methodenaufrufs zurück gegeben.

Nochmal zur Klarstellung, der Programmierer ist dafür verantwortlich, alle Methoden als synchronized zu deklarieren, d.h. es ist durchaus erlaubt ein Klasse wie folgt zu definieren:

```
class NotReallyMonitoredClass {
 ...Attribute ...
  synchronized method1 {...}

  method2 {...}
}
```

In diesem Fall kann stets nur ein Thread method1 ausführen, aber die Ausführung von method2 ist unsynchronisiert, beliebig viele Threads können somit gleichzeitig method2 ausführen.

Java verfügt auch über eine Variante von Condition Variablen, wobei es pro Objekt nur eine einzige Condition Variable gibt (diese existiert implizit). Da es nur eine Condition Variable pro Objekt gibt, haben die **`waitC`**- bzw. **`signalC`**-Operationen keine formalen Parameter. Die Benennung der Operationen ist auch leicht verschieden von der bisher verwendeten Definition. Es gibt die folgenden Methoden für Condition Variablen:

- wait(): Der Thread wartet an der Queue des Objekts
- notify(): Einer der am Objekt wartenden Threads wird entblockiert
- notifyAll(): Alle der am Objekt wartenden Threads werden entblockiert

Beachte, dass in Java beim Aufruf von notify() der Lock des Objekts beim aufrufenden Thread bleibt. Zudem haben wartende Threads die gleiche Priorität, den Lock des Monitors (bzw. Objekts) zu erlangen, wie neue Threads, d.h. die Prioritäten der Queues für Condition Variablen sind geordnet als $W = E < S$. Dadurch ist es möglich, dass ein entblockierter Prozess erst dann wieder den Lock erhält, wenn die Bedingung (auf die er gewartet hatte) erneut nicht erfüllt ist. Dies umgeht man, indem man zirkulär die Bedingung prüft und erneut wartet, solange die Bedingung falsch ist, d.h. in Pseudo-Code anstelle von

```
if (not Bedingung) wait() ;
```

die **`while`**-Schleife

```
while (not Bedingung)
  wait();
```

benutzt.

Da Objekte nur eine einzige Condition Variable anbieten, bleibt zu klären, wie man Algorithmen mit mehreren verschiedenen Condition Variablen in Java programmiert. Man würde gerne nur die entsprechenden wartenden Prozesse bezüglich einer wahr-werdenden Bedingung entblockieren, d.h. man würde gerne ein Programm der Art

```
synchronized method1() {
 if (x==0)
   wait();  }

synchronized method2() {
 if (y==0)
   wait();  }
...
if some condition
    notify(auf x ≠ 0 wartende Prozesse)
else
    notify(auf y ≠ 0 wartende Prozesse)
```

schreiben. Diese Verwendung von **notify** ist allerdings nicht erlaubt. Man kann dies jedoch lösen, indem man die wartenden Prozesse zirkulär warten lässt und anstelle von **notify** den Aufruf **notifyAll** verwendet.

```
synchronized method1() {
 while (x==0)
   wait();  }

synchronized method2() {
 while (y==0)
   wait();  }
...
if some condition
    notifyAll();
else
    notifyAll();
```

Erwähnenswert ist noch, dass der Modifizierer **synchronized** nicht nur für Methoden, sondern für beliebige Codeblöcke verwendet werden kann. In diesem Fall muss das Objekt angegeben werden dessen Lock verwendet werden soll. Mutual-Exclusion kann somit in Java auch wie folgt implementiert werden:

```
Object obj = new Object();

synchronized (obj) {
  Kritischer Abschnitt
}
```

Abschließend werden zwei Beispielprogramme in Java betrachtet. Ein bounded Buffer zur Lösung des Erzeuger / Verbraucher-Problems kann wie folgt implementiert werden:

```
import java.util.LinkedList;

// Die Klasse fuer den Buffer

class BBufferMon<V> {
```

```
 private LinkedList<V> buffer = new LinkedList<V>();
 private Integer length; // Fuellstand
 private Integer size;   // maximale Gr"oesse

 BBufferMon(Integer n){
  size = n;
  length = 0;
}

 synchronized void produce(V elem) {
      while (length == size) {
        try {wait();} catch (InterruptedException e) {};
      }
      buffer.add(elem);
      length++;
      notifyAll();
 }

 synchronized public V consume() {
      while (length == 0) {
        try {wait();} catch (InterruptedException e) {};
      }
      V e = buffer.removeFirst();
      length--;
      notifyAll();
      return e;
 }
}
```

Zum Testen wird noch der Rest der Codes gezeigt. Zunächst zwei Klassen für Erzeuger und Verbraucher:

```
class Producer extends Thread {
  BBufferMon<Integer> buff;
  Integer number;
  Producer(BBufferMon<Integer> b, Integer i) {
     buff = b;
     number = i;
 }
  public void run() {
    for (int i = 1; i <= 10; i++) {
      buff.produce(i);
    }
  }
}
```

```
class Consumer extends Thread {
  BBufferMon<Integer> buff;
  Integer number;
  Consumer(BBufferMon<Integer> b,Integer i) {
     buff = b;
     number = i;
 }
  public void run() {
    for (int i = 1; i <= 50; i++)  {
      Integer e = buff.consume();
    }
  }
}
```

Und schließlich noch eine ausführbare Klasse, die einen Puffer der Größe 5 erzeugt und anschließend einige Erzeuger und Verbraucher erzeugt.

```
class Main {
public static void main(String[] args) {
 // Puffer-Objekt erzeugen
      BBufferMon<Integer> b = new BBufferMon<Integer>(5);
 // Erzeuger-Threads erzeugen
      for (int i=1; i <= 50; i++) {
        Producer q = new Producer(b,i);
        q.start();
       }
 // Verbraucher-Threads erzeugen
      for (int i=1; i <= 10; i++) {
        Consumer q = new Consumer(b,i);
        q.start();
       }
      while (true) {}
    }
}
```

Eine Lösung zum Readers/Writers Problem kann wie folgt implementiert werden (aus (Ben-Ari, 2006)):

```
class RWMonitor {
  volatile int readers = 0;
  volatile boolean writing = false;

  synchronized void StartRead() {
    while (writing)
      try {
         wait();
      } catch (InterruptedException e) {}
```

```
    readers = readers + 1;
    notifyAll();
  }

  synchronized void EndRead() {
    readers = readers - 1;
    if (readers == 0) notifyAll();
  }

  synchronized void StartWrite() {
    while (writing || (readers != 0))
      try {
         wait();
      } catch (InterruptedException e) {}
    writing = true;
  }
  synchronized void EndWrite() {
    writing = false;
    notifyAll();
  }
}
```

3.8 Kanäle

Die bisher untersuchten Programmierkonstrukte – Semaphore und Monitore – benutzen gemeinsamen Speicher, um die Kommunikation von Prozessen zu ermöglichen. In diesem Abschnitt werden so genannte *Kanäle* (engl. Channels) betrachtet, die benutzt werden, um ohne gemeinsamen Speicher, sondern über den Austausch von Nachrichten zu kommunizieren. Solche Kanäle können auch in Systemen mit gemeinsamen Speicher implementiert werden, aber der Kernpunkt ist, dass sie nicht notwendigerweise gemeinsamen Speicher benötigen. Deshalb eignen sich Kanäle beispielsweise für verteilte Systeme. In diesem Kapitel wird die *synchrone* Kommunikation (d.h. synchrone Kanäle) betrachtet, d.h. der Datenaustausch zwischen zwei Prozessen geschieht in einem Schritt und nicht *asynchron*. Wenn ein sendender Prozess senden möchte, aber der empfangende Prozess noch nicht bereit ist, wird der sendende Prozess solange blockiert bis das Senden stattfinden kann. Umgekehrt werden Empfänger solange blockiert, bis ein noch nicht bereiter sendender Prozess bereit zum Senden ist.

Kanäle wurden von C.A.R. Hoare im so genannten „Communicating Sequential Processes“-Formalismus (CSP) eingeführt. Diese Art der nebenläufigen Programmierung hatte Einfluss auf einige Programmiersprachen und wird zumindest als Bibliothek für einige Programmiersprachen bereit gestellt. In der Programmiersprache Go (http://golang.org) sind Kanäle nativ eingebaut.

3.8.1 Definition von Kanälen

Ein Kanal verbindet einen sendenden Prozess mit einem empfangenden Prozess. Kanäle sind im Allgemeinem typisiert, d.h. nur Elemente (d.h. Nachrichten) gleichen Typs dürfen über

einen Kanal versendet werden. Kanäle haben einen Namen und es gibt zwei Operationen auf Kanälen. Sei ch ein Kanal, dann schreibt man

- ch $\Leftarrow$ w, wenn der Wert w über den Kanal ch versendet werden soll. Hierbei muss w ein Wert passendes Typs sein. Der Einfachheit halber schreibt man auch ch $\Leftarrow$ x für eine Programmvariable x und meint damit, dass der Wert der Variablen x über den Kanal ch versendet wird.
- ch $\Rightarrow$ y, bedeutet, dass der aufrufende Prozess ein Element über den Kanal ch empfangen möchte. Der empfangene Wert wird der Programmvariablen y zugewiesen. Beachte, dass im Konstrukt ch $\Rightarrow$ e nur Programmvariablen an der Position von e zulässig sind.

Die Ausführung nebenläufiger Programme mit Kanaloperationen verlangt dann, dass die Kommunikation zwischen sendendem und empfangendem Prozess erst dann stattfindet, wenn beide Programmzeiger auf den entsprechenden Operationen angelangt sind.

Zur Verdeutlichung betrachte das folgende Beispiel zum Lösen des Erzeuger/Verbraucher Problem, wobei nicht beliebig viele Elemente im Puffer gespeichert werden können, sondern dieser Puffer eigentlich gar kein Element zwischenspeichern kann. Abbildung 3.32 stellt den Programmcode für den Erzeuger- und den Verbraucherprozess dar.

ch: Kanal über dem Typ τ
y: Programmvariable vom Typ τ

Erzeuger:	Verbraucher:
loop forever	**loop forever**
(1) erzeuge e (vom Typ τ);	(1) ch $\Rightarrow$ y;
(2) ch $\Leftarrow$ e;	(2) verbrauche y;
end loop	**end loop**

Abbildung 3.32: Erzeuger / Verbraucher mit Kanal

Die beiden Prozesse müssen zum Kommunizieren über den Kanal ch in den entsprechenden Code-Zeilen sein (der Erzeuger in Zeile (2), der Verbraucher in Zeile (1)). Die Ausführung dieser beiden Zeilen geschieht dann atomar in einem Schritt, d.h. der Wert wird in einem Schritt vom Verbraucher an den Erzeuger versendet. Ist beispielsweise der Erzeuger in Zeile (2), aber der Verbraucher noch beim Verbrauchen eines vorherig empfangenden Elements (also auch in seiner Zeile (2)), dann wird der Erzeuger-Prozess automatisch blockiert und muss solange warten, bis der Verbraucher in seiner Zeile (1) ist. Ein analoger Fall tritt auf, wenn der Erzeuger noch in Zeile (1) ist, der Verbraucher aber in seiner Zeile (1) ist und empfangen möchte: Dann wird der Verbraucher solange blockiert, bis der Erzeuger in Zeile (2) ist. Dieses Blockieren und entblockieren geschieht automatisch.

Es stellt sich die Frage, was passiert, wenn beispielsweise zwei Prozesse auf dem selben Kanal senden möchten an dem ein Empfänger bereit steht. In diesem Fall ist zunächst nicht klar, welcher der sendenden Prozesse seine Nachricht an den Empfänger verschicken darf. Tatsächlich ist es so, dass dies quasi zufällig vom Scheduler entschieden wird. Jede Möglichkeit kann auftreten. Betrachte folgendes Beispiel mit drei Prozessen:

Prozess 1:	Prozess 2:	Prozess 3:
ch $\Leftarrow$ True	ch $\Leftarrow$ False	ch $\Rightarrow$ x
		print x;

Prozesse 1 und 2 versuchen beide über den Kanal ch eine Nachricht zu versenden. Je nach Ausführungreihenfolge wird Prozess 3 die Nachricht von Prozess 1 oder die Nachricht von Prozess 2 empfangen. D.h. bei unterschiedlichen Abläufen des Programms wird entweder True oder False ausgedruckt. Dieser Nichtdeterminismus tritt natürlich auch dann auf, wenn es mehrere Empfänger aber nur einen Sender für einen Kanal gibt. Man könnte einerseits argumentieren, dass es sich in diesem Fall um Race Conditions handelt, da der Ablauf des Programms das Ergebnis beeinflusst. Andererseits ist ein solcher Effekt oft gewünscht. Betrachte z.B. die Situation, dass es mehrere Server gibt, die alle den gleichen Dienst anbieten (und am selben Kanal darauf warten eine Anfrage eines Clients zu bearbeiten): Wenn viele Clients auf die Server zugreifen wird quasi immer ein freier Server zufällig ausgewählt.

3.8.2 Mutual-Exclusion mit Kanälen

Nun wird das Mutual-Exclusion-Problem bei Verwendung von Kanälen betrachtet. Der wechselseitige Ausschluss lässt sich mit Kanälen beispielsweise implementieren, indem man einen Prozess als Wächter für den kritischen Abschnitt definiert. Dieser Wächterprozess ist mit einem Kanal zu allen Prozessen, die in den kritischen Abschnitt eintreten möchten, verbunden. Erst nachdem der Wächter eine Nachricht an einen der anderen Prozesse gesendet hat, darf dieser den kritischen Abschnitt betreten. Im Abschlusscode sendet der Prozess dem Wächter eine Nachricht, wodurch dieser weiß, dass der kritische Abschnitt wieder frei ist. Abbildung 3.33 zeigt den Code zu dieser Idee. Allerdings kann die Nachricht nach dem Verlassen des kritischen Abschnitts auch an einen anderen warten Prozess geschickt werden (sie muss nicht unbedingt beim Wächter ankommen), der dann direkt in den kritischen Abschnitt eintreten kann, (d.h. der Wächter wird sozusagen übergangen). Dies ist jedoch kein Problem. Der Wächter ist jedoch notwendig: Betrachte z.B. den Fall, dass nur ein Prozess in den kritischen Abschnitt möchte. Ohne Wächter, käme er weder in kritischen Abschnitt herein, bzw. wieder heraus.

Die Lösung erfüllt den wechselseitigen Ausschluss und ist Deadlock-frei, da ein Prozess stets eine Nachricht erhalten wird. Sie ist jedoch nicht Starvation-frei, da stets ein Prozess „ignoriert“ werden kann und nie eine Nachricht erhält. Beachte, dass dieser Fall nicht durch die Fairness-Annahme ausgeschlossen wird, da der wartende Prozess nicht jederzeit einen Auswertungsschritt ausführen kann (dies kann er nur, wenn der Wächter sendebereit ist).

3.8.3 Modellierung von gemeinsamen Speicher durch Kanäle

Obwohl kein gemeinsamer Speicher im Kanalmodell vorhanden ist, über den die Prozesse kommunizieren können, kann man diesen mithilfe von Kanälen nachbilden. Hierfür wird eine einzige Speicherzelle betrachtet, die gelesen und beschrieben werden kann. Es gibt einen Server-Prozess, der die Zelle verwaltet und die Lese- und Schreibeoperationen annimmt. In Abbildung 3.34 ist der Programmcode für den Server und die Implementierung der Zugriffsmethoden read und write angegeben. Die Implementierung benutzt zwei Kanäle: Der Kanal requestChannel wird benutzt, um Anfragen an die Zelle zu senden, der Kanal replyChannel wird während einer read-Anfrage benutzt, um das Ergebnis (den Wert der Zelle) an den anfragenden Prozess zurück zu senden. Der Kanal requestChannel erwartet Paare, wobei die erste Komponente ein boolescher Wert ist (dieser entscheidet darüber, ob es sich um eine Lese- oder Schreibeanfrage handelt (True = schreibender Zugriff, False = lesender Zugriff)) und die zweite Komponente vom Typ des Zelleninhalts ist (abstrakt als CellType). Die zweite Komponente spielt nur eine Rolle

mutex: Kanal über dem Typ Bool
local: lokale Variable, Initialwert egal

Prozess i
loop forever
(1) Restlicher Code;
(2) mutex ⇒ local;
(3) Kritischer Abschnitt;
(4) mutex ⇐ True;
end loop

Wächter
loop forever
(1) mutex ⇐ True;
(2) mutex ⇒ local;
end loop

Abbildung 3.33: Mutual-Exclusion mit Kanälen

bei Schreibzugriffen, denn sie beinhaltet den neu zu setzenden Wert der Zelle. Das Serverprogramm wartet auf eine Nachricht am requestChannel. Je nachdem, ob die erste Komponente der Nachricht wahr oder falsch ist, wird entweder der Wert der Zelle neu gesetzt oder der bisherige Wert über den replyChannel an den fragenden Prozess gesendet. Die Programme für die beiden Zugriffsmethoden sind analog, wobei eine lesende Anfrage noch einen Dummy-Wert für die zweite Komponente des Paares mitschicken muss.

Man kann auf die lokale Variable x im Serverprozess verzichten, wenn man den Serverprozess rekursiv definiert. Diese Variante ist in Abbildung 3.35 dargestellt.

3.8.4 Selective Input

Bei vielen Sprachen, die auf CSP beruhen, gibt es noch ein weiteres Konstrukt, um eine angenehmere Programmierung zu ermöglichen. Dabei handelt es sich um den so genannten „selective input“: Bisher war es so, dass ein Nachrichten-empfangender Prozess an genau einem Kanal auf eine Nachricht warten muss. Mittels „selective input“ ist es möglich an mehreren Kanälen zu warten und, sobald eine Nachricht an *einem* Kanal empfangen wird, fortzufahren. Dies lässt sich schreiben als eine Ver-Oder-ung der Empfangsprimitive:

```
either
  ch1 ⇒ var1
or
  ch2 ⇒ var2
or
  ch3 ⇒ var3
```

Führt ein Prozess dieses Kommando aus, so kann er Nachrichten auf den Kanälen ch1, ch2 oder ch3 empfangen. Gibt es mehrere verschiedene Sender, so wählt die Auswertung zufällig

Datentypen:
CellType sei der Typ der Zelle, z.B. Bool, Int, ...
RequestChannelType = (Bool,CellType)

Kanäle:
requestChannel : Kanal über dem Typ RequestChannelType
replyChannel: Kanal über dem Typ CellType

Variablen:
x: Lokale Variable des Server-Prozesses vom Typ CellType
dummy: Irgendeine Variable vom Typ CellType

Prozess für die Zelle (Server-Prozess):

```
loop forever
  requestChannel ⇒ r;
  if fst(r) then// write-Operation
    x := snd(r);
  else // read-Operation
    replyChannel ⇐ x;
end loop
```

Methoden für den Zugriff auf die Zelle:

```
read(requestChannel, replyChannel) {
  requestChannel ⇐ (False,dummy);
  replyChannel ⇒ x;
  return(x);
}

write(reqCh, replyCh, x) {
  requestChannel ⇐ (True,x)
}
```

Abbildung 3.34: Implementierung einer Zelle mit Kanälen

Prozess für die Zelle (Server-Prozess):

```
cell(x) {
  requestChannel ⇒ r;
  if fst(r) then// write-Operation
    cell(snd(r));
  else // read-Operation
    replyChannel ⇐ x;
    cell(x); }
```

Abbildung 3.35: Prozess für die Zelle mit Rekursion

(bzw. nichtdeterministisch) eine der möglichen Kommunikationen aus. Dadurch lassen sich viele Probleme eleganter lösen. Z.B. kann ein Serverdienst auf mehreren Kanälen darauf warten, dass er eine Anfrage eines Clients erhält. Sobald auf einem der Kanäle eine Anfrage kommt, bearbeitet der Server nur diese. Implizit werden weitere Clients auf anderen Kanälen hierdurch blockiert, bis der Server wieder bereit ist (und die nichtdeterministische Auswahl bereitstellt).

In einem späteren Kapitel werden noch genauere theoretische Grundlagen zu solchen Message-Passing-Modellen erörtert.

3.8.5 Speisende Philopsophen mit Kanälen

Wir betrachten eine Lösung für das Problem der speisenden Philosophen mithilfe von Kanälen. Hierbei wird jede Gabel durch einen Prozess dargestellt, der mit einem Kanal sowohl mit dem linken als auch mit dem rechten Philosophen-Prozess verbunden ist. Abbildung 3.36 stellt den Code für die Philosophen und die Gabeln dar. Ein Philosoph versucht nacheinander beide Gabeln zu erhalten, indem er über die zugehörigen Kanäle etwas (`True`) empfängt. War dies erfolgreich, so kann er essen. Anschließend legt er die Gabeln zurück, indem er über die zu den Gabeln zugehörigen Kanäle `True` zurück sendet. Die Gabel-Prozesse machen nichts anderes als abwechselnd `True` zu senden und anschließend etwas zu empfangen. Sind Gabeln bereits belegt, so wird der entsprechende Philosoph automatisch blockiert, da er keine Nachricht empfangen kann.

Diese Implementierung ist zunächst nur eine Modellierung aber noch keine Lösung des Problems, da ein Deadlock auftreten kann (jeder Philosoph könnte die linke Gabel haben, dann kann keiner die rechte erlangen). Man kann aber die Programme leicht modifizieren, um eine korrekte Lösung zu erreichen. Z.B. könnte der letzte Philosoph die Gabeln in umgekehrter Reihenfolge hoch heben.

3.8.6 Kanäle in der Programmiersprache Go

Die Programmiersprache Go unterstützt Känäle nativ. Zum Initialisieren eines Kanals wird `make(chan type)` verwendet. Diese Anweisung öffnet einen Kanal mit Inhalt vom Typ `type`, z.B. kann mit

```
kanal := make(chan string)
```

ein Kanal geöffnet werden, auf dem Strings gesendet und empfangen werden. Die Anweisung zum Senden (ch $\Leftarrow$ w in Pseudo-Code) schreibt man in Go als `ch <- w`. Z.B. kann mit

```
kanal <- "Hallo"
```

der String `"Hallo"` auf den an `kanal` gebundenen Kanal gesendet werden.

Für das Empfangen (ch $\Rightarrow$ x in Pseudo-Code) schreibt man in Go `x := <- ch`. Z.B. kann mit

```
x := <- kanal
```

vom Kanal `kanal` ein Element empfangen werden und an die Variable `x` gebunden werden. Wenn man das empfangene Element nicht benötigt, so schreibt man in Go:

forks : Feld von Kanälen über dem Typ Bool
x: lokale Variablen

Philosoph i
loop forever
(1) Denke;
(2) forks[i] ⇒ x
(3) forks[i+1]⇒ x
(4) Esse;
(5) forks[i] ⇐ True
(6) forks[i+1] ⇐ True
end loop

Gabel i
loop forever
(1) forks[i] ⇐ True
(2) forks[i] ⇒ x
end loop

Abbildung 3.36: Speisende Philosophen mit Kanälen (nicht Deadlock-Frei)

```
    <- kanal
```

Ein Beispielprogramm (entnommen von https://de.wikipedia.org/wiki/Go_(Programmiersprache)) zum Senden und Empfangen in Go ist:

```
package main

import "fmt"

func zehnMal(kanal chan string) {
 // Argument empfangen
 sag := <-kanal
 // Zehnmal zurückschreiben
 for i := 0; i < 10; i++ {
  kanal <- sag
 }
 // Kanal schließen
 close(kanal)
}

func main() {
 // synchronen Kanal öffnen
 kanal := make(chan string) // oder make(chan string, 0)
 // Starten der parallelen Go-Routine „zehnMal()“
 go zehnMal(kanal)
 // Senden eines Strings
 kanal <- "Hallo"
 // Empfangen der Strings, bis der Channel geschlossen wird
 for s := range kanal {
  fmt.Println(s)
 }
```

```
 fmt.Println("Fertig!")
}
```

Die Go-Routine `zehnMal` schreibt empfängt dabei zunächst eine Zeichenkette über den Kanal und sendet diesen anschließend zehn Mal über den Kanal und schließt ihn danach. Das Hauptprogramm öffnbet den Kanal, startet die Go-Routine `zehnMal` und sendet anschließend die Zeichenkette `"Hallo` an den Kanal. Schließlich empfängt das Hauptprogramm die durch die `zehnMal`-Routine gesendeten Strings und druckt diese aus.
Die Mutal-Exclusion-Lösung mit Kanälen aus Abschnitt 3.8.2 kann in Go wie folgt programmiert werden, wobei im kritischen Abschnitt hier zur Demonstration mehrere Ausgaben auf der Standardausgabe erfolgen:

```
package main

import ("fmt"
        "strconv"
        "bufio"
        "os"
)

func worker(mutex chan bool,  work string) {
 for {
  <- mutex
  fmt.Println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
  fmt.Println("> Thread " + work + " jetzt im Kritischen Abschnitt")
  fmt.Println("> Thread " + work + " noch im Kritischen Abschnitt")
  fmt.Println("> Thread " + work + " verlaesst gleich den Kritischen Abschnitt")
  fmt.Println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
  mutex <- true
 }
}

func guard(mutex chan bool) {
 for {
  mutex <- true
  <- mutex
 }
}

func main() {
 // synchronen Kanal öffnen
 mutex := make(chan bool)
 go guard(mutex)
 for i := 0; i < 10; i++ {
  go worker(mutex,strconv.Itoa(i))
 }
 // Auf Benutzereingabe warten
 reader := bufio.NewReader(os.Stdin)
 reader.ReadString('\n')
}
```

In der Programmiersprache Go kann mithilfe der `select`-Primitive an mehreren Kanälen gleichzeitig gewartet werden.

Anstelle von

either
  ch1 ⇒ var1
or
  ch2 ⇒ var2
or
  ch3 ⇒ var3

schreibt man in Go

```
select {
  case var1 := <- ch1:
    code1
  case var2 := <- ch2:
    code2
  case var3 := <- ch2:
    code3
}
```

Ein Beispiel unter Verwendung von `select` ist durch den folgenden Code gegeben. Hier wartet der Hauptthread an zwei Kanälen gleichzeitig auf den Empfang von Nachrichten:

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func sleepAndWriteToChannel(c chan string,content string) {
 var n = rand.Intn(1000)
 time.Sleep(time.Duration(n) * time.Millisecond)
 c <- content
}

func main() {
 c1 := make(chan string)
 c2 := make(chan string)

 // Go-Routine starten, die erst wartet und dann "one" auf den Kanal c1 schreibt
 go sleepAndWriteToChannel(c1,"one")

 // Go-Routine starten, die erst wartet und dann "two" auf den Kanal c2 schreibt
 go sleepAndWriteToChannel(c2,"two")

 for i := 0; i < 2; i++ {
  // Gleichzeitiges Lauschen an  Kanaelen c1 und c2 "selective input"
  select {
   case msg1 := <-c1:
    fmt.Println("received", msg1)
   case msg2 := <-c2:
    fmt.Println("received", msg2)
  }
 }
}
```

Mittels `select` kann in Go sowohl auf das Empfangen als auch auf das Senden (oder auch eine Mischung davon) gewartet werden. Im folgenden Beispiel sendet oder empfängt der Haupthread an den beiden Kanälen:

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func sleepAndWriteToChannel(c chan string,content string) {
        var n = rand.Intn(1000)
        time.Sleep(time.Duration(n) * time.Millisecond)
        c <- content
}

func sleepAndReceive (c chan string) {
        var n = rand.Intn(1000)
        time.Sleep(time.Duration(n) * time.Millisecond)
         <- c
}

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    // Go-Routine starten, die erst wartet und dann "one" auf den Kanal c1 schreibt
    go sleepAndWriteToChannel(c1,"one")

    // Go-Routine starten, die erst wartet und dann  "two" auf den Kanal c2 schreibt
    go sleepAndWriteToChannel(c2,"two")

    // Go-Routine starten, die erst wartet und dann ein Element von c1 liest
    go sleepAndReceive(c1);

    // Go-Routine starten, die erst wartet und dann ein Element von c2 liest
    go sleepAndReceive(c2);


    for i := 0; i < 4; i++ {
    // Gleichzeitiges Lauschen und Schreiben an den Kanaelen c1 und c2
        select {
         case msg1 := <-c1:
            fmt.Println("received", msg1)
         case msg2 := <-c2:
            fmt.Println("received", msg2)
         case c2 <- "three":
            fmt.Println("send three on c2")
         case c1 <- "three":
            fmt.Println("send three on c1")
         }
    }
}
```

Schließlich zeigen wir noch die (Deadlock-freie) Implementierung des Problems der speisenden Philosophen mit Kanälen (siehe Abschnntt 3.8.5) in Go:

```
package main

import ("fmt"
        "strconv"
        "bufio"
        "os"
)

func fork (forks[](chan bool),i int) {
 for {
  forks[i] <- true
  <- forks[i]
 }
}

func philosopher (forks[](chan bool),i int) {
 for {
  fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Denke...")
  if (i == 9) {
      <- forks[0]
      fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Habe rechte Gabel")
  } else {
      <- forks[i]
      fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Habe linke Gabel")
  }
  if (i == 9) {
      <- forks[i]
      fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Habe linke Gabel")
  } else {
      <- forks[i+1]
      fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Habe rechte Gabel")
  }
  fmt.Println("Philosoph: " + strconv.Itoa(i) + ": Esse...")
  forks[i] <- true
  if ( i == 9 ) {
      forks[0] <- true
  } else {
      forks[i+1] <- true
}}}

func main() {
 //Gabeln erstellen
 forks:=make([](chan bool),10)
 for i := range forks{
  forks[i] = make(chan bool)
 }
 for i:=0; i < 10; i++ {
  go fork(forks,i)
 }
 // Philosophen erzeugen
 for i:=0; i < 10; i++ {
   go philosopher(forks,i)
 }
 // Auf Eingabe warten
 reader := bufio.NewReader(os.Stdin)
 reader.ReadString('\n')
}
```

3.9 Tuple Spaces: Das Linda-Modell

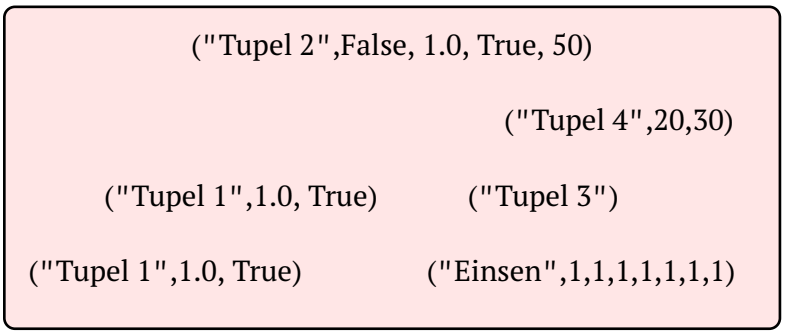
Die im vorherigen Abschnitt vorgestellten synchronen Kanäle haben neben ihren vorgestellten Vorteilen auch einige Nachteile: Der Name des Kanals, über den Sender und Empfänger kommunizieren, muss beiden Kommunikationspartnern bekannt sein. Gerade bei der Implementierung von Client/Server-Architekturen ist dies nachteilig, da z.B. der Server seine Kanalnamen exportieren muss bzw. die Clients entsprechend der Kanalnamen und des entsprechenden Servers konfiguriert werden müssen. Ein weiterer Nachteil des Kanalmodells besteht darin, dass nur aktive Prozesse Nachrichten verschicken und senden können. Hat ein Nachrichtenerzeugender Prozess beispielsweise seine Berechnung längst beendet und könnte terminieren, wenn er seine Nachricht dem Empfänger geschickt hat, so muss er im Kanalmodell solange aktiv bleiben, bis der Empfänger die Nachricht entgegen genommen hat.

In diesem Abschnitt wird ein weiteres Modell vorgestellt, was einige dieser Nachteile behebt, und sich durch seine einfache Verständlichkeit auszeichnet. Das so genannte Linda-Modell (bzw. Linda Programmiersprache), ist eine Koordinationssprache, d.h. es ist eine Sprache, die auf der eigentlichen Programmiersprache aufgesetzt ist, um Prozesskommunikation zu ermöglichen. Es gibt einige Implementierungen von Linda in verschiedenen Programmiersprachen. Prominente Beispiele sind C-Linda, Suns JavaSpaces und TSpace entwickelt von IBM.

Das wesentliche Konstrukt von Linda ist der so genannte Tuple Space (Tupel Raum). Hierbei handelt es sich um einen „Platz“ in dem Tupel (von Daten) abgelegt und entnommen werden können. Alle Prozesse können über Operationen auf den Tuple Space zugreifen. In manchen Implementierungen gibt es die Möglichkeit, mehrere verschiedene Tuple Spaces anzulegen. Der Einfachheit halber wird nur von einem einzigen Raum ausgegangen.

Jedes Tupel im Tuple Space hat einen Typ, d.h. alle Komponenten des Tupel sind getypt. Ein Beispiel für ein Tupel ist ("MeinErstesTupel":String, 1.0:Float,True:Bool). Im folgenden werden die Typen an den Komponenten weggelassen, wenn dies nicht zu Mehrdeutigkeiten führt. In einem Tuple Space können beliebig viele Tupel beliebiger Typen enthalten sein (in den Implementierungen können oft nur Basistypen für die Tupelkomponenten verwendet werden). Als weitere Konvention sei die erste Komponente eines Tupels stets ein String ist, der das Tupel beschreibt.

Man kann sich einen Tuple Space wie eine Tafel mit Notizzetteln vorstellen. Auf den Notizzetteln stehen die einzelnen Tupel. Die folgende Abbildung soll einen Tuple Space mit sechs Tupeln darstellen.



3.9.1 Operationen auf dem Tuple Space

Zum Einfügen, Entnehmen und Lesen von Tupeln stellt Linda drei Basisoperationen zur Verfügung:

- Die Operation **out**$(N, v_1, \ldots, v_n)$ fügt das Tupel $(N, v_1, \ldots, v_n)$ in den Tupel Space ein. N ist hierbei der Name des Tupels (also ein String). Die Komponenten $v_1, \ldots, v_n$ sind Werte (des entsprechenden Typs) oder Programmvariablen. Für Programmvariablen werden vor dem Einfügen des Tupels in den Raum die aktuellen Variablenbelegungen gelesen und deren Werte für das einzufügende Tupel genommen.

  Beispielsweise führt die Befehlssequenz

  ```
  (1) out("Tupel 1", 10, True)
  (2) x := 50;
  (3) b := True;
  (4) out("Tupel 2", x, 100, b)
  ```

  dazu, dass in den Tupel Space die beiden Tupel ("Tupel 1",10,True) und ("Tupel 2",50,100,True) eingefügt werden. Beachte, dass es erlaubt ist, gleiche Tupel mehrfach in den Tupel Space einzufügen.
- Die Operation **in**$(N, x_1, \ldots, x_n)$ versucht ein Tupel aus dem Tupel Space zu entfernen. N ist hierbei ein Name (String), und $x_1, \ldots, x_n$ sind Programmvariablen, die durch die **in**-Operation gebunden werden. Die **in**-Operation entnimmt ein *passendes* (engl. matching) Tupel. Passend heißt hierbei, dass das Tupel im Tupel Space genauso viele Komponenten, pro Komponente den gleichen Typ, und den identischen Namen zum Anfragetupel $(N, x_1, \ldots, x_n)$ haben muss. Z.B. sind die Tupel ("Tupel",10:Int, 20:Int) und ("Tupel",50:Int, 60:Int) beide passend zum Anfragetupel ("Tupel",x:Int, y:Int), während weder ("Tupel",10.0:Float, 20:Int), ("Tupel 2",10:Int, 20:Int) noch ("Tupel",10:Int) passend sind. Gibt es ein oder mehrere passende Tupel im Tuple Space so wird nichtdeterministisch eines der passenden Tupel ausgewählt, vom Tuple Space entfernt und die Variablen des Anfragetupels auf die entsprechenden Werte des entnommenen Tupels gesetzt. Nehme an, für die Operation **in**("Tupel",x:Int, y:Int) wird das Tupel ("Tupel",10:Int, 20:Int) gewählt, dann wird anschließend die Variable x auf den Wert 10 und die Variable y auf den Wert 20 gesetzt.

  Gibt es kein passendes Tupel im Tuple Space, so *blockiert* der Prozess, der die **in**-Operation ausführt, solange, bis ein passendes Tupel eingefügt wird. Warten mehrere Prozesse auf das Einfügen eines passendes Tupels und ein neu eingefügtes Tupel passt für mehrere dieser Prozesse, so wird wiederum nichtdeterministisch einer der wartenden Prozesse ausgewählt, der entblockiert wird und seine **in**-Operation beenden darf.
- Die Operation **read**$(N, x_1, \ldots, x_n)$ entspricht semantisch der **in**-Operation mit dem einzigen Unterschied, dass das passende Tupel im Tuple Space nicht entfernt wird, sondern erhalten bleibt. Beachte, dass auch eine **read**-Operation zum Blockieren führt, wenn kein passendes Tupel vorhanden ist. Wird ein passendes Tupel anschließend eingefügt, so werden *alle* **read**-ausführenden Prozesse entblockiert, für die das Tupel passt.

Neben diesen drei Operationen, gibt es meistens noch zwei weitere Operationen:

- Die Operation **inp**$(N, x_1, \ldots, x_n)$ verhält sich wie die **in**-Operation, allerdings ist sie nicht blockierend. Ist kein passendes Tupel vorhanden, tritt eine Exception auf, die beim Aufruf abgefangen werden kann.

- Die Operation **`readp`**$(N, x_1, \ldots, x_n)$ verhält sich wie die **`readp`**-Operation, allerdings ist sie nicht blockierend. Ist kein passendes Tupel vorhanden, tritt eine Exception auf, die beim Aufruf abgefangen werden kann.

Linda stellt noch eine weitere Operation **`eval`** zur Verfügung, um nebenläufige Berechnungen von Tupelkomponenten zu starten (so genannte „active tupels“). Es wird im folgenden auf diese Operation verzichtet, da das Erzeugen von Prozessen durch vorhandene Konstrukte der Programmiersprache erfolgen kann.

3.9.2 Die Bibliothek pSpaces und goSpace

Das Projekt pSpaces (Programming with Spaces)[1] ist ein akademisches Projekt zur Implementierung und Entwicklung von Tuple Space-Implementierungen und ihrer Anwendungen. Das Projekt stellt Implementierungen für Java, C#, Go, JavaScript und Swift bereit. Wir erläutern die Verwendung in der Programmiersprache Go. Die Bibliothek kann mittels

```
import ("github.com/pspaces/gospace")
```

importiert werden. Ein neuer (lokaler) Tuple Space wird mit `NewSpace` erstellt. Z.B. kann man programmieren:

```
meinTupleSpace := gospace.NewSpace("Name des Tuple Space")
```

Tupel sind über den Typ `Tuple` verfügbar. Sie können z.B. mit `CreateTuple` erzeugt werden und mit `GetFieldAt` kann auf die Komponenten zugegriffen werden. Z.B.:

```
var tuple Tuple = CreateTuple("Milch", 1)
tuple.GetFieldAt(i)
```

Die Typen der Komponenten müssen beim Zugriff gecasted werden:

```
var mtuple Tuple = CreateTuple("Milch", 1)
var was string
was  = (mtuple.GetFieldAt(0)).(string)
var wieviel int
wieviel = (mtuple.GetFieldAt(1)).(int)
```

Die Operationen auf dem Tuple Space heißen anders, als die vorgestellten:

`Put`$(x_1, x_2, \ldots, x_n)$	entspricht	`out`$(x_1, \ldots, x_n)$
`Get`$(x_1, x_2, \ldots, x_n)$	entspricht	`in`$(x_1, \ldots, x_n)$
`Query`$(x_1, x_2, \ldots, x_n)$	entspricht	`read`$(x_1, \ldots, x_n)$

Es gibt nicht-blockierende Varianten: `GetP`, `QueryP`. Alle Operationen haben als Rückgabe ein Paar `(tuple, error)`.

Die Operation `Put` erzeugt direkt ein Tupel aus den Argumenten und fügt es ein

[1] siehe `https://github.com/pSpaces`

```
// Tuple Space erzeugen
meinTupelSpace := NewSpace("MeinTupelSpace")
var tuple Tuple = CreateTuple("Milch", 1)
// Tupel einfügen
meinTupelSpace.Put(tuple.GetFieldAt(0),tuple.GetFieldAt(1))
meinTupelSpace.Put("Butter",2)
meinTupelSpace.Put("Saft","Orange",3)
meinTupelSpace.Put("Saft","Apfel",2)
meinTupelSpace.Put
```

Beachte, dass `meinTupelSpace.Put(tuple)` ein einstelliges Tupel einfügt, welches selbst aus einem Paar besteht (es wird `((Milch,1))` und *nicht* `(Milch,1)` in den Tuple Space eingefügt).

Tupel können mit `Get` bzw. nicht-blockierend mit `GetP` entnommen werden (`Query` und `QueryP` funktionieren analog, aber lesen das Tupel nur, es verbleibt im Tuple Space).

```
tuple2, _ := meinTupelSpace.Get("Butter",2)
fmt.Println(tuple2)
// ein weiteres
meinTupelSpace.Get("Butter",2)
// blockiert
// nicht blockierend:
_, err:= meinTupelSpace.GetP("Butter",2)
fmt.Println(err)
```

Die Ausführung der letzten Zeile druck den Fehler aus:

```
Space.main("Butter", 2): operation on this space failed.
```

Sollen die Werte von Variablen durch `Get` oder `Query` gebunden werden, dann müssen die Adressen der Variablen anstelle der Variablen übergeben werden (d.h. statt `x` muss `&x` übergeben werden) und zusätzlich müssen nach dem Aufruf die Werte extrahiert und gebunden werden. Betrachte z.B.

```
var numberOSaft int = 0
t,_ := meinTupelSpace.GetP("Saft","Orange",&numberOSaft)
numberOSaft = t.GetFieldAt(2).(int)  // notwendig!
```

Wird `GetP("Saft","Orange",numberOSaft)` statt `GetP("Saft","Orange",&numberOSaft)` ausgeführt, so wird nach dem Tupel `("Saft","Orange",0)` gesucht.

3.9.3 Einfache Problemlösungen mit Tuple Spaces

In diesem Abschnitt werden einige Grundprobleme der nebenläufigen Programmierung und deren Implementierung im Linda-Modell betrachtet.

Der wechselseitige Ausschluss kann mit einem einzigen Tupel im Tuple Space implementiert werden. Abbildung 3.37 zeigt das Programm für Prozess i, wobei davon ausgegangen wird, dass

initial ein Tupel ("MUTEX") im Tuple Space vorhanden ist. Um in den kritischen Abschnitt zu gelangen, muss der Prozess mittels einer **in**-Operation das Tupel aus dem Tupel Space entfernen. Da nun kein passendes Tupel mehr im Tupel Space vorhanden ist, wird jeder weitere Prozess beim Ausführen von Zeile (2) blockiert. Nach Verlassen des kritischen Abschnitts wird das Tupel ("MUTEX") durch eine **out**-Operation wieder in den Tuple Space gelegt. Es ist einfach zu prüfen, dass die Implementierung den wechselseitigen Ausschluss garantiert und Deadlockfrei ist. Bei zwei Prozessen ist die Implementierung auch Starvation-frei, da die **out**-Operation in Zeile (4) einen in Zeile (2) wartenden Prozess entblockieren muss. Für mehr als 2 Prozesse ist die Implementierung nicht Starvation-frei.

Initial: Tupel ("MUTEX") im Tuple Space

Prozess i:
loop forever
(1) Restlicher Code;
(2) **in**("MUTEX");
(3) Kritischer Abschnitt;
(4) **out**("MUTEX");
end loop

Abbildung 3.37: Mutual-Exclusion mit Linda

In Go und unter Verwendung von goSpace kann das Beispiel programmiert werden als:

```
package main
import ("fmt" ; . "github.com/pspaces/gospace"; "strconv"; "bufio"; "os")
func worker(id int, space Space) {
 for {
  space.Get("Mutex")
  fmt.Println("a) Worker " + strconv.Itoa(id)+ " im kritschen Abschnitt.")
  fmt.Println("b) Worker " + strconv.Itoa(id)+ " im kritschen Abschnitt.")
  fmt.Println("c) Worker " + strconv.Itoa(id)+ " im kritschen Abschnitt.")
  space.Put("Mutex")
 }
}
func main() {
 // Tuple Space erzeugen
 space := NewSpace("MeinTupelSpace")
 space.Put("Mutex")
 for i:=0; i < 10; i++ {
  go worker(i,space)
 }
 // Auf Benutzereingabe warten
 reader := bufio.NewReader(os.Stdin)
 reader.ReadString('\n')
}
```

Als weiteres Beispiel betrachte, wie Semaphore mit Tuple Spaces modelliert bzw. simuliert werden können. Hierbei sei k der Wert mit dem der Semaphor initialisiert wird. Wir nehmen dann an, dass im Tupel Space k viele Tupel ("SEM") vorhanden sind. Die Semaphor-Operationen **wait** und **signal** können dann implementiert werden als

```
wait(){              signal(){
  in("SEM");           out("SEM");
}                    }
```

Nach k-maligen Ausführen der wait-Operationen, sind alle k Tupel entfernt und jede weitere wait-Operation führt zum Blockieren. Die signal-Operation fügt ein neues Tupel ("SEM") in den Tuple Space ein. Gibt es blockierte Prozesse, so wird einer der wartenden Prozess durch das Einfügen entblockiert.

Die Operationen für den Tuple Space ermöglichen es auch Tupel atomar (d.h. ohne Race Conditions) zu aktualisieren. Das Programm hierfür ist einfach:

```
(1)  in("Tupel",x);
(2)  x := f(x);
(3)  out("Tupel",x);
```

Da in Zeile (1) das Tupel aus dem Tuple Space entfernt wird, kann kein anderer Prozess mehr auf das Tupel zugreifen. In Zeile (2) wird der erhaltene Wert für x benutzt, um einen neuen Wert zu berechnen (durch Anwenden der Funktion f). Erst nachdem das aktualisierte Tupel in Zeile (3) in den Tuple Space gelegt wurde, können andere Prozesse wieder auf das Tupel zugreifen.

3.9.4 Erweiterung der in-Operation

Bisher wurde angenommen, dass Anfragetupel $(N, x_1, \ldots, x_n)$ der **in**-Operation neben dem Namen nur aus Variablen x_i bestehen, die wie formale Parameter wirken: Sie werden durch Ausführen der **in**-Operation an Werte gebunden. Variablen der **out**-Operation wirken hingegen wie aktuelle Parameter: Ihr aktueller Wert wird für das einzufügende Tupel benutzt. Man kann Anfragetupel jedoch auch um aktuelle Parametervariablen erweitern (diese Variablen werden im folgenden mit einem nachgestellten Gleichheitszeichen, z.B. $x =$ notiert). Die Semantik für solche Anfragetupel besteht darin, dass der aktuelle Wert der Variablen mit dem Wert des passenden Tupels *übereinstimmen* muss. Betrachte als Beispiel die folgenden beiden Programme:

x:=10; **in**("Tupel 1", x=);	x:=10; **in**("Tupel 1", x);

Während das rechte Programm, alle Tupel matcht, die als erste Komponente den String "Tupel 1" haben und als zweite Komponente eine Zahl vom Typ Int, matcht das linke Programm nur Tupel der Form ("Tupel 1",10). Ein weiterer Unterschied ist, dass das linke Programm den Wert von x unverändert lässt, während das rechte Programm den Wert von x auf den erhaltenen Wert setzt.

In Go und goSpace ist die Notation verschieden: x in Go verhält sich wie x= in Linda und &x in Go verhält sich wie x in Linda.

Mithilfe der erweiterten Tupel ist flexiblere Programmierung möglich. Betrachte als Beispiel zwei Server, die jeweils einen Dienst anbieten. Anfragen durch Clients erfolgen über Tupel.

Hierbei muss sichergestellt werden, dass Clients nur mit solchen Servern verbunden werden, die auch den richtigen Dienst anbieten. Server führen eine **in**-Operation aus, um Anfragen von Clients zu bearbeiten. Ohne die erweiterten Tupel müsste der entsprechende Dienst im Tupelnamen kodiert werden, damit der richtige Server den richtigen Client bearbeitet. Das kann sich durch das ganze Programm ziehen, überall unterscheiden sich beide Server an diesen Stellen. Mithilfe der erweiterten Tupel kann der Dienst als Tupelkomponente kodiert werden. Die **in**-Operation benutzt ein Anfragetupel der Form $(\ldots, \text{dienst}=, \ldots)$ wobei dienst eine vorherig definierte Variable ist. Je nach angebotenem Dienst wird die Variable entsprechend belegt. Die Programmierung der beiden Server wird dadurch vereinheitlicht, da die **in**-Operation identisch ist, nur die Belegung von dienst unterscheidet sich.

Als weiteres Beispiel wird die Simulation von synchronen Kanälen mithilfe von Tuple Spaces erörtert. Die beiden Operationen zum Senden und Empfangen lassen sich wie folgt implementieren:

- ch $\Leftarrow$ e:

  ```
  get := "get";
  got := "got";
  in("ch1",get=);
  out("ch2",e);
  in("ch3",got=);
  ```

- ch $\Rightarrow$ x:

  ```
  out("ch1","get");
  in("ch2",x);
  out("ch3","got");
  ```

Beachte, dass die Implementierung sicherstellt, dass beide Prozesse warten, bis die Kommunikation stattgefunden hat. Des Weiteren werden die Variablen get und got verwendet, um bei **in**-Anfragen auf entsprechende Strings zu matchen. Im Folgenden werden diese Hilfsvariablen nicht immer definiert, sondern direkt die entsprechenden Werte in das Anfragetupel der **in**-Anfrage geschrieben. D.h. anstelle von **in**("ch1",get=) wird die Anfrage als **in**("ch1","get") geschrieben.

3.9.5 Beispiele

In diesem Abschnitt betrachten werden einige klassische Beispiele der nebenläufigen Programmierung und Lösungen hierfür in Linda betrachtet.

Betrachte zunächst die Implementierung einer *Barriere* (siehe Abschnitt 3.4.6). Initial befindet sich im Tuple Space ein Tupel ("ankommen",N), wobei N die Anzahl der Prozesse ist, die sich an der Barriere synchronisieren. In Abbildung 3.38 ist der Code für Prozess i und die Prozedur barrier() dargestellt.

Ein Prozess ruft nach Beenden einer Phase die Prozedur barrier() auf. Dort wird zunächst das Tupel ("ankommen",i) dekrementiert (beachte dies geschieht atomar). Anschließend wartet der Prozess darauf, dass er das Tupel mit Namen "verlassen" im Tuple-Space findet. Dies wird vom letzten Prozess der angekommen ist bewerkstelligt. Analog muss der letzte verlassende Prozess das initial Tupel mit dem Namen "ankommen" wieder in den Tuple-Space legen.

Als nächstes Beispiel wird das Problem der speisenden Philosophen betrachtet. Dabei wird die

Initial: Ein Tupel ("ankommen",N);

Prozess i:
```
loop forever
  Code vor der Barriere
  barrier()
  Code nach der Barriere
end loop
```

```
barrier (){
//Ankommen
 in("ankommen",x);
 x := x-1;
 if x > 0
   {out("ankommen",x);}
 else
   {out("verlassen",N);} //Letzter Ankommer
 in("verlassen",y);
 y := y-1;
 if y > 0
   {out("verlassen",y);}
 else
   {out("ankommen",N);} //Letzter Verlasser
}
```

Abbildung 3.38: Barriere mit Tuple Spaces

gleiche Idee wie bei der Lösung mit Semaphore und einem Wächter (der Algorithmus aus Abbildung 3.8) verfolgt, d.h. es wird versucht sicherzustellen, dass nur $N-1$ Philosophen gleichzeitig an die Gabeln dürfen. Die Gabeln werden durch Tupel ("Gabel",i) modelliert, wobei für Philosoph i ("Gabel",i) seine linke und ("Gabel",$i+1 \bmod N$) seine rechte Gabel darstellt. Die Gabeln liegen initial im Tuple Space. Des Weiteren liegen $N-1$ Tupel ("Raum") im Tupel Space. Der Algorithmus für Philosoph i ist in Abbildung 3.39 dargestellt.

Ein Philosoph versucht zunächst eines der ("Raum")-Tupel aus dem Tuple Space zu entfernen. Da anfänglich nur $N-1$ solcher Tupel vorhanden sind, muss ein Philosoph (wenn alle N zugreifen) dort warten. Anschließend nimmt der Philosoph seine Gabeln aus dem Tuple Space, isst und legt die Gabeln zurück. Zum Schluss legt er das Tupel ("Raum") zurück. Diese Lösung ist Deadlock-frei und Starvation-frei.

Als weiteres Beispiel wird das Erzeuger/Verbraucher-Problem betrachtet. In Abbildung 3.40 ist der Code für die Erzeuger und Verbraucher angegeben. Es werden Tupel ("notFull") und ("notEmpty") verwendet, um Erzeuger bei leerem Buffer und Verbraucher bei vollem Buffer zu blockieren. Anfänglich liegen N Tupel ("notFull") im Tupel Space. Dadurch können N Erzeuger-Aufrufe durchgeführt werden, bis der Puffer voll ist. Der Puffer wird durch Tupel der Form ("buffer",e) dargestellt. Hierbei ist e ein Element im Puffer. Beachte, dass der Puffer in diesem Fall keine FIFO-Queue ist, d.h. das erste eingefügte Element wird nicht notwendigerweise als erstes entnommen. Vielmehr erfolgt die Entnahme zufällig. Man kann die Implementierung erweitern und eine FIFO-Queue mithilfe von Tupeln implementieren. Diese Erweiterung ist in Abbildung 3.41 dargestellt. Zur Repräsentation der FIFO-Queue werden die Tupel ("FIFO-Queue","tail",i) – ein Zeiger auf das Ende der Queue, ("FIFO-Queue",head",i) – ein Zeiger auf das erste Elemente der Queue, und Tupel der Form ("buffer",i,e) verwendet. Die letzteren Tupel stellen die Elemente e an Position i der Queue dar.

Beim Einfügen in die Queue wird zunächst der Index des letzten Elements der Queue ermittelt

```
philosoph(i) {
  loop forever
    Denke;
    in("Raum");
    in("Gabel",i=);
    j := i+1 mod N;
    in("Gabel",j=);
    Esse;
    out("Gabel",i);
    out("Gabel",j);
    out("Raum");
  end loop
}
```

Abbildung 3.39: Philosophen in Linda

Initial: N Tupel ("notFull") im Tuple-Space

```
produce(Element e) {
  in("notFull");
  out("buffer",e);
  out("notEmpty");
}
```

Code des Erzeugers:

```
loop forever
(1)   erzeuge e
(2)   produce(e)
end loop
```

```
consume() {
  in("notEmpty");
  in("buffer",x);
  out("notFull");
  return(x);
}
```

Code des Verbrauchers:

```
loop forever
(1)   e := consume();
(2)   verbrauche e;
end loop
```

Abbildung 3.40: Erzeuger / Verbraucher in Linda

Initial: N Tupel ("notFull") im Tuple-Space
("FIFO-Queue","tail",0)
("FIFO-Queue","head",0)

```
produce(Element e) {
  in("notFull");
  in("FIFO-Queue","tail", tindex);
  tindex' = tindex +1;
  out("buffer", tindex', e);
  out("FIFO-Queue", "tail", tindex');
  out("notEmpty");
}

consume() {
  in("notEmpty");
  in("FIFO-Queue", "head", hindex);
  hindex' = hindex+1;
  in("buffer", hindex=, x);
  out("FIFO-Queue", "head", hindex');
  out("notFull");
  return(x);
}
```

Abbildung 3.41: Erzeuger / Verbraucher mit FIFO-Queue

und um eins erhöht. Anschließend wird an dieser Stelle das neue Element eingefügt. Beim Entfernen wird zunächst der Index des ersten Elements der Queue gelesen, anschließend das Element mit diesem Index entfernt und schließlich der Index des ersten Elements auf das nächste Element (Index + 1) gesetzt.

Als abschließendes Beispiel wird das Readers & Writers-Problem betrachtet. Eine Lösung benutzt ein Tupel der Form ("RW",Anzahl Readers, Anzahl Writers). Am Anfang befindet sich das Tupel ("RW",0,0) im Tuple Space. Abbildung 3.42 stellt den Code für lesende und schreibende Prozesse dar.

Initial: Tupel ("RW",0,0) im Tuple Space

Methoden:

```
startread() {
  in("RW",countR,0);
  countR' := countR + 1;
  out("RW",countR',0);
}

startWrite() {
  in("RW",0,0);
  out("RW",0,1);
}
```

```
stopRead() {
  in("RW",countR,0);
  countR' := countR - 1;
  out("RW",countR',0);
}

stopWrite() {
  in("RW",0,1);
  out("RW",0,0);
}
```

```
Reader:
  startRead();
  Kritischer Abschnitt;
  stopRead();
```

```
Writer:
  startWrite();
  Kritischer Abschnitt;
  stopWrite();
```

Abbildung 3.42: Readers & Writers in Linda

Lesende Prozesse dürfen nur dann zugreifen, wenn keine schreibenden Prozesse vorhanden sind, d.h. sie warten darauf, dass das Tupel die Form ("RW",i,0) hat . Anschließend erhöhen sie die Anzahl der Reader um 1. Schreibende Prozesse dürfen nur dann zugreifen, wenn es weder lesende noch andere schreibende Prozesse gibt. Deshalb warten sie darauf, dass das Tupel die Form ("RW",0,0) hat. Um Reader- und Writer-Prozesse zu blockieren, schreiben sie das Tupel ("RW",0,1) zurück in den Tuple Space. Nach Verlassen des kritischen Abschnitts setzen lesende Prozesse die Anzahl an lesenden Prozesse um 1 herunter, schreibende Prozesse setzen beide Komponenten auf 0.

Diese Lösung gibt lesenden Prozessen den Vorrang, da schreibende Prozesse stets warten müssen, wenn immer wieder Reader-Prozesse hinzukommen. Man kann eine Variante angeben, die Writer-Prozesse bevorzugt. Hierbei wird das Tupel um eine Komponente erweitert, die die Anzahl der wartenden Writer-Prozesse zählt. Lesende Prozesse dürfen nur dann zugreifen, wenn es keine wartenden Schreiber gibt. Abbildung 3.43 stellt den Code für diese Variante dar.

Initial: Ein Tupel ("RW",0,0,0)

```
startread() {
  in("RW",countR,0,0);
  countR' := countR + 1;
  out("RW",countR',0,0);
 }

startWrite() {
 in("RW",countR,countW,waitingW);
 w' := waitingW+1;
 out("RW",countR,countW,w');
 in("RW",0,0,waitingW);
 w' := waitingW-1;
 out("RW",0,1,w');
}
```

```
stopRead() {
 in("RW",countR,0,waitingW);
 countR' := countR - 1;
 out("RW",countR',0,waitingW);
}

 stopWrite() {
  in("RW",0,1,waitingW);
  out("RW",0,0,waitingW);
  }
```

Abbildung 3.43: Priorität für Writer-Prozesse

3.10 Quellennachweis

Die Darstellung von Semaphore, Monitoren, Kanälen und Linda folgt im wesentlichen (Ben-Ari, 2006), wobei teilweise die Syntax und einige Algorithmen aus (Taubenfeld, 2006) entnommen wurden. Die Klassifizierung verschiedener Typen von Monitoren wurde in (Buhr et al., 1995) genauer untersucht. Die Implementierung von Zellen mithilfe von synchronen Kanälen richtet sich nach (Reppy, 2007), wobei einige Modifikation vorgenommen wurden. Eine Übersicht über Linda mit vielen Beispielen findet man in (Carriero & Gelernter, 1992; Gelernter, 1985), woraus auch einige der Beispiele entnommen wurden.

4

Zugriff auf mehrere Ressourcen

4.1 Deadlocks bei mehreren Ressourcen

In diesem Abschnitt werden Deadlocks betrachtet, die auftreten können, wenn Prozesse *mehrere* Ressourcen belegen möchten. In Kapitel 2 wurden bereits Deadlocks bzw. Deadlock-Freiheit für das Mutual-Exclusion-Problem definiert. Hierbei hatten alle Prozesse jedoch ein ähnliches Programm und sie betraten nur einen kritischen Abschnitt. Dies kommt dem Erlangen *einer* Ressource gleich. Nun werden Deadlocks allgemeiner gefasst und das Problem betrachtet, dass Prozesse mehrere Ressourcen belegen möchten und diese durch Locks (mit Semaphoren, Monitoren, etc.) vor dem Zugriff durch andere Prozesse sperren.

Definition 4.1.1. *Eine Menge von Prozessen ist* deadlocked*, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, dass nur ein anderer Prozess aus der Menge herbei führen kann.*

Das Ereignis ist dabei im allgemeinen das Freigeben einer Ressource.

Einige Situationen, in denen ein solcher Deadlock auftreten kann, sind:

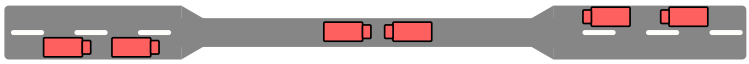
- Beim Mutual-Exclusion-Problem tritt ein Deadlock auf, wenn es nicht mehr möglich ist, dass irgendein Prozess den kritischen Abschnitt betritt, obwohl es Prozesse gibt, die den kritischen Abschnitt betreten möchten.
- Zwei Prozesse, die Guthaben von einem Bankkonto auf ein anderes Bankkonto transferieren möchten. Damit der Saldo der Konten konsistent bleibt, sperren die Prozesse den Zugriff durch andere Prozesse auf die Konten während ihrer Transaktion. Die Sperrung der Konten erfolgt nacheinander: Zunächst wird das Konto gesperrt, von dem Geld abgebucht wird, anschließend das Konto, dem der Betrag gutgeschrieben wird. Prozess P möchte Guthaben von einem Bankkonto A auf Bankkonto B überweisen, während Prozess Q Guthaben von Konto B auf Konto A überweisen möchte. Um das Beispiel explizit zu machen, seien KontoA und KontoB binäre Semaphore, um die beiden Konten zu sperren. Prozesse P und Q führend dann folgende Programme aus:

Prozess P	Prozess Q
wait(KontoA);	wait(KontoB);
wait(KontoB);	wait(KontoA);
buche von A nach B	buche von B nach A
signal(KontoA);	signal(KontoB);
signal(KontoB);	signal(KontoA);

  In einer unglücklichen Ausführungsreihenfolge kann das Programm in einem Deadlock enden: Prozess P belegt Semaphor KontoA, anschließend belegt Prozess Q Semaphor KontoB. Nun warten beide Prozesse jeweils an dem anderen Semaphor, der nie freigegeben wird. Eine bessere Lösung wäre für dieses Problem, dass beide Prozesse versuchen

die Konten in der gleichen Reihenfolge zu sperren. Tatsächlich kann dann kein Deadlock auftreten.

- Ein ähnliches Beispiel zum vorherigen, wurde bereits bei den falschen Lösungen zum Problem der speisenden Philosophen behandelt. Diese Programme können im Deadlock enden, wenn alle Philosophen die jeweils linke Gabel haben.
- Ein anschauliches Beispiel ist das Überqueren (von Fahrzeugen) von Brücken oder anderen Engstellen, die nur eine Fahrspur haben. Fahren die Fahrzeuge unkontrolliert in die Engstelle herein, kann es zum Deadlock kommen. Das folgende Bild veranschaulicht die Situation



Im Allgemeinen gibt es vier Arten mit Deadlocks umzugehen:

Ignorieren: Man trifft gar keine Vorkehrungen und geht einfach davon aus, dass Deadlocks nur selten auftreten und deshalb nicht stören. Tatsächlich ist diese Herangehensweise nicht selten anzutreffen.

Deadlock-Erkennung und -Beseitigung: Hierbei wird davon ausgegangen, dass das Laufzeitsystem Deadlocks erkennen kann und diese beseitigt. Das zu lösende Problem hierbei ist es, einen Algorithmus zu finden, der Deadlocks im jeweiligen System erkennt und ein Verfahren bzw. auch eine Semantik zum Auflösen von Deadlocks (z.B. alle Locks freigeben o.ä) anzugeben.

Deadlock-Vermeidung: Hierbei wird davon ausgegangen, dass das System bereits beim Zugriff auf die Ressourcen / Locks mögliche Deadlocks erkennen kann und solche Locks gar nicht erst zulässt. Das Ziel ist es hierbei, einen Algorithmus (Prozess) für die Ressourcenverwaltung zu implementieren, der dafür sorgt, dass der Zugriff auf Ressourcen ohne Deadlock abläuft.

Deadlock-Verhinderung: Der Programmierer entwirft die Programme so, dass Deadlocks nicht auftreten können.

Die letzte Lösung ist (aus Sicht des Laufzeitsystems) offensichtlich die einfachste und aus Sicht des Programmierers die sicherste. Die interessante Frage ist daher, wie man Deadlocks bei der Programmierung vermeiden kann.

Definition 4.1.2 (Notwendige Bedingungen für einen Deadlock)**.** *Damit ein Deadlock auftreten kann, müssen die folgenden vier Bedingungen alle gleichzeitig erfüllt sein:*

Wechselseitiger Ausschluss (Mutual-Exclusion): *Nur ein Prozess kann gleichzeitig auf eine Ressource zugreifen.*

Halten und Warten (Hold and Wait): *Ein Prozess kann eine Ressource anfordern (auf eine Ressource warten), während er eine andere Ressource bereits belegt hat.*

Keine Bevorzugung (No Preemption): *Eine Ressource kann nur durch den Prozess freigegeben (entsperrt) werden, der sie belegt hat.*

Zirkuläres Warten: *Es gibt eine zyklische Abhängigkeit zwischen wartenden Prozessen: Jeder wartende Prozess möchte Zugriff auf die Ressource, die der nächste Prozesse im Zyklus belegt hat.*

Will man Deadlocks verhindern, genügt es daher, dafür Sorge zu tragen, dass mindestens eine der vier Bedingungen nie erfüllt ist. In den nächsten beiden Abschnitten werden zunächst Me-

thoden zur Deadlock-Verhinderung und anschließend ein Verfahren zur Deadlock-Vermeidung erörtert.

4.2 Deadlock-Verhinderung

Um Deadlocks bei der Programmierung / beim Design zu verhindern, genügt es, eine der vier notwendigen Bedingungen aus Definition 4.1.2 anzugreifen.

Der wechselseitige Ausschluss kann in manchen Situationen verhindert werden, indem man einen „Spooler“ dazwischen schaltet. Z.B. kann man den Zugriff auf den Drucker nicht-exklusiv konstruieren, indem man einen Druckserver zwischenschaltet, der die Druckaufträge der einzelnen Prozesse entgegennimmt. Allerdings gibt es viele Situationen, in denen solche Lösungen nicht funktionieren.

Die „Keine Bevorzugung“-Bedingung anzugreifen, macht wenig Sinn, da man dann z.B. Prozessen den Drucker entzieht, obwohl sie noch gar nicht fertig gedruckt haben.

Die „Halten und Warten“-Bedingung kann man angreifen, indem man fordert, dass ein Prozess zu Beginn alle Ressourcen auf einmal anfordert (blockiert), die er benötigt. Das Problem dabei ist allerdings, dass es oft schwierig ist, zu Beginn zu wissen, was alle benötigten Ressourcen sind. Hier werden möglicherweise viel zu viel Ressourcen blockiert. Eine solche Lösung war das Philosophen-Programm, welches nur einem Philosophen den exklusiven Zugriff auf alle Gabeln gibt. Wie wir bereits gesehen haben, endete dies in einer sehr schlechten Ausnutzung der Ressourcen (nur zwei der N Gabeln waren maximal gleichzeitig in Gebrauch).

Ein Variation des Ansatzes ist das so genannte Zwei-Phasen-SperrprotOkoll (engl. „Two Phase Locking“): Jeder Prozess arbeitet dabei in zwei Phasen:

1. Phase: Der Prozess versucht alle benötigten Ressourcen zu belegen. Ist eine benötigte Ressource nicht frei, so gibt der Prozess alle belegten Ressourcen zurück und startet mit Phase 1 von vorne.
2. Phase: Der Prozess hat alle benötigten Ressourcen. Er kann nun fertig rechnen. Danach gibt er alle Ressourcen frei.

Will man einen solchen Algorithmus beispielsweise mit Semaphore implementieren, benötigt man nicht-blockierende Operationen, die im Falle von nicht erfolgreichem Belegen der Semaphore nicht blockieren, sondern beispielsweise eine abfangbare Ausnahme zurück gegeben.

Für das Philosophenproblem würde eine solche Implementierung bedeuten: Alle Philosophen versuchen in Phase 1 beide Gabeln hochzuheben. Erhält ein Philosoph nicht beide Gabeln, so legt er die Gabel, die er evtl. erhalten hat zurück. Den nächsten Versuch startet der Philosoph erst dann, wenn wieder Phase 1 an der Reihe ist. Die Phasen kann man beispielsweise mit Barrieren synchronisieren. Allerdings ist die Synchronisierung der Phasen über alle Prozesse nicht unbedingt notwendig.

Man kann nachweisen, dass es in diesem Fall keinen Deadlock geben wird. Allerdings kann ein anderes Problem auftreten: Ein so genannter *Livelock*: Keiner der Prozesse erhält alle benötigten Ressourcen in Phase 1 und das immer wieder. Für das Philosophen-Problem kann ein solcher Livelock auftreten, wenn alle Philosophen die linke Gabel erhalten und sie wieder hinlegen, da keiner die rechte Gabel erhält. Charakteristisch an einem Livelock ist, dass die Prozesse im Gegensatz zum Deadlock nicht blockiert werden, aber kein Prozess in seiner eigentlichen Berechnung voran schreitet.

Eine weitere Methode die Hold-and-Wait-Bedingung anzugreifen, besteht im so genannten *Timestamping-ordering* Verfahren. Die Idee dabei ist es, eine Ordnung auf den Prozessen einzuführen: Wollen zwei Prozesse die gleiche Ressource belegen, so gewinnt der kleinere der beiden Prozesse. Bevor ein Prozess damit beginnt die Ressourcen wie beim 2-Phase-Locking zu belegen, erhält er einen eindeutigen Zeitstempel. Jeder spätere Prozess erhält einen Zeitstempel, der echt größer ist als der vorherige. Das Zwei-Phasen-SperrprotOkoll mit Timestamping läuft dann wie folgt ab:

1. Phase: Der Prozess versucht alle benötigten Ressourcen auf einmal zu sperren. Ist eine benötigte Ressource bereits belegt und der Zeitstempel des belegenden Prozesses ist kleiner als sein eigener Zeitstempel, dann gibt der Prozess alle Ressourcen frei und beginnt von vorne mit Phase 1. Ist sein Zeitstempel jedoch kleiner als der des belegenden Prozesses, so wartet der Prozess darauf auch die restlichen Ressourcen zu belegen. Dies wird immer gelingen, da sich alle Prozesse dem ProtOkoll unterwerfen.
2. Phase: Wenn der Prozess erfolgreich in diese Phase gekommen ist, hat er alle benötigten Ressourcen. Er benutzt sie und gibt sie anschließend wieder frei.

Genauer nehmen wir dabei an, dass ein neuer Zeitstempel nur nach erfolgreichem Durchlauf des Prozesses vergeben wird, wobei bei Neustart in Phase 1 kein neuer Zeitstempel vergeben wird.

Für die Philosophen bedeutet dies, dass die Philosophen Zeitstempel erhalten und diese den Gabeln, die sie belegen, mitteilen. Bei bereits belegten Gabeln müssen die Philosophen herausfinden, ob der eigene Zeitstempel kleiner ist als der Zeitstempel des belegenden Prozesses und dem entsprechend handeln.

Tatsächlich erhält man mit diesem Verfahren Deadlock- und Starvationfreiheit. Auch Livelocks können nicht auftreten. Der Implementierungsaufwand ist jedoch immens. Hierbei wird Starvation allgemeiner gefasst (ohne Einschränkung auf das Mutual-Exclusion-Problem) definiert als:

Definition 4.2.1. *Starvation ist eine Situation, in der ein Prozess niemals (nach beliebig vielen Berechnungsschritten) in der Lage ist, alle benötigten Ressourcen zu belegen.*

Ein Problem beim Zwei-Phasen-SperrprotOkoll (mit Timestamping) ist jedoch, dass jeder Prozess a priori wissen muss, welche Ressourcen er benötigt, damit er all diese in der ersten Phase sperren kann.

Die effektivste Methode zur Deadlock-Verhinderung besteht darin die Bedingung des „Zirkulären Wartens“ zu verhindern. Beim Philosophen-Problem war die dazu passende Lösung, den N. Philosophen die Gabeln in umgekehrter Reihenfolge aufheben zu lassen. Dadurch war das Aufnehmen der Gabeln nicht länger zyklisch. Man konnte auf den Gabeln eine totale Ordnung erkennen, wobei Gabel 1 die kleinste Gabel, und Gabel N die größte Gabel war. Alle Philosophen haben die Gabeln entsprechend ihrer Ordnung belegt (deswegen hat der letzte Philosoph zuerst die rechte Gabel (Gabel 1) und anschließend die linke Gabel (Gabel N) belegt). Das folgende Theorem zeigt, dass man mit totaler Ordnung Deadlocks vermeiden kann:

Theorem 4.2.2 (Totale-Ordnung Theorem)**.** *Sind alle gemeinsamen Ressourcen durch eine totale Ordnung geordnet und jeder Prozess belegt seine benötigten Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung, dann ist ein Deadlock unmöglich.*

Beweis. Der Beweis erfolgt durch Widerspruch. Die Annahme ist daher zunächst, dass es einen Deadlock gibt, obwohl die Ressourcen entsprechend der totalen Ordnung belegt wurden. Da es einen Deadlock gibt, muss es ein zirkuläres Warten geben, d.h. es gibt Ressourcen $R_1, \ldots, R_n$ und Prozesse $P_1, \ldots, P_n$, sodass

- für $i = 2, \ldots, n$: Prozess P_i hat Ressource R_{i-1} belegt, und P_1 hat Ressource R_n belegt
- für $i = 1, \ldots, n$: Prozess P_i wartet auf Ressource R_i

Sei R_j die kleinste Ressource aus $\{R_1, \ldots, R_n\}$ bezüglich der totalen Ordnung. Dann wartet Prozess P_j auf Ressource R_j, wobei er Ressource R_{j-1} (bzw. für $j = 1$ die Ressource R_n) schon belegt hat. Dies ist jedoch unmöglich, da Prozess P_j die Ressourcen in aufsteigender Reihenfolge bezüglich der totalen Ordnung belegen muss. Daher ergibt sich ein Widerspruch, und die Annahme war daher falsch. □

Wie bereits zu Beginn dieses Abschnitts erwähnt, kann mit diesem Verfahren auch das Überweisungsproblem von Bankkonten gelöst werden, indem z.B. die Prozesse die Bankkonten beginnend mit der kleinsten Kontonummer und anschließend aufsteigend belegen.

Man kann zeigen, dass ein Deadlock-freies System, in dem die Belegung von (allen) *einzelnen* Ressourcen Starvation-frei ist, auch insgesamt Starvation-frei ist. Damit lässt sich folgern: Wenn es eine totale Ordnung der Ressourcen gibt, die Ressourcen entsprechend dieser Ordnung belegt werden und die Belegung einzelner Ressourcen Starvation-frei ist, dann ist das Gesamtsystem Starvation-frei.

4.3 Deadlock-Vermeidung

Wie bereits erwähnt, geht es bei der Deadlock-Vermeidung darum, die Ressourcenvergabe von einem Algorithmus überwachen zu lassen, so dass eine mögliche Deadlock-Situation frühzeitig erkannt wird und deshalb im Vorhinein verhindert wird. Betrachte hierzu das so genannte Problem vom „Deadly Embrace“ (Tödliches Umarmen), welches von E.W.Dijstra formuliert und mit Hilfe des Bankier-Algorithmus gelöst werden kann.

Nehme an, es gibt ein Anzahl m von gleichen Ressourcen und jeder Prozess benötigt eine bestimmte maximale Anzahl der Ressourcen während seines Ablaufs. Dabei wird angenommen, dass diese maximale Anzahl im Vorhinein bekannt ist, die Prozesse die Ressourcen aber nach und nach anfordern. Ebenso wird angenommen, dass ein Prozess, sobald er alle benötigten Ressourcen erhalten hat, diese benutzt und anschließend wieder frei gibt. Die Aufgabe besteht nun darin einen Algorithmus (Prozess) dazwischen zu schalten, der für jede Anforderung eines Prozesses entscheidet, ob diese bedient wird, oder der Prozess warten soll.

Der Bankier-Algorithmus von Dijkstra trägt diesen Namen, da man das Problem auch anders formulieren kann: Es gibt einen Bankier, der einen Vorrat an Geld in seiner Bank hat. Die Kunden wollen einen maximalen Betrag an Geld vom Bankier geliehen bekommen. Sie fordern das Geld nach und nach an. Sobald sie ihren maximalen Betrag erhalten haben, zahlen sie ihn zurück. Die Aufgabe für den Bankier besteht darin nur dann an Kunden Geld auszuzahlen, wenn sichergestellt ist, dass er all sein Geld zurückerhalten wird.

Betrachte als Beispiel, dass der Bankier insgesamt 98 EUR hat, und es zwei Kunden gibt, die jeweils maximal 50 EUR benötigen und beide schon 48 EUR erhalten haben. Angenommen der erste und der zweite Kunde fordern nun nacheinander jeweils 1 EUR an. Bedient der Bankier

nun beide Kunden, so hat er kein Geld mehr, die Kunden jeweils 49 EUR und kein Kunde wird jemals seinen maximalen Betrag erhalten. Es ist ein Deadlock aufgetreten. Es gilt nun einen Algorithmus zu schreiben, der solche Situationen verhindert. Ein offensichtlicher Algorithmus wäre es, die Kunden nach einer totalen Ordnung zu bedienen: Erst wenn der kleinste Kunde vollständig bedient wurde, erhält der nächste Geld usw. Allerdings erzwingt diese Lösung die Sequentialisierung. D.h. das nebenläufige Problem wird durch eine sequentielle Lösung gelöst, was nicht im Sinne der Problemstellung ist. Deshalb gilt als zusätzliche Anforderung, dass der Algorithmus möglichst viel Nebenläufigkeit erlauben soll.

Genauer möchte man jede Anforderung bedienen solange sie nicht *zwingend* zu einem Deadlock führt. D.h. frei nach Dijkstra möchte man Anforderungen zulassen, solange sie nicht dazu führen, dass ein Prozess nur dann weiterrechnen kann, indem ein anderer getötet werden muss (daher der Name „Tödliche Umarmung“).

Im Folgenden wird eine erweiterte Variante des Bankieralgorithmus erläutert, der eine Anzahl von verschiedenen Ressourcen erlaubt. D.h. im Bild des Bankiers hat er verschiedene Währungen. Die aktuell (dem Bankier) zur Verfügung stehenden Ressourcen werden durch den Vektor $\overrightarrow{A}$ bezeichnet. Jede Komponente von $\overrightarrow{A}$ ist eine nicht-negative Ganzzahl, die die verfügbare Anzahl der entsprechenden Ressource angibt.

Sei $\mathcal{P}$ die Menge der Prozesse. Für $P \in \mathcal{P}$ sei:

- $\overrightarrow{M_P}$ der Vektor der maximal durch Prozess P anzufordernden Ressourcen
- $\overrightarrow{C_P}$ der Vektor der bereits an Prozess P vergebenen Ressourcen

Ein Zustand (mit all seinen Vektoren) ist *sicher*, wenn es eine Permutation π der Prozesse $P_1, \ldots, P_n \in \mathcal{P}$ gibt, sodass es für jeden Prozess P_i entsprechend der Reihenfolge der Permutation genügend Ressourcen gibt, wenn er dran ist. Genügend Ressourcen bedeutet hierbei, dass $\overrightarrow{A} + \left(\sum_{\pi(j)<\pi(i)} \overrightarrow{C_{P_{\pi j}}}\right) - \overrightarrow{M_{P_i}} + \overrightarrow{C_{P_i}} \geq \overrightarrow{0}$ gilt. D.h. zu den aktuell verfügbaren Ressourcen $\overrightarrow{A}$ dürfen zunächst die momentan vergebenen Ressourcen hinzu addiert werden, deren zugehörige Prozesse entsprechend der Permutation vor P_i vollständig bedient werden. Anschließend müssen die maximal durch Prozess P_i angeforderten Ressourcen gedeckt werden können, wobei die schon an P_i zugeteilten Ressourcen entsprechend gegen gerechnet werden können.

Der Bankiers-Algorithmus erhält als Eingabe eine Ressourcenanfrage $\overrightarrow{L_P}$ eines Prozesses $P \in \mathcal{P}$. Hierbei muss aufgrund der Annahmen gelten, dass $\overrightarrow{L_P} + \overrightarrow{C_P} \leq \overrightarrow{M_P}$ (sonst würde der Prozess mehr Ressourcen anfordern, als er vorher angegeben hat). Anschließend geht der Bankier-Algorithmus wie folgt vor:

- Wenn nicht genügend Ressourcen vorhanden sind, dann lehnt er sofort die Anfrage ab, d.h. Prozess P muss warten.
- Ansonsten passt er den aktuellen Zustand an die Anfrage an, d.h. er führt aus:

$$\begin{array}{rcl} \overrightarrow{A} & := & \overrightarrow{A} - \overrightarrow{L_P} \\ \overrightarrow{C_P} & := & \overrightarrow{C_P} + \overrightarrow{L_P} \end{array}$$

- Nun prüft er, ob der so erhaltene Zustand sicher ist (s.u.).
- Ist der Zustand sicher, so erhält P die angeforderten Ressourcen, andernfalls wird die Anfrage abgelehnt und P muss warten, der vorherige Zustand wird wieder hergestellt.

Erreicht ein Prozess sein Maximum und gibt die Ressourcen zurück, so wird der Vektor $\overrightarrow{A}$ (automatisch) angepasst.

Es bleibt den Algorithmus zu spezifizieren. Der Algorithmus ist in Abbildung 4.1 dargestellt.

```
function testeZustand(P, A⃗):
  if P = ∅ then
    return "sicher"
  else
    if ∃P ∈ P mit M⃗_P − C⃗_P ≤ A⃗ then
      A⃗ := A⃗ + C⃗_P;
      P := P \ {P};
      testeZustand(P, A⃗)
    else
      return "unsicher"
```

Abbildung 4.1: Bankiers-Algorithmus: Ist ein Zustand sicher?

Der Algorithmus sucht aus der Menge der Prozesse stets einen aus, dessen Ressourcenbedarf er komplett erfüllen kann. Da er dann davon ausgehen kann, dass dieser Prozess anschließend die Ressourcen zurück gibt, nimmt er diesen Prozess in die zu findende Permutationsfolge auf (d.h. er streicht ihn aus $\mathcal{P}$) und passt $\overrightarrow{A}$ an. Wenn alle Prozesse in die Folge einsortiert wurden (und daher $\mathcal{P} = \emptyset$ gilt), dann ist der Zustand sicher. Wenn jedoch Prozesse vorhanden sind, die nicht mehr einsortiert werden können, dann ist der Zustand unsicher, da man einen Deadlock herleiten kann.

Die Laufzeit des Algorithmus ist im worst-case $O(|\mathcal{P}|^2)$, da es maximal $|\mathcal{P}|$ rekursive Aufrufe gibt, und pro Durchlauf im schlimmsten Fall das Finden eines ausführbaren Prozesses (der $\exists$-Quantor) das Durchsuchen aller $|\mathcal{P}|$-Prozesse erfordert.

Das Interessante an dem Algorithmus ist, dass es genügt mit Laufzeit $O(|\mathcal{P}|^2)$ auszukommen, obwohl es $|\mathcal{P}|!$ verschiedene Permutationen gibt. Der Grund hierfür ist, dass wenn es einen Prozess gibt, der mit den aktuell zur Verfügung stehenden Ressourcen bedient werden kann, dann ist es egal, ob er direkt die Ressourcen bekommt oder später.

Ein kleiner Nachteil bei dieser Methode ist allerdings, dass nicht optimiert wird, d.h. es wird dabei nicht darauf geachtet wie gut (bzgl. der Nebenläufigkeit) die Reihenfolge der Ausführung ist. Ein weiterer Nachteil des kompletten Ansatzes besteht darin, dass die Maximalwerte $\overrightarrow{M_P}$ von Anfang an bekannt sein müssen, was in der Praxis oft nicht der Fall ist.

Betrachten abschließend ein Beispiel mit vier Ressourcen EUR, USD, JYN, SFR und vier Prozessen A,B,C,D. Der aktuelle Zustand sei durch die folgenden Vektor-Belegungen beschrieben:

Maximal-Werte	Erhaltene Werte	Verfügbare Ressourcen
$\overrightarrow{M_A} = (4,7,1,1)$	$\overrightarrow{C_A} = (1,1,0,0)$	$\overrightarrow{A} = (2,2,3,3)$
$\overrightarrow{M_B} = (0,8,1,5)$	$\overrightarrow{C_B} = (0,5,0,3)$	
$\overrightarrow{M_C} = (2,2,4,2)$	$\overrightarrow{C_C} = (0,2,1,0)$	
$\overrightarrow{M_D} = (2,0,0,2)$	$\overrightarrow{C_D} = (1,0,0,1)$	

Zunächst prüfen wir, ob dieser Zustand wirklich sicher ist: Beginne mit $\mathcal{P} = \{A,B,C,D\}$

- erster Aufruf von testeZustand mit $\mathcal{P} = \{A,B,C,D\}$ und $\overrightarrow{A} = (2,2,3,3)$

  - Da $\mathcal{P} \neq \emptyset$, prüfe, ob es einen Prozess $P \in \mathcal{P}$ gibt, der noch ausführbar ist:
  - A kann nicht ausgeführt werden, denn $\overrightarrow{M_A} - \overrightarrow{C_A} = (4,7,1,1) - (1,1,0,0) = (3,6,1,1) \not\leq (2,2,3,3) = \overrightarrow{A}$
  - B kann nicht ausgeführt werden, denn $\overrightarrow{M_B} - \overrightarrow{C_B} = (0,8,1,5) - (0,5,0,3) = (0,3,1,2) \not\leq (2,2,3,3) = \overrightarrow{A}$
  - C kann ausgeführt werden, denn $\overrightarrow{M_C} - \overrightarrow{C_C} = (2,2,4,2) - (0,2,1,0) = (2,0,3,2) \leq (2,2,3,3) = \overrightarrow{A}$ also passe $\overrightarrow{A}$ an (als ob C alle Ressourcen erhält und anschließend zurück gibt):

$$\begin{array}{lll} \overrightarrow{A} & := & \overrightarrow{A} + \overrightarrow{C_C} = (2,2,3,3) + (0,2,1,0) = (2,4,4,3) \\ \mathcal{P} & := & \mathcal{P} \setminus \{C\} = \{A,B,D\} \end{array}$$

  - rufe testeZustand rekursiv auf

- Aufruf von testeZustand mit $\mathcal{P} = \{A,B,D\}$ und $\overrightarrow{A} = (2,4,4,3)$
  - Da $\mathcal{P} \neq \emptyset$, prüfe, ob es einen Prozess $P \in \mathcal{P}$ gibt, der noch ausführbar ist:
  - A kann nicht ausgeführt werden, denn $\overrightarrow{M_A} - \overrightarrow{C_A} = (4,7,1,1) - (1,1,0,0) = (3,6,1,1) \not\leq (2,4,4,3) = \overrightarrow{A}$
  - B kann ausgeführt werden, denn $\overrightarrow{M_B} - \overrightarrow{C_B} = (0,8,1,5) - (0,5,0,3) = (0,3,1,2) \leq (2,4,4,3) = \overrightarrow{A}$ also passe $\overrightarrow{A}$ an (als ob B alle Ressourcen erhält und anschließend zurück gibt):

$$\begin{array}{lll} \overrightarrow{A} & := & \overrightarrow{A} + \overrightarrow{C_B} = (2,4,4,3) + (0,5,0,3) = (2,9,4,6) \\ \mathcal{P} & := & \mathcal{P} \setminus \{B\} = \{A,D\} \end{array}$$

  - rufe testeZustand rekursiv auf

- Aufruf von testeZustand mit $\mathcal{P} = \{A,D\}$ und $\overrightarrow{A} = (2,9,4,6)$
  - Da $\mathcal{P} \neq \emptyset$, prüfe, ob es einen Prozess $P \in \mathcal{P}$ gibt, der noch ausführbar ist:
  - A kann ausgeführt werden, denn $\overrightarrow{M_A} - \overrightarrow{C_A} = (4,7,1,1) - (1,1,0,0) = (3,6,1,1) \leq (2,9,4,6) = \overrightarrow{A}$ also passe $\overrightarrow{A}$ an (als ob A alle Ressourcen erhält und anschließend zurück gibt):

$$\begin{array}{lll} \overrightarrow{A} & := & \overrightarrow{A} + \overrightarrow{C_A} = (2,9,4,6) + (1,1,0,0) = (3,10,4,6) \\ \mathcal{P} & := & \mathcal{P} \setminus \{A\} = \{D\} \end{array}$$

  - rufe testeZustand rekursiv auf

- Aufruf von testeZustand mit $\mathcal{P} = \{D\}$ und $\overrightarrow{A} = (3,10,4,6)$
  - Da $\mathcal{P} \neq \emptyset$, prüfe, ob es einen Prozess $P \in \mathcal{P}$ gibt, der noch ausführbar ist:
  - D kann ausgeführt werden, denn $\overrightarrow{M_D} - \overrightarrow{C_D} = (2,0,0,2) - (1,0,0,1) = (1,0,0,1) \leq (3,10,4,6) = \overrightarrow{A}$ also passe $\overrightarrow{A}$ an (als ob D alle Ressourcen erhält und anschließend zurück gibt):

$$\begin{array}{lll} \overrightarrow{A} & := & \overrightarrow{A} + \overrightarrow{C_D} = (3,10,4,6) + (1,0,0,1) = (4,10,4,7) \\ \mathcal{P} & := & \mathcal{P} \setminus \{D\} = \emptyset \end{array}$$

  - rufe testeZustand rekursiv auf

- Aufruf von testeZustand mit $\mathcal{P} = \emptyset$ und $\overrightarrow{A} = (4, 10, 4, 7)$
  - Liefert "sicher", da $\mathcal{P} = \emptyset$.

Der Zustand ist somit sicher.

Nun wird die Anfrage $L_A = (2, 2, 0, 0)$ behandelt, d.h. Prozess A möchte zwei weitere EUR und zwei weitere USD belegen.
Nach Aktualisierung ($\overrightarrow{A} := \overrightarrow{A} - \overrightarrow{L_A}$ und $\overrightarrow{C_A} := \overrightarrow{C_A} + \overrightarrow{L_A}$) erhält man den Zustand:

Maximal-Werte	Erhaltene Werte	Verfügbare Ressourcen
$\overrightarrow{M_A} = (4, 7, 1, 1)$	$\overrightarrow{C_A} = (3, 3, 0, 0)$	$\overrightarrow{A} = (0, 0, 3, 3)$
$\overrightarrow{M_B} = (0, 8, 1, 5)$	$\overrightarrow{C_B} = (0, 5, 0, 3)$	
$\overrightarrow{M_C} = (2, 2, 4, 2)$	$\overrightarrow{C_C} = (0, 2, 1, 0)$	
$\overrightarrow{M_D} = (2, 0, 0, 2)$	$\overrightarrow{C_D} = (1, 0, 0, 1)$	

und es muss geprüft werden, ob dieser Zustand noch sicher wäre

- Aufruf von testeZustand mit $\mathcal{P} = \{A, B, C, D\}$ und $\overrightarrow{A} = (0, 0, 3, 3)$
  - Da $\mathcal{P} \neq \emptyset$, prüfe, ob es einen Prozess $P \in \mathcal{P}$ gibt, der noch ausführbar ist:
  - A kann nicht ausgeführt werden, denn $\overrightarrow{M_A} - \overrightarrow{C_A} = (4, 7, 1, 1) - (3, 3, 0, 0) = (1, 4, 1, 1) \not\leq (0, 0, 3, 3) = \overrightarrow{A}$
  - B kann nicht ausgeführt werden, denn $\overrightarrow{M_B} - \overrightarrow{C_B} = (0, 8, 1, 5) - (0, 5, 0, 3) = (0, 3, 1, 2) \not\leq (0, 0, 3, 3) = \overrightarrow{A}$
  - C kann nicht ausgeführt werden, denn $\overrightarrow{M_C} - \overrightarrow{C_C} = (2, 2, 4, 2) - (0, 2, 1, 0) = (2, 0, 3, 0) \not\leq (0, 0, 3, 3) = \overrightarrow{A}$
  - D kann nicht ausgeführt werden, denn $\overrightarrow{M_D} - \overrightarrow{C_D} = (2, 0, 0, 2) - (1, 0, 0, 1) = (1, 0, 0, 1) \not\leq (0, 0, 3, 3) = \overrightarrow{A}$
  - D.h. es gibt keinen Prozess in $\mathcal{P}$ mit $\overrightarrow{M_P} - \overrightarrow{C_P} \leq \overrightarrow{A}$ und der Algorithmus liefert „unsicher".

Die Anfrage sollte daher nicht ausgeführt werden.

4.4 Transactional Memory

In diesem Abschnitt wird ein relativ neuer Programmieransatz vorgestellt, der darauf abzielt, dass der Programmierer mehr oder weniger sequentiell programmieren kann, während das System dabei garantiert, dass keine Deadlocks usw. auftreten.

Zunächst kann man sich die Frage stellen, warum die vorhandenen Mechanismen und Programmierabstraktionen nicht ausreichend sind, bzw. wo deren Probleme zu sehen sind.

Betrachte erneut das Problem der Überweisung eines Betrages zwischen zwei Bankkonten. Eine Lösung des Problems war: Alle Überweisungsprozesse sperren die Konten entsprechend einer totalen Ordnung, führen die Überweisung aus und geben danach die Konten wieder frei. Angenommen das Programm soll erweitert werden, so dass nur dann eine Abbuchung durchgeführt wird, wenn das Konto gedeckt ist. Dann gibt es mehrere Möglichkeiten: Eine Möglichkeit besteht darin, die Transaktion in diesem Falle abzubrechen und es später erneut zu versuchen. Dann wird jedoch ständig gerechnet und solange keine Starvation-Freiheit garantiert ist, kann

es sein, dass es zwar Zustände gibt, in denen das Konto gedeckt ist, aber eine andere Transaktion „zufälligerweise“ vorher das Geld wieder weg bucht. Ein andere Möglichkeit wäre es, an einer Condition Variablen zu warten bis das Konto gedeckt ist. In diesem Fall muss man darauf achten, dass in der Zwischenzeit das Konto nicht gesperrt bleibt, denn schließlich müssen Zubuchungen erfolgen können. Dieses Beispiel zeigt, dass selbst in diesem kleinen Beispiel eine korrekt funktionierende Lösung schwer zu finden ist, und eine Reihe von Locks (Kontosperren, Condition Variablen, usw.) benötigt werden.

S.L. Peyton Jones nennt in (Peyton-Jones, 2007, Kapitel 24) einige Gründe, warum Lock-basierte Programmierung „schlecht“ ist:

- Setzen zu weniger Locks: Es kann schnell passieren, dass der Programmierer vergisst eine Sperre zu setzen und dadurch ein Speicherplatz ungeschützt durch zwei Prozesse modifiziert wird.
- Setzen zu vieler Locks: Werden zu viele Locks gesetzt, weil der Programmierer quasi übervorsichtig ist (bzw. das Problem nicht genau genug kennt), kann (im Bestfall) weniger Nebenläufigkeit vorhanden sein, als möglich wäre, d.h. der Programmierer erzwingt unnötige Sequentialisierung. Im schlimmsten Fall erzeugt der Programmierer einen Deadlock.
- Setzen der falschen Locks: In den Ansätzen der Lock-basierten Programmierung ist meist keine eindeutige Zuordnung zwischen Locks und den Daten, die durch sie geschützt werden sollen, vorhanden. Betrachte z.B. Condition Variablen: Hier „soll“ der Programmierer eine Bezeichnung der Variablen wählen, die der Bedingung entspricht, er muss es aber nicht. Diese Vorgehensweise führt dazu, dass der (erste) Programmierer weiß, warum er welche Locks setzt, aber weder der Compiler noch andere Programmierer wissen dies. Spätere Wartungsarbeiten am Code können leicht dazu führen, dass falsche Locks gesetzt bzw. nötige Locks nicht gesetzt werden.
- Setzen von Locks in der falschen Reihenfolge: Das Total-Order Theorem kann nur eingehalten werden, wenn jeder Programmierer weiß, wie die Ordnung der Locks aussieht und sie dementsprechend setzt.
- Fehlerbehandlung kann schwierig sein, da stets berücksichtigt werden muss, dass ein Fehler nicht dazuführt, dass Locks für immer gesetzt bleiben. Z.B. muss beim Abfangen von Fehlern, darauf geachtet werden, dass gesetzte Locks entfernt werden und ein konsistenter Zustand der Daten hergestellt wird.
- Vergessene **`signal`**-Operationen oder vergessenes erneutes Prüfen von Bedingungen führt zu fehlerhaften Systemen. Es ist leicht solche Programmierfehler zu machen.

Das wesentliche Gegenargument gegen die Lock-basierte Programmierung ist jedoch, dass die Programmierung nicht modular ist. D.h. im Allgemeinen ist es nicht möglich aus kleinen korrekten Programmen ein großes korrektes Programm zusammen zu setzen. Betrachte z.B. die Kontoabbuchung: Angenommen es soll eine neue Form der Überweisung eingeführt werden: Buche den Betrag von Konto A1 *oder* A2 auf Konto B, je nachdem welches Konto gedeckt ist. Hier funktioniert das Warten mit der Condition Variable an Konto A1 nicht länger. Das gesamte Programm muss umstrukturiert werden.

Die Idee des *Transactional Memory* beruht auf der Beobachtung, dass ähnliche Probleme wie bei der nebenläufigen Programmierung beim Zugriff auf Datenbanken bestehen. Allerdings ist in diesem Bereich der parallele Zugriff seit Jahren kein echtes Problem mehr (aus der Sicht des Anwenderprogrammierers, nicht des Implementierers der Datenbank), da dort der Datenbankmanager für konsistenten Zugriff auf die Datenbank sorgt. Beim Transactional Memory

ist die Grundidee nun, den gemeinsamen Speicher genau wie eine Datenbank zu behandeln. Die Zugriffe auf den Speicher sind wie Datenbanktransaktionen und ein Manager stellt sicher, dass der Speicherzustand konsistent bleibt. Aus Programmiersicht ist eine solche Lösung wünschenswert, denn er kann seine Anfragen schreiben, als würde er sequentiell programmieren. Er muss dabei keine Locks setzen, sondern übergibt die Anfrage dem Transaktionsmanager, der die Transaktion im Wesentlichen entweder durchführt, abbricht und neu startet oder abbricht und das Fehlschlagen zurück meldet.

Die Semantik von Datenbanktransaktionen lässt sich dadurch beschreiben, dass die Transaktion (Abfrage, Änderungen, usw.) so durchgeführt werden würden, als ob die Transaktion die einzige Aktion wäre, die auf der Datenbank geschieht. Der Datenbankmanager garantiert dabei, dass sich die Transaktion von außen beobachtet genauso verhält.

Für die Erörterung von Transactional Memory also Transaktionen auf dem Speicher und nicht auf der Datenbank werden zunächst Datenbanktransaktionen betrachtet und erläutert wie diese auf das Speichermodell übertragen werden können. Für jede Datenbanktransaktion müssen vier Eigenschaften erfüllt sein, die so genannten *ACID-Eigenschaften*:

- Atomicity: Alle Operationen, die innerhalb von Transaktionen durchgeführt werden sollen, werden entweder durchgeführt, oder *keine* Operation wird durchgeführt. D.h. es ist verboten, dass eine Operation fehlschlägt, aber die Transaktion als erfolgreich durchgeführt wird. Genauso ist es verboten, dass eine fehlgeschlagene Transaktion beobachtbare Unterschiede in der Datenbank hinterlässt. Eine erfolgreiche Transaktion *commits*, eine fehlgeschlagene Transaktion *aborts* (bricht ab),
- Consistency: Eine Transaktion verändert den Zustand der Datenbank. Diese Änderung muss konsistent sein. Eine abgebrochene Transaktion lässt aufgrund der atomicity-Bedingung den Zustand unverändert und damit konsistent. Konsistenz einer committed Transaktion hängt von der jeweiligen Anwendung ab, die die Konsistenz definiert (z.B. der Kontostand eines Bankkontos darf nicht beliebig groß negativ sein, oder ein neu hinzugefügtes Konto muss eine eindeutige Kontonummer erhalten, usw.).
- Isolation: Eine Transaktion liefert ein korrekt Resultat unabhängig davon wieviele weitere nebenläufige Transaktionen durchgeführt werden.
- Durability: Das Ergebnis einer committed Transaktion ist permanent. D.h. es wird auf der Festplatte oder ähnlichem dauerhaft gespeichert.

Die ersten drei Eigenschaften für Datenbanktransaktionen sind auch für Speichertransaktionen erforderlich. Durability kann i.a. für Speichertransaktionen nicht sichergestellt (und damit auch nicht gefordert) werden, da Hauptspeicher i.a. ein flüchtiger Speicher ist.

Es gibt jedoch einige Unterschiede zwischen dem Zugriff auf Datenbanken und dem Zustand auf den Speicher:

- Daten in einer Datenbank sind normalerweise auf Festplatten gespeichert. Da der Festplattenzugriff einige Millisekunden benötigt, welcher es auf der anderen Seite ermöglicht Millionen von CPU-Operationen auszuführen, kann man bei der Datenbankprogrammierung Rechenzeit und Datenzugriff gut gegeneinander verrechnen. Da Transactional Memory auf dem viel schnelleren Hauptspeicher operiert, können nicht viele Rechenoperationen während des Speicherzugriffs durchgeführt werden, d.h. eine Verrechnung ist nicht mehr so leicht möglich.
- Wie bereits erwähnt ist Transactional Memory nicht permanent, da nach der Programmterminierung die Daten im Hauptspeicher weg sind. Das macht die Implementierung von

Transactional Memory i.a. einfacher als bei Datenbanken.

- Transactional Memory muss in existierende Programmiersprachen integriert werden und darf keine völlig neue „Sprache“ darstellen, damit er von Programmierern benutzt und akzeptiert wird.

4.4.1 Basisprimitive für Transactional Memory

In diesem Abschnitt werden einige Operationen beschrieben, die als Basis für Transactional Memory gesehen werden können.

4.4.1.1 Atomare Blöcke

Mit dem Schlüsselwort **`atomic`** lassen sich Codeblöcke als Transaktionen kennzeichnen, die die erwähnten ACI-Eigenschaften erfüllen sollen.

```
atomic {
    Code der Transaktion
}
```

Wie die genaue Implementierung dahinter funktioniert, wird zunächst ignoriert. Gegenüber Monitoren ist der Vorteil, dass die Programmvariablen, die verändert werden sollen, nicht explizit angegeben werden müssen. Der Vorteil daran ist, dass alle Variablen geschützt sind und somit keine Variable vergessen werden kann.

Mit solchen als atomar gekennzeichneten Blöcken lassen sich trotz allem noch „falsche“ Programme schreiben, insbesondere wenn der Code innerhalb der Transaktion nicht terminieren kann. Dies wird auch nicht weiter eingeschränkt (dann hätte man die Turing-Mächtigkeit verloren), sondern die Semantik wird entsprechend angepasst: Die Ausführung einer Transaktion (atomaren Blocks) hat drei mögliche Ergebnisse:

- commit, wenn die Transaktion erfolgreich war,
- abort, wenn die Transaktion abgebrochen wurde
- undefiniert, wenn die Transaktion nicht terminiert

Beachte, dass Transaktionen im Transactional Memory-Modell auf zwei Arten abgebrochen werden können: Durch das System (den Transaktionsmanager), weil ein Konflikt auftritt, oder durch den Programmierer (der innerhalb der Transaktion eine Abbruchsmöglichkeit definiert). Geschieht der Abbruch durch das System, so wird der Transaktionsmanager erneut versuchen die Transaktion durchzuführen, es sei denn der Programmierer hat spezifiziert, was in diesem Fall passieren soll.

Im Unterschied zu Datenbanktransaktionen gibt es aber ein weiteres Problem: Auf die Variablen, die innerhalb einer Transaktion verändert werden, kann auch außerhalb von Transaktionen zugegriffen werden, und damit ungewolltes Verhalten provoziert werden, welches die ACI-Bedingungen verletzt. Aufgrund dieser Unterschiede wird die Semantik von **`atomic`** oft so spezifiziert, dass die Ausführung eines solchen Codeblockes so wäre, als ob alle **`atomic`**-Blöcke durch einen globalen Lock geschützt sind. Dies muss nicht bedeuten, dass die Implementierung es auch so macht, es wird nur das Verhalten spezifiziert. Wird außerhalb von Transaktionen

auf Variablen zugegriffen, so lässt diese Semantik es zu, dass unerwünschte Ergebnisse möglich sind. Es ist hierbei die Aufgabe des Programmierers, entsprechend alle Transaktionen als solche zu kennzeichnen.

4.4.1.2 Der abort-Befehl

Mithilfe der **`abort`**-Operation kann eine Transaktion abgebrochen werden. Der Ursprungszustand wird wieder hergestellt.

4.4.1.3 Der retry-Befehl

In den meisten Systemen zum Transactional Memory gibt es den **`retry`**-Befehl, der es ermöglicht, Transaktionen zu koordinieren. Wird **`retry`** ausgeführt, so wird die Transaktion abgebrochen (mit konsistenter Herstellung des Zustandes!) und erneut versucht die Transaktion von vorne an durchzuführen. Damit lässt sich ein ähnliches Verhalten wie bei Condition Variablen erzeugen. Betrachte als Beispiel die Programmierung eines Bounded Buffer beim Erzeuger / Verbraucher Problem. Hier muss sichergestellt werden, dass der Verbraucher nur dann auf den Puffer zugreift, wenn dieser nicht leer ist. Dies kann mit **`retry`** wie folgt programmiert werden:

```
atomic{
  if isEmpty(buffer) then retry;
  Lese erstes Element des Puffers usw.
}
```

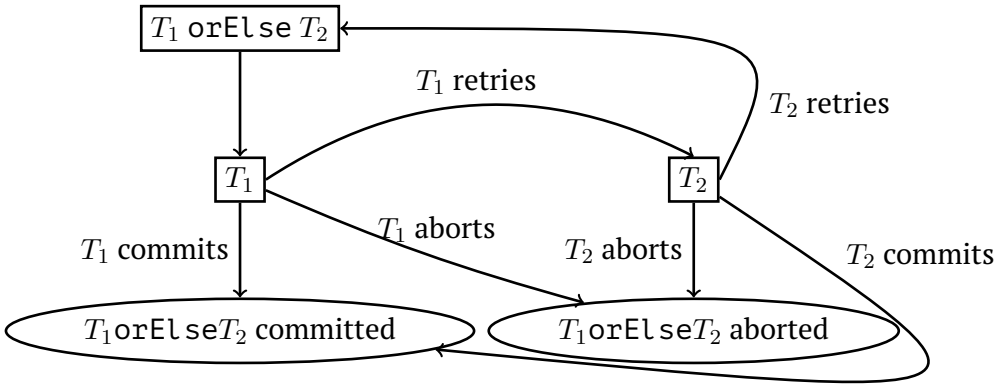
Wird die Transaktion durchgeführt, dann wird sie abgebrochen und neu gestartet, falls der Puffer leer ist.

4.4.2 Der orElse-Befehl

Mithilfe der **`orElse`**-Anweisung ist es möglich, eine Alternative zu spezifizieren, falls die Transaktion abgebrochen wird. Sie kombiniert zwei Transaktionen. Seien T_1 und T_2 Transaktionen, dann ist T_1**`orElse`**T_2 die Transaktion, die:

- zunächst T_1 durchführt. Falls T_1 committed oder T_1 explizit via Befehl abgebrochen wird, dann wird T_2 *nicht* ausgeführt. Die Transaktion T_1**`orElse`**T_2 committed oder wird abgebrochen, je nachdem was die Ausführung von T_1 macht.
- Falls T_1 durch den Transaktionsmanager abgebrochen wird oder **`retry`** ausgeführt wird, dann startet dieser T_1 nicht neu, sondern versucht T_2 durchzuführen.
- Falls T_2 explizit abgebrochen wurde oder erfolgreich committed, dann ist die gesamte Transaktion beendet.
- Falls T_2 ein **`retry`** durchführt (oder vom System abgebrochen wird), dann wird wieder mit T_1**`orElse`**T_2 gestartet.

Zusammengefasst kann man die Semantik von `orElse` durch das folgende Diagramm darstellen:



Mit dem **`orElse`**-Befehl ist es einfach möglich, das Abbuchen von Konto A1 oder Konto A2 auf Konto B zu implementieren:

```
atomic {
  {
    B := B + Betrag;
    A1 := A1 - Betrag;
    if A1 ≤ 0 then retry;
  }
  orElse
  {
    B := B + Betrag;
    A2 := A2 - Betrag;
    if A2 ≤ 0 then retry;
  }
}
```

4.4.3 Eigenschaften von TM Systemen

In diesem Abschnitt werden verschiedene Eigenschaften dargestellt, anhand derer sich bestehende Transactional Memory-Systeme klassifizieren und beschreiben lassen. Es gibt einige Software TM-Systeme, die also innerhalb verschiedener Programmiersprachen implementiert sind. Es gibt mittlerweile jedoch auch Hardware TM-Systeme, d.h. TM-Instruktionen werden auf Hardwareebene unterstützt

4.4.3.1 Weak / Strong Isolation

Weak Isolation meint Systeme, in denen nur Speicherplätze geschützt sind, wenn der Zugriff über Transaktionen erfolgt. Operationen außerhalb von Transaktionen können die Isolation-Bedingung verletzen und Inkonsistenzen herbei führen. Systeme, die Strong-Isolation unterstützen, sorgen automatisch dafür, dass durch Transaktionen geschützte Variablen automatisch immer geschützt bleiben, z.B. dadurch, dass **`atomic`**-Blöcke automatisch vom Compiler eingefügt werden.

4.4.3.2 Behandlung von geschachtelten Transaktionen

TM-Systeme unterscheiden sich in der Behandlung von geschachtelten **atomic**-Blöcken. Hierbei kann unterschiedliches Verhalten festgelegt werden, was passieren soll, wenn eine innere Transaktionen fehlschlägt. Man unterscheidet drei Vorgehensweisen: *geglättete* Transaktionen, *geschlossene* Transaktionen und *offene Transaktionen*.

Sind Transaktionen *geglättet*, so bedeutet dies, dass das Abbrechen einer inneren Transaktion zum Abbruch der gesamten Transaktion führt und umgekehrt, dass eine committed innere Transaktion erst dann sichtbar nach außen wird, wenn auch die äußere Transaktion erfolgreich ist.

Z.B. wird bei der folgenden Transaktion die Variable x den Wert 1 behalten:

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
    abort;
  }
}
```

Bei *geschlossenen* Transaktion führt eine innere abbrechende Transaktion nicht zum Abbruch der äußeren Transaktion. Für obiges Programm wird x den Wert 2 erhalten. Bei einer erfolgreichen Transaktion verhält sich das Programm genauso wie bei geglätteten Transaktionen. Insbesondere sind Änderungen durch innere Transaktionen erst nach außen sichtbar, wenn die äußere Transaktion committed ist.

Bei *offenen* Transaktionen werden innere erfolgreiche Transaktionen nach außen sichtbar. Selbst wenn die äußere Transaktion abbricht, bleiben die Änderungen der inneren Transaktion erhalten. Im folgenden Beispiel hätte x den Wert 3, obwohl die äußere Transaktion abbricht:

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
  }
  abort;
}
```

4.4.3.3 Granularität

TM-Systeme unterscheiden sich auch danach, nach welcher Größe Konflikte erkannt werden. Hierbei unterscheidet man in *Objektgranularität*, d.h. Konflikte aufgrund von Änderungen an Objekten werden beobachtet, und in *Wort- bzw. Blockgranularität*, wobei Änderungen sich direkt auf Speicherworte beziehen. Beachte, dass Objektgranularität gröber ist: Sobald zwei Transaktion das gleiche Objekt modifizieren, wird ein Konflikt erkannt und eine der Transaktionen wird abgebrochen. Dies geschieht auch dann, wenn die Transaktionen verschiedene Attribute des Objekts modifizieren.

4.4.3.4 Direktes und verzögertes Update

Wenn ein TM-System *direktes Update* benutzt, dann modifizieren ausführende Transaktionen die Objekte sofort, und das System sorgt dafür, dass nebenläufige Prozesse dies nicht gleichzeitig tun. Die alten Werten (vor dem Update) werden vom System gespeichert und im Falle eines Abbruchs zurück geschrieben. Bei *verzögertem Update* verhält es sich umgekehrt: Es werden nur lOkale Kopien modifiziert und erst beim commit wird das Original durch die lOkale Kopie überschrieben.

4.4.3.5 Zeitpunkt der Konflikterkennung

TM-Systeme unterscheiden sich darin, wann Konflikte erkannt werden. Sie können Konflikte bereits beim ersten Zugriff auf ein Objekt erkennen, zwischen drin, wenn sie einige Objekte verändert haben oder auch erst ganz am Ende, wenn die Transaktion committed werden soll.

4.4.3.6 Konfliktmanagement

Beim Auftreten von Konflikten müssen Transaktionen abgebrochen werden. Allerdings sollte sichergestellt werden, dass das Gesamtsystem einen Schritt macht (also nicht alle Transaktionen abgebrochen werden). Hierbei gibt es zahlreiche Strategien, welche Transaktionen durchgeführt werden.

4.4.4 Korrektheitskriterien für STM-Systeme

In mehreren Forschungarbeiten wird und wurde untersucht, welche Anforderungen und Eigenschaften an ein TM-System gestellt werden, um dieses als „korrekt“ zu bezeichnen. Ein ziemlich umfassender Überblick über 16 verschiedene Begriffe ist in (Dziuma et al., 2015) zu finden.

Wir erläutern an dieser Stelle eine Auswahl an Korrektheitskriterien und folgen dabei der Darstellung in (Guerraoui & Kapalka, 2008) mit Anpassungen aus (Dziuma et al., 2015).

Für eine formale Modellierung der STM-Systeme kann man die Transaktionsausführung als schrittweise Abarbeitung ansehen, wobei ein Schritt einem Aufruf einer Operation auf einem nebenläufigen Objekt (wie ein atomares Register, Compare-and-Swap-Objekt, etc.) und zusätzlich lOkalen Berechnungen entspricht. Wir beschränken unsere Ausführungen auf atomare Register. Um weiter zu abstrahieren, verwendet man Historien der Transaktionsausführung:

Definition 4.4.1 (Historie und Ereignisse)**.** *Eine Historie H ist eine* Folge von Ereignissen*, wobei ein Ereignis sein kann:*

- *Transaktion T_i führt einen Operationsaufruf durch. Dies kann eine Operation auf einen atomaren Register sein: T_i.read(x) (Lesen der transaktionalen Variablen x), T_i.write(x, v) (Beschreiben der transaktionalen Variablen x mit dem Wert v), oder T_i.commit (T_i möchte committen) und T_i.abort (T_i möchte abbrechen).*
- *Transakton T_i erhält eine Rückgabe für eine der vier Operationen read, write, commit, abort. Bei read ist dies das Ereignis $T_i.w$, wobei w der Wert ist, bei write ist dies das Ereignis T_i.Ok. Die Rückgabe kann allerdings immer auch der Spezialwert A_{T_i} sein (geschrieben als Ereignis*

$T_i.A_{T_i}$), der anzeigt, dass die Transaktion T abgebrochen wurde. Die Rückgabe von abort ist stets A_{T_i}, die Rückgabe von commit ist Ok oder A_{T_i}.

Betrachte z.B. ein TM-System, wobei die Werte von x, y, z am Anfang jeweils 1 seien und die beiden Transaktionen:

$$T_1 : \quad x := y + 1 \qquad\qquad T_2 : y := z + 1$$

Ereignisse (die auch Aufrufe sind) der Transaktion T_1 sind dann T_1.read(y), T_1.write(x) und T_1.commit. Rückgaben sind Ereignisse wie $T_1.1$ (der letzte Aufruf von T_1 liefert 1) oder T_1.Ok (der letzte Aufruf von T_1 wurde erfolgreich abgeschlossen). Auch $T_1.A_{T_1}$ ist möglich falls, die Transaktion T_1 abgebrochen wurde.
Eine mögliche Historie H_1 für obiges Szenario ist:

T_1.read(y)
T_2.read(z)
T_2.1
T_2.write(y,2)
T_1.1
T_2.Ok
T_1.write(x,2)
T_1.Ok
T_1.commit
T_1.Ok
T_2.commit
$T_2.A_{T_2}$

Man kann Historien auch entlang der Zeitachse als Intervalle darstellen, dabei sind Aufrufe und Rückgabe der Anfang und das Ende der Intervalle. Abbildung 4.2 zeigt eine entsprechende Darstellung.

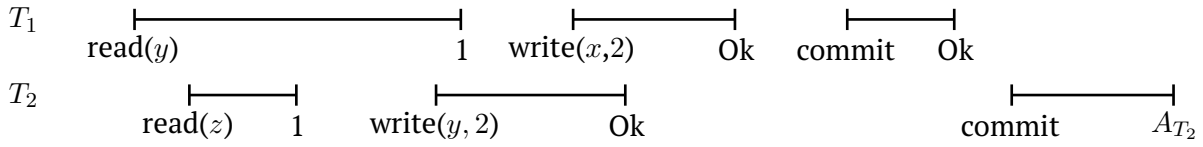


Abbildung 4.2: Intervalldarstellung einer Historie

Wir führen weitere Notation und Begriffe für Historien ein, bevor wir mit diesen die verschiedenen Korrektheitsbegriffe definieren.´

Definition 4.4.2 (Restriktionen einer Historie)**.** *Für eine Historie H und eine Transaktion T schreiben wir $H|T$ für die* Restriktion von H auf T*: Diese entsteht aus der Historie H, indem alle Ereignisse gelöscht werden, die nicht zu T gehören. Analog schreiben wir $H|\{T_1, \ldots, T_n\}$ für die Historie H nach Löschen aller Ereignisse, die nicht zu den Transaktionen $T_1, \ldots, T_n$ gehören. Für eine transaktionale Variable x, bezeichne $H|x$ die Historie H nach Löschen aller Ereignisse, die nicht zu x gehören.*

Z.B. ist $H_0|T_2$ (mit H_0 wie zuvor definiert):

T_2.read(z)
T_2.1
T_2.write(y,2)
T_2.Ok
T_2.commit
$T_2.A_{T_2}$

Da H_0 nur die Transaktionen T_1 und T_2 beinhaltet gilt $H_O|\{T_1, T_2\} = H_0$. Als weiteres Beispiel zeigen wir die Histore $H_0|y$:

T_1.read(y)
T_2.write(y,2)
T_1.1
T_2.Ok

Als nächstes definieren wir die Wohl-Geformtheit von Historien. Diese verlangt, dass die Historie pro Transaktion, die Ereignisse in einer zulässigen Reihenfolge enthält. Sie verlangt dabei nicht, dass jede Transaktion bis zum Ende gelaufen ist.

Definition 4.4.3. *Eine Historie H ist* wohl-geformt*, wenn für jede Transaktion T gilt: $H|T$ besteht abwechselnd aus Aufrufen und Rückgaben, $H|T$ enthält keinerlei Ereignisse nach dem Ereignis $T.A_T$, wenn $H|T$ das Ereignis T.commit enthält, dann folgt entweder kein Ereignis mehr oder entweder das Ereignis $T.A_T$ oder das Ereignis T.Ok, und wenn $H|T$ das Ereignis T.abort enthält, folgt danach entweder kein Ereignis mehr oder nur noch das Ereignis $T.A_T$.*

Wir betrachten im Weiteren nur wohl-geformte Historien. TM-Systeme die keine wohl-geformten Historien liefern werden daher als ungültig (nicht korrekt) angesehen.

Anhand der Historie können wir Transaktionen als erfolgreich beendet (committed), abgebrochen (aborted) oder als laufend klassifizieren:

Definition 4.4.4. *Eine Transaktion T ist* committed *in einer Historie H, wenn $H|T$ mit T.commit, T.Ok endet. Sie ist* aborted*, wenn $H|T$ mit $T.A_T$ endet. Eine Transaktion ist* vollständig *in einer Historie H, wenn sie committed oder aborted in H ist. Eine nicht-vollständige Transaktion nennen wir* laufend in H.

Wir schreiben $comm(H)$ für $H|\{T_1, \ldots, T_n\}$, wobei $T_1, \ldots, T_n$ alle Transaktionen in H sind, die committed sind, d.h. $comm(H)$ ist H nach Entfernen aller aborted und aller laufenden Transaktionen.

Definition 4.4.5 (Sequentielle Historie)**.** *Eine Historie ist* sequentiell*, wenn alle Ereignisse einer Transaktion hintereinander in einer Teilsequenz stehen.*

Beispiel 4.4.6. *Die Historie H_0 ist nicht sequentiell. Die Historie H_1, definiert als:*

T_1.*read*(y)
T_1.*1*
T_1.*write*(x,*2*)
T_1.*Ok*
T_1.*commit*
T_1.*Ok*
T_2.*read*(z)
T_2.*1*
T_2.*write*(y,*2*)
T_2.*Ok*
T_2.*commit*
$T_2.A_{T_2}$

ist sequentiell.

Definition 4.4.7 (Äquivalenz von Historien)**.** *Zwei Historien H, H' sind äquivalent (geschrieben als $H \sim H'$), wenn sie die gleichen Transaktionen verwenden und für jede Transaktion T gilt $H|T = H'|T$ (die Historien pro Transaktion sind identisch).*

Beispiel 4.4.8. *Betrachte die folgenden Historien*

Historie H_1:	*Historie H_2:*	*Historie H_3:*	*Historie H_4:*
T_1.*read*(x)	T_1.*read*(x)	T_1.*read*(x)	T_2.*read*(x)
T_2.*read*(x)	T_2.*read*(x)	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	T_1.*write*(x,v')	T_2.*read*(y)
$T_2.v$	$T_1.v$	T_1.*Ok*	$T_2.v''$
T_2.*read*(y)	T_1.*write*(x,v')	T_1.*commit*	T_2.*commit*
T_1.*write*(x,v')	T_1.*Ok*	T_1.*Ok*	T_2.*Ok*
T_1.*Ok*	T_1.*commit*	T_2.*read*(x)	T_1.*read*(x)
T_1.*commit*	T_2.*read*(y)	$T_2.v'$	$T_1.v$
$T_2.v''$	T_1.*Ok*	T_2.*read*(y)	T_1.*write*(x,v')
T_1.*Ok*	$T_2.v''$	$T_2.v''$	T_1.*Ok*
T_2.*commit*	T_2.*commit*	T_2.*commit*	T_1.*commit*
T_2.*Ok*	T_2.*Ok*	T_2.*Ok*	T_1.*Ok*

Dann sind die Historien H_3 und H_4 sequentiell, die Historien H_1 und H_2 sind äquivalent, H_1 und H_4 sind äquivalent, aber H_1 und H_3 sind nicht äquivalent.

Eine Historie impliziert eine partielle Ordnung der Transaktionen: Wenn der Zeitpunkt des Beendens (mit A_T oder Ok) von Transaktion T_i vor dem ersten Ereignis von Transaktion T_j liegt, dann liegt Transaktion T_i echt vor Transaktion T_j:

Definition 4.4.9 (Realzeit-Ordnung $\prec_H$)**.** *Für eine Historie H ist die* Realzeit-Ordnung *der Transaktionen $\prec_H$ die partielle Ordnung, die definiert ist durch: Wenn T_i vollständig in H ist und das erste Ereignis von T_j liegt nach dem letzten Ereignis von T_i in H, dann gilt $T_i \prec_H T_j$.*

Für sequentielle Historien H deren Transaktionen alle vollständig in H sind, ist die Realzeit-Ordnung stets eine totale Ordnung.

Definition 4.4.10 (Legale sequentielle Historien von committed Transaktionen)**.** *Eine sequentielle Historie H, wobei alle Transaktionen committed sind bis auf möglicherweise die letzte Transaktion ist* legal*, wenn für alle transaktionale Variablen gilt: $H|x$ stimmt überein mit der sequentiellen Beschreibung von atomaren Registern (die read- und write-Operationen und deren Rückgaben verhalten sich, alsob man sie in einem sequentiellen Programm ausgeführt hätte).*

Beispiel 4.4.11. *Die folgende Historie H_5 ist nicht legal, da $H_5|x$ nicht der sequentiellen Beschreibung von atomaren Registern entspricht. Die letzte read-Operation hätte 4 liefern müssen*

Historie H_5:	*Historie $H_5\|x$:*	*Historie $H_5\|y$:*
T_1.*read*(y)	T_1.*write*(x,4)	T_1.*read*(y)
T_1.3	T_1.*Ok*	T_1.3
T_1.*write*(x,4)	T_1.*read*(x)	T_2.*read*(y)
T_1.*Ok*	T_1.4	T_2.3
T_1.*read*(x)	T_2.*read*(x)	
T_1.4	T_2.2	
T_1.*commit*		
T_1.*Ok*		
T_2.*read*(x)		
T_2.2		
T_2.*read*(y)		
T_2.3		

Für eine sequentielle Historie H (deren Transaktionen allesamt vollständig in H sind) kann der obige Begriff der Legalität nicht direkt verwendet werden, da nach aborted Transaktionen sich die read- und write-Operationen anders verhalten können, als in einem sequentiellen Programm. Um dennoch einen Begriff der Legalität für solche Historien zu definieren, werden alle Transaktionen einzeln betrachtet und für jede Transaktion T nur die Ereignisse jener Transaktionen T' in der Historie betrachtet, die vor dem Beginn von T committed sind:

Definition 4.4.12 (Legale vollständige, sequentielle Historien)**.** *Eine sequentielle Historie H, deren Transaktionen allesamt vollständig in H sind, ist* legal*, wenn für jede Transaktion T_i in H die Historie H_i legal ist, wobei H_i aus H entsteht, indem genau jene Ereignisse $T_j.e$ aus H in H_i übernommen werden mit*

- *$j = i$, oder*
- *T_j ist committed in H und $T_j \prec T_i$ in H.*

Beachte, dass die Historie H_i die Anforderungen an den zuvor definierten Begriff der Legalität erfüllt: Alle Transaktionen außer evtl. die Transaktion T_i sind in H_i committed.

Beispiel 4.4.13. *Betrachte die Historie H_6 und die Historien $H_{6,1}$, $H_{6,2}$ und $H_{6,3}$:*

Historie H_6	*Historie* $H_{6,1}$	*Historie* $H_{6,2}$	*Historie* $H_{6,3}$
T_1.*write*$(x, 3)$	T_1.*write*$(x, 3)$	T_3.*read*(x)	T_3.*read*(x)
T_1.*Ok*	T_1.*Ok*	$T_3.4$	$T_3.4$
T_1.*read*(x)	T_1.*read*(x)	T_3.*write*$(y, 2)$	T_3.*write*$(y, 2)$
$T_1.3$	$T_1.3$	T_3.*Ok*	T_3.*Ok*
T_1.*abort*	T_1.*abort*	T_3.*commit*	T_3.*commit*
$T_1.A_{T_1}$	$T_1.A_{T_1}f$	T_3.*Ok*	T_3.*Ok*
T_3.*read*(x)		T_2.*read*(y)	
$T_3.4$		$T_2.2$	
T_3.*write*$(y, 2)$		T_2.*read*(x)	
T_3.*Ok*		$T_2.4$	
T_3.*commit*			
T_3.*Ok*			
T_2.*read*(y)			
$T_2.2$			
T_2.*read*(x)			
$T_2.4$			

Da $H_{6,1}$, $H_{6,2}$ und $H_{6,3}$ legale sequentielle Historien von committed Transaktionen sind, ist H_6 eine legale vollständige, sequentielle Historie.

Als wichtigste Eigenschaften eines korrekten STM-Systems sind zu nennen:

- Alle Transaktionen, die committed sind, erscheinen, als ob sie unteilbar in einem Schritt durchgeführt wurden.
- Abgebrochene Transaktionen erscheinen, als ob sie gar nicht durchgeführt wurden.

Zwei weitere Aspekte, die man üblicherweise fordert, sind:

Erhaltung der Realzeit-Ordnung: Der Zeitpunkt zudem die Effekte einer Transaktion erscheinen, liegt irgendwo in der Laufzeit der Transaktion. D.h. die Transaktion selbst sollte keine veralteten Speicherzustände lesen. Dieser Effekt kann bei falschen Implementierungen tatsächlich z.B. dadurch auftreten, dass durch die Verwendung von Caches noch alte Werte der transaktionalen Variablen gespeichert sind. Dies ist zu vermeiden. Vielmehr sollte gelten, dass wenn eine Transaktion T_1 ein Objekt x modifiziert und committed, und danach eine Transaktion T_2 startet und x liest, dann sollte T_2 den von T_1 geschriebenen Wert und nicht einen älteren Wert lesen.

Formal definiert man für Historien: Eine Historie H' *erhält die Realzeitordnung* einer Historie H, wenn gilt $\prec_H \subseteq \prec_{H'}$.

Ausschluss inkonsistener Ansichten: Man könnte die Meinung vertreten, dass laufende Transaktionen, die weder committed sind noch abgebrochen sind, auch inkonsistente Speicherzustände lesen dürfen, da sie später sowieso einfach abgebrochen werden können. Allerdings kann dies zu verschiedenen Problemen führen, da Transaktionen nicht in einer völlig abgekapselten Umgebung laufen: Seien x und y zwei (transaktionale) Variablen, welche die Invariante $y = x^2$ und $x \geq 2$ stets erfüllen sollen. Der Programmierer erfüllt die Invariante, indem darauf achtet, dass alle Transaktionen sie erfüllen. Sei $x = 4$ und $y = 16$ und betrachte nun die Transaktionen

$$\begin{array}{lll} T_1: & x := 2; & T_2 \quad z := 1/(y - x) \\ & y := 4; & \end{array}$$

Nehme an, dass T_2 zuerst x liest, dann T_1 durchläuft und dann T_2 weiterrechnet und die aktualisierten Werte dabei sehen darf. Als Historie geschrieben

T_2.read(x)
T_2.4
T_1.write($x, 2$)
T_1.Ok
T_1.write($y, 4$)
T_1.Ok
T_1.commit
T_1.Ok
T_2.read(y)
T_2.4
...

In diesem Fall stürzt T_2 mit einem Fehler (Division durch 0) ab, aus dem Grund, dass inkonsistente Werte gelesen wurden. Daher sollte T_2 entweder die als Rückgabe $T_2.16$ erhalten, oder $T_2.A_{T_2}$ um direkt abzubrechen.

Wir erläutern nun die Korrektheitsbegriffe:

Definition 4.4.14 (Linearisierbarkeit)**.** *Ein STM-System ist erfüllt die Eigenschaft* Linearisierbarkeit*, wenn gilt: Jede committed Transaktion wirkt wie ein atomarer Funktionsaufruf auf den transaktionalen Variablen, der in einem Schritt ausgeführt wird. Die erfolgreichen Transaktionen wirken wie eine Sequenz solcher Schritte.*

Eng verwandt ist der Begriff der Sequentialisierbarkeit:

Definition 4.4.15 (Sequentialisierbarkeit)**.** *Ein STM-System ist* sequentialisierbar*, wenn für jede Historie H des TM-Systems gilt: Es gibt eine sequentielle und legale Historie S mit $S \sim comm(H)$. D.h. die Historie der erfolgreichen Transaktionen ist äquivalent zur Historie eines sequentiellen Ablaufs der Transaktionen.*

Die strikte Sequentialisierbarkeit fordert zusätzlich, dass die gesuchte sequentielle Historie die Realzeitordnung der gegebenen Historie erhält:

Definition 4.4.16 (Strikte Sequentialisierbarkeit)**.** *Ein STM-System ist* strikt sequentialisierbar*, wenn für jede Historie H des TM-Systems gilt: Es gibt eine sequentielle und legale Historie S mit $S \sim comm(H)$ und $\prec_{comm(H)} \subseteq \prec_S$.*

Bisher gingen wir davon aus (wie es die Sequentialisierbarkeit in der Literatur auch tut), dass es nur Lese- und Schreibe-Operationen gibt. Der Ansatz der sogenannten „globalen Atomarität“ erlaubt auch anderere nebenläufige Objekte und Kopien dieser Objekte. Der Ansatz von (Guerraoui & Kapalka, 2008) erlaubt ebenfalls beliebige nebenläufige Objekte mit sequentieller Beschreibung. Wir erweitern unseren Formalismus an dieser Stelle nicht und verweisen auf die Literatur.

Ein Kritikpunkt der bisherigen Begriffe ist, dass sie fast keine Aussagen über abgebrochene Transaktionen oder noch laufende Transaktionen machen.

Die *Recoverability* fordert: Wenn eine Transaktion T_i eine transaktionale Variable beschreibt, dann darf keine andere Transaktion T_j die selbe transaktionale Variable lesen, bevor T_i erfolgreich oder abgebrochen ist. Diese Forderung ist einerseits zu stark, da sie manchmal zuviel Sequentialisierung der Transaktionsausführung erzwingt und andererseits zu schwach, da sie (selbst in Kombination mit globaler Atomarität) für abbrechenede Transaktionen nicht verbietet, inkonsistente Zustände zu lesen (siehe (Guerraoui & Kapalka, 2008)).

In (Guerraoui & Kapalka, 2008) wird daher als Korrektheitskriterium die sogenannte Opazität vorgeschlagen: Diese verlangt eine äquivalente sequentielle und legale Historie zur gegeben Historie, die jedoch vervollständigt wird, so dass laufende Transaktionen abgebrochen werden, und Transaktionen die ein commit ausgeführt haben, aber noch nicht committed sind entweder committed oder abgebrochen werden. Die Vervollständigung ist dabei nicht eindeutig, es genügt, wenn eine Vervollständigung die geforderten Eigenschaften hat.

Definition 4.4.17. *Ein TM-System erfüllt die* Opazität *(opacity), wenn jede Historie des TM-Systems opak ist:*

Eine History H ist opak*, wenn es eine vollständige, sequentielle, legale Historie S gibt, die äquivalent zu einer History in $complete(H)$ ist, so dass S die Realzeitordnung von H respektiert.*

Dabei liefert $complete(H)$ eine Menge von Historien H', so dass alle Transaktionen die in H vorkommen in H' vollständig sind, wobei

- *H' wohl-geformt ist,*
- *H' enthält gegenüber H nur zusätzliche commit und abort-Ereignisse, und zwar nur für Transaktionen, die laufend sind in H.*
- *laufende Transaktionen T aus H, für die T.commit in H vorkommt sind committed oder aborted in H'*
- *laufende Transaktionen T aus H, für die kein T.commit in H vorkommt sind aborted in H'.*

Dieser Korrektheitsbregriff impliziert, dass die Operationen jeder erfolgreichen Transaktion wie atomar in einem Schritt durchgeführt werden. D.h. insbesondere: Entfernt man alle Schritte von abgebrochenen oder laufenden Transaktionen, so ist die (nebenläufige) Ausführung der erfolgreichen Transaktionen äquivalent zu einer sequentiellen Ausführung dieser Transaktionen. Zusätzlich muss gelten, dass die sequentielle Ausführung die Realzeit-Ordnung einhält. Effekte von abgebrochenen Transaktionen sind niemals sichtbar für andere Transaktionen. Jede Transaktion (egal ob erfolgreich oder abgebrochen) sieht nur konsistente Zustände, d.h. solche die von committeten Transaktionen entstanden sind.

Weitere Begriffe und Begründungen für diesen Begriff der Korrektheit sind in der entsprechenden Literatur zu finden. Der im nächsten Abschnitt vorgestellte TL2-Algorithmus erfüllt Opazität.

4.4.5 Der TL2-Algorithmus

In diesem Abschnitt wird der sogenannte „Transactional Locking“-Algorithmus von Dice, Shalev und Shavit (Dice et al., 2006) vorgestellt, wobei wir größtenteils der Darstellung aus (Guerraoui & Kapalka, 2010) folgen. Dieser Algorithmus implementiert den Transaktionsmanager und wird oft in Implementierungen von STM-Systemen verwendet. Die **`retry`**- und die **`orElse`**-Operation und auch verschachtelte Transaktionen sind erstmal nicht in diesem Algorithmus vorgesehen.

Der TL2-Algorithmus verwendet einen globalen Zähler `gc`, der beim committen von Transaktionen erhöht wird. Dieser Zähler wird durch ein fetch-and-increment Objekt implementiert.

Die Transaktionalen Variablen $x_1, x_2, \ldots$ (d.h.) die echten gemeinsamen Speicherplätze seien in einem Feld TVar und jedem Speicherplatz TVar[i] ist ein Compare-and-Swap-Objekt C[i] zugeordnet, welches ein Paar (ver,lock) enthält, wobei

- Die Komponente ver ist ein Zeitstempel (des globalen Zählers), der den letzten Schreibzugriff auf die Speicherstelle vermerkt.
- Die Komponente lock ist ein Wahrheitswert, der anzeigt, ob der Speicherplatz als gesperrt gilt.

Die lokalen Datenstrukturen jeder Transaktion sind

- Eine Menge readset, in der sich die Adressen der von der Transaktion gelesenen Speicherplätze gemerkt wird.
- Ein Menge writeset, in der sich die Adressen der von Transaktion geschriebenen Adressen gemerkt wird.
- Lokale Speicherplätze writelog[i], sodass writelog[i] den von der Transaktion geschriebenen Wert für TVar[i] enthält. Alle Schreiboperationen der Transaktionen werden in diesen lOkalen Speicherplätzen festgehalten und zunächst nicht auf dem echten Speicher durchgeführt. Die neuen Inhalte werden erst beim (erfolgreichen) Committen in den gemeinsamen Speicher geschrieben.
- Ein Zeitstempel readver, der den Wert des globalen Zählers bei dem ersten Lesezugriff der Transaktion erhält
- Eine Menge lockset, in der sich die Adressen der von der Transaktion aktuell gesperrten Speicherplätze gemerkt wird

Die Operationen des TL2-Algorithmus zum Lesen, zum Schreiben einer Transaktionalen Variablen, sowie zum Abbrechen und zum Committen sind in Abbildung 4.3 dargestellt. Wir erläutern die einzelnen Operationen.

Ausführen einer Lese-Operation für TVar x_m Die erste Leseoperation setzt readver auf den aktuellen Wert des globalen Zählers.

Zunächst überprüft wird überprüft, ob die Adresse der transaktionale Variablen schon bereits in der Menge writeset ist (also schon von der Transaktion beschrieben wurde). Ist das der Fall wird der neue Wert aus dem Feld writelog zurück gegeben.

Ist m nicht in der Menge writeset, so wird zunächst das Paar (ver_1,locked_1) aus dem zugehörigen Compare-and-Swap-Objekt, anschließend der Inhalt von x_m aus TVar[m], und schließlich erneut das Compare-and-Swap-Objekt gelesen (dies ergebe (ver_2,locked_2)

In folgenden Fällen bricht die Transaktion komplett ab und startet neu, da ein Konflikt festgestellt wurde:

- Der Speicherplatz ist gesperrt, da locked_2 auf `True` gesetzt ist.
- $\text{ver}_2 >$ `readver`, denn dann wurde x_m nach dem Start der Transaktion verändert.
- $\text{ver}_1 \neq \text{ver}_2$, denn dann war das Lesen des Wertes von TVar[m] nicht atomar, d.h. eine andere Transaktion hat TVar[m] währenddessen geändert.

Wurde kein Konflikt festgestellt, so wird die Adresse m in die Menge readset eingefügt und der gelesene Wert ist die Rückgabe der Leseoperation.

Gemeinsame Datenstrukturen:

gc	(global counter), fetch-and-increment-Objekt, initial gc=0
TVar[1..]	Feld von atomaren Registern, initial TVar[i] = 0
C[1..]	Feld von Compare-and-Swap-Objekten mit Paaren als Inhalt, initial C[i]=(0,false)

Lokale Datenstrukturen:

readver	lokaler Zählerstand, initial readver$= \perp$
readset, writeset, lockset	Mengen von Adressen, initial alle $= \emptyset$
writelog[1..]	Feld von Werten

Operation T_k.read(x_m)
if readver = $\perp$ **then** readver := read(gc);
if $m \in$writeset **then return** writelog[m];
(ver$_1$,locked$_1$) := read(C[m]);
result := read(TVar[m]);
(ver$_2$,locked$_2$) := read(C[m]);
if ver$_1 \neq$ ver$_2$ **or** locked$_2$ **or** ver$_2 >$ readver
then T_k.abort();
readset := readset $\cup\{m\}$;
return result;

Operation T_k.write(x_m, val)
writeset := writeset $\cup\{m\}$;
writelog[m] := val;
return Ok;

Operation T_k.abort()
for all $m \in$ lockset **do**
(ver,locked) := read(C[m]);
write(C[m],(ver,`False`));
readset := $\emptyset$;
writeset := $\emptyset$;
lockset := $\emptyset$;
readver := $\perp$;
return A_{T_k};

Operation T_k.commit()
for all $m \in$ writeset **do**
(ver,locked) := read(C[m]);
if locked **then** T_k.abort();
lock := compare-and-swap(
C[m],(ver,`False`),(ver,`True`));
if not lock **then** T_k.abort();
lockset := lockset $\cup\{m\}$;
writever := 1+fetch-and-increment(gc);
if writever $\neq$ readver+1 **then**
for all $m \in$ readset **do**
(ver,locked) := read(C[m]);
if ver $>$ readver
or (locked **and** $m \notin$ writeset)
then T_k.abort();
for all $m \in$ writeset **do**
write(TVar[m],writelog[m]);
write(C[m],(writever,`False`))
readset := $\emptyset$;
writeset := $\emptyset$;
lockset := $\emptyset$;
readver := $\perp$;
return Ok;

Abbildung 4.3: Der TL2-Algorithmus

Ausführen einer Schreib-Operation für TVar x_m Die Schreiboperation findet nur in der Menge writeset und im Feld writelog statt, d.h. der neue Wert wird in writelog[m] gespeichert (bzw. überschreibt er einen vorherigen Wert) und m wird der Menge writeset hinzugefügt.

Commit-Phase Hat eine Transaktion alle STM-Befehle abgearbeitet, so beginnt die Commit-Phase. Es werden die folgenden Schritte durchgeführt:

- Sperren aller Speicherplätze aus der Menge writeset: Die Transaktion setzt die Wahrheitswerte der compare-and-swap-Objekte auf `True`, um diese zu sperren. Erhält die Transaktion nicht alle Sperren, so bricht sie ab, gibt alle Sperren frei, die sie hält und startet neu. Für die Implementierung dieses Schrittes muss sich die Transaktion merken, welche Speicherplätze bereits gesperrt wurden (dafür wird lockset verwendet) Für das Sperren der Speicherplätze empfiehlt es sich diese in einer totalen Ordnung zu sperren, um Deadlocks zu vermeiden.
- Erhöhen des globalen Zählers. Der neue Wert des Zählers wird in der lOkalen Variablen `wv` abgespeichert.
- Validiieren der Menge readset: In der Menge readset sind die Adressen der gelesenen Speicherplätze. Für jede dieser Adressen wird der aktuelle Zeitstempel ver dahingehend überprüft, ob readver $\leq$ ver gilt, d.h. ob zwischenzeitlich das Objekt beschrieben wurden. Schlägt einer dieser Tests fehl, oder ist einer der Speicherplätze gesperrt, so wird die Transaktion abgebrochen und neu gestartet. Die Verifikation der Menge readset kann entfallen, wenn writever=readver+1 gilt, denn dann hat kein andere Transaktion zwischen dem ersten Lesen und der commit-Phase die transaktionalen Variablen beschrieben.
- Schreiben der zur Menge writeset zugehörigen Einträge in den gemeinsamen Speicher: Die neuen Werte aus dem Feld writelog werden in die entsprechenden Speicherzellen geschrieben.
- Entsperren und Setzen der Zeitstempel: Die gesperrten Speicherplätze werden entsperrt und dabei wird der Wert writever als neuer Zeitstempel gesetzt.

Durch Prüfen der Menge readset und Sperren der zur Menge writeset zugehörigen Compare-and-Swap-Objekte wird sichergestellt, dass der Effekt der Transaktion wie eine atomar ausgeführte Funktion von gelesenen Speicherplätzen auf geschriebene Speicherplätze wirkt.

Abbruch einer Transaktion Beim Abbruch werden alle lOkalen Datenstrukturen neu initialisiert. Findet der Abbruch während des commits statt, werden mithilfe der Menge lockset, die gesperrten Speicherplätze wieder entsperrt.

Eine Besonderheit des TL2-Algorithmus ist, dass wenige Sperren verwendet werden. So wird die Sperre der Speicherplätze nur beim Committen gesetzt aber z.B. nicht beim Lesen des aktuellen Wertes.

Dem Algorithmus fehlt noch das Erzeugen neuer transaktionaler Speicherplätze, was jedoch leicht durch Hinzufügen eines weiteren Menge von neuen Speicherplätzen, die erst zur Commit-Zeit in den gemeinsamen Speicher geschrieben werden, bewerkstelligt werden kann.

Eine weitere Optimierung des Algorithmus kann durchgeführt werden, indem Transaktionen, die ausschließlich lesen, anders behandelt werden: Diese können auf die Verwaltung der Menge writeset aber auch der Menge readset verzichten. Sie validiieren die Gültigkeit der Lesezugriff nur durch den Zeitstempel readver und den Vergleich von readver mit ver_2 bei jedem

Lesezugriff. Führen diese zu keinem Abbruch, so ist sichergestellt, dass die Transaktion eine konsistente Sicht der transaktionalen Variablen gelesen hat.

Ob eine Transaktion nur liest kann auf verschiedene Weisen festgestellt werden:

- Der Programmierer gibt dies durch eine Annotation an.
- Eine Programmanalyse stellt dies fest.
- Jede Transaktion wird zunächst als nur lesende Transaktion ausgeführt. Sobald sie jedoch eine Schreibaktion durchführen will, wird sie neu als lesende und schreibende Transaktion gestartet.

Semantisch hat der TL2-Algorithmus eine Schwäche: Nichtterminierende Transaktionen werden nicht abgebrochen, auch dann, wenn die Nichtterminierung nur bedingt auftritt. Betrachte z.B. die beiden Transaktionen, die auf den transaktionalen Speicherplatz `x` zugreifen:

Transaktion 1	Transaktion 2
if x **then**	x := False;
loop forever	
else return	

Wenn `x` mit `True` belegt ist und Transaktion 1 zuerst ausgeführt wird, und in die Endlosschleife läuft, und Transaktion 2 währenddessen ausgeführt wird, sollte Transaktion 1 neu gestartet werden, denn der gelesene Wert für `x` ist nicht mehr aktuell. Der TL2-Algorithmus stellt dies jedoch nicht fest. Eine semantische korrekte Behandlung würde erfordern, dass die Menge readset in Zeitabständen auf Validität untersucht wird und die Transaktion bei nicht valider Menge readset abgebrochen wird.

Im Kapitel zur Programmiersprache Haskell werden noch weitere Algorithmen für den Transaktionsmanager betrachtet, insbesondere der im Standard-Haskell-Compiler GHC verwendete Algorithmus.

4.4.6 Fazit

Als Fazit sollte man sich merken, dass Transactional Memory sich noch in der Entwicklung befindet, aber doch oft verwendet wird. Ein großes Problem der TM-Systeme ist, dass nur solche Operationen innerhalb von Transaktionen verwendet werden sollten, die auch rückgängig gemacht werden können. Z.B. wird bei einem System mit direktem Update ein `print „Hallo“`-Befehl sofort ausgeführt werden und „Hallo“ auf dem Bildschirm erscheinen. Dieser Befehl ist schlecht rückgängig zu machen (Bildschirm löschen?).

4.5 Quellennachweis

Abschnitte 4.1 – 4.3 stammen im Wesentlichen aus (Taubenfeld, 2006), Abschnitt 4.4 über Transactional Memory orientiert sich an (Larus & Rajwar, 2006; Larus & Rajwar, 2010) und (Peyton-Jones, 2007, Kapitel 24). Die Darstellung der Korrektheitskriterien aus Abschnitt 4.4.4 ist eine Zusammenfassung der Darstellung aus (Guerraoui & Kapalka, 2008). Weitere Kriterien und ein umfassender Überblick dazu ist in (Dziuma et al., 2015). Der TL2-Algorithmus wurde in (Dice et al., 2006) beschrieben.

5

Nebenläufigkeit in der Programmiersprache Haskell

In diesem Kapitel betrachten wir die funktionale Programmiersprache Haskell (Peyton Jones, 2003) und insbesondere deren Möglichkeiten zur nebenläufigen Programmierung. Wir verzichten an dieser Stelle auf eine Einführung in Haskell. Diese kann z.B. in (Bird, 1998; O'Sullivan et al., 2008) gefunden werden. Bevor wir auf das nebenläufige Programmieren in Haskell eingehen, beschäftigen wir uns mit Ein- und Ausgabe in Haskell.

5.1 I/O in Haskell

In einer rein funktionalen Programmiersprache mit verzögerter Auswertung wie Haskell sind Seiteneffekte zunächst verboten. Fügt man Seiteneffekte einfach hinzu (z.B. durch eine „Funktion" `getZahl`), die beim Aufruf eine Zahl vom Benutzer abfragt und anschließend mit dieser Zahl weiter auswertet, so erhält man einige unerwünschte Effekte der Sprache, die man im Allgemeinen nicht haben möchte.

- Rein funktionale Programmiersprachen sind *referentiell transparent*, d.h. eine Funktion angewendet auf gleiche Werte, ergibt stets denselben Wert im Ergebnis. Die referentielle Transparenz wird durch eine Funktion wie `getZahl` verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.
- Ein weiteres Gegenargument gegen das Einführen von primitiven Ein-/ Ausgabefunktionen besteht darin, dass übliche (schöne) mathematische Gleichheiten wie $e + e = 2 * e$ für alle Ausdrücke der Programmiersprache nicht mehr gelten. Setze `getZahl` für e ein, dann fragt $e * e$ zwei verschiedene Werte vom Benutzer ab, während $2 * e$ den Benutzer nur einmal fragt. Würde man also solche Operationen zulassen, so könnte man beim Transformieren innerhalb eines Compilers übliche mathematische Gesetze nur mit Vorsicht anwenden.
- Durch die Einführung von direkten I/O-Aufrufen besteht die Gefahr, dass der Programmierer ein anderes Verhalten vermutet, als sein Programm wirklich hat. Der Programmierer muss die verzögerte Auswertung von Haskell beachten. Betrachte den Ausdruck `length [getZahl, getZahl]`, wobei `length` die Länge einer Liste berechnet als

  ```
  length [] = 0
  length (_:xs) = 1 + length xs
  ```

  Da die Auswertung von `length` die Listenelemente gar nicht anfasst, würde obiger Aufruf, keine `getZahl`-Aufrufe ausführen.
- In reinen funktionalen Programmiersprachen wird oft auf die Festlegung einer genauen Auswertungsreihenfolge verzichtet, um Optimierungen und auch Parallelisierung von Programmen durchzuführen. Z.B. könnte ein Compiler bei der Auswertung von $e_1 + e_2$ zunächst e_2 und danach e_1 auswerten. Werden in den beiden Ausdrücken direkte I/O-Aufrufe

benutzt, spielt die Reihenfolge der Auswertung jedoch eine Rolle, da sie die Reihenfolge der I/O-Aufrufe wider spiegelt.

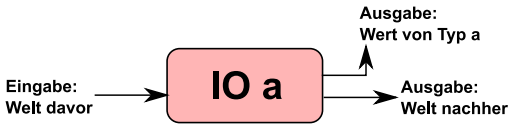
Aus all den genannten Gründen, wurde in Haskell ein anderer Weg gewählt. I/O-Operationen werden mithilfe des so genannten *monadischen I/O* programmiert. Hierbei werden I/O-Aufrufe vom funktionalen Teil gekapselt. Zu Programmierung steht der Datentyp `IO a` zu Verfügung. Ein Wert vom Typ `IO a` stellt jedoch kein Ausführen von Ein- und Ausgabe dar, sondern eine *I/O-Aktion*, die erst beim *Ausführen* (außerhalb der funktionalen Sprache) Ein-/Ausgaben durchführt und anschließend einen Wert vom Typ `a` liefert.

D.h. man setzt innerhalb des Haskell-Programms I/O-Aktionen zusammen. Die große (durch `main`) definierte I/O-Aktion wird im Grunde dann außerhalb von Haskell ausgeführt.

Eine anschauliche Vorstellung dabei ist die folgende. Eine I/O-Aktion ist eine Funktion, die als Eingabe einen Zustand der Welt (des Rechners) erhält und als Ausgabe den veränderten Zustand der Welt sowie ein Ergebnis liefert. Als Haskell-Typ geschrieben:

```
type IO a = Welt -> (a,Welt)
```

Man kann dies auch durch folgende Grafik illustrieren:



Aus Sicht von Haskell sind Objekte vom Typ `IO a` bereits *Werte*, d.h. sie können nicht weiter ausgewertet werden. Dies passt dazu, dass auch andere Funktionen Werte in Haskell sind. Allerdings im Gegensatz zu „normalen“ Funktionen kann Haskell kein Argument vom Typ „Welt“ bereit stellen. Die Ausführung der Funktion geschieht erst durch das Laufzeitsystem, welche die Welt auf die durch `main` definierte I/O-Aktion anwendet.

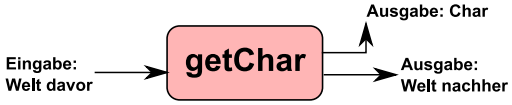
Um nun I/O-Aktionen in Haskell zu Programmieren werden zwei Zutaten benötigt: Zum Einen benötigt man (primitive) Basisoperationen, zum Anderen benötigt man Operatoren, um aus kleinen I/O-Aktionen größere zu konstruieren.

5.1.1 Primitive I/O-Operationen

Wir gehen zunächst von zwei Basisoperationen aus, die Haskell primitiv zur Verfügung stellt. Zum Lesen eines Zeichens vom Benutzer gibt es die Funktion `getChar`:

```
getChar :: IO Char
```

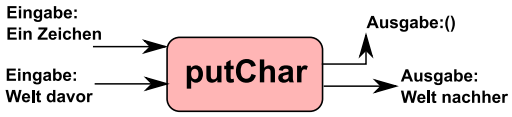
In der Welt-Sichtweise ist `getChar` eine Funktion, die eine Welt erhält und als Ergebnis eine veränderte Welt sowieso ein Zeichen liefert. Man kann dies durch folgendes Bild illustrieren:



Analog dazu gibt es die primitive Funktion `putChar`, die als Eingabe ein Zeichen (und eine Welt) erhält und nur die Welt im Ergebnis verändert. Da alle I/O-Aktionen jedoch noch ein zusätzliches Ergebnis liefern müssen, wird hier der 0-Tupel `()` verwendet.

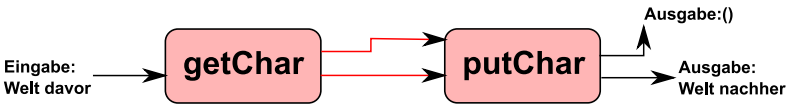
```
putChar :: Char -> IO ()
```

Auch `putChar` lässt sich mit einem Bild illustrieren:



5.1.2 Komposition von I/O-Aktionen

Um aus den primitiven I/O-Aktionen größere Aktionen zu erstellen, werden Kombinatoren benötigt, um I/O-Aktionen miteinander zu verknüpfen. Z.B. könnte man zunächst mit `getChar` ein Zeichen lesen, welches anschließend mit `putChar` ausgegeben werden soll. Im Bild dargestellt möchte man die beiden Aktionen `getChar` und `putChar` wiefolgt sequentiell ausführen und dabei die Ausgabe von `getChar` als Eingabe für `putChar` benutzen (dies gilt sowohl für das Zeichen, aber auch für den Weltzustand):



Genau diese Verknüpfung leistet der Kombinator `>>=`, der „bind“ ausgesprochen wird. Der Typ des Kombinators ist:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

D.h. er erhält eine IO-Aktion, die einen Wert vom Typ `a` liefert, und eine Funktion die einen Wert vom Typ a verarbeiten kann, indem sie als Ergebnis eine IO-Aktion vom Typ `IO b` erstellt.

Wir können nun die gewünschte IO-Aktion zum Lesen und Asgegeben eines Zeichens mithilfe von `>>=` definieren:

```
echo :: IO ()
echo = getChar >>= putChar
```

Ein andere Variante stellt der `>>`-Operator (gesprochen: „then“) dar. Er wird benutzt, um aus zwei I/O-Aktionen die Sequenz beider Aktionen zur erstellen, wobei das Ergebnis der ersten Aktion *nicht* von der zweiten Aktion benutzt wird (die Welt wird allerdings weitergereicht). Der Typ von `>>` ist:

```
(>>) :: IO a -> IO b -> IO b
```

Allerdings muss der Kombinator `>>` nicht primitiv zur Verfügung gestellt werden, da er leicht mithilfe von `>>=` definiert werden kann:

```
(>>) :: IO a -> IO b -> IO b
(>>) akt1 akt2 = akt1 >>= \_ -> akt2
```

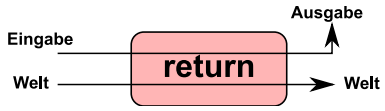
Mithilfe der beiden Operatoren kann man z.B. eine IO-Aktion definieren, die ein gelesenes Zeichen zweimal ausgibt:

```
echoDup :: IO ()
echoDup = getChar >>=(\x -> putChar x >>putChar x)
```

Angenommen wir möchten eine IO-Aktion erstellen, die zwei Zeichen liest und diese anschließend als Paar zurück gibt. Dann benötigen wir eine weitere Operation, um einen beliebigen Wert (in diesem Fall das Paar von Zeichen) in eine IO-Aktion zu verpacken, die nichts anderes tut, als das Paar zu liefern (die Welt wird einfach von der Eingabe zur Ausgabe weitergereicht). Dies leistet die primitive Funktion `return` mit dem Typ

```
return :: a -> IO a
```

Als Bild kann man sich die `return`-Funktion wie folgt veranschaulichen:



Die gewünschte Operation, die zwei Zeichen liest und diese als Paar zurück liefert kann nun definiert werden:

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>=\x ->
              getChar >>=\y ->
              return (x,y)
```

Das Verwenden von >>= und dem nachgestellten „`\x -> ...`“-Ausdruck kann man auch lesen als: Führe `getChar` durch, und binde das Ergebnis an `x` usw. Deswegen gibt es als syntaktischen Zucker die do-Notation. Unter Verwendung der do-Notation erhält die `getTwoChars`-Funktion folgende Definition:

```
getTwoChars :: IO (Char,Char)
getTwoChars = do {
                 x <- getChar;
                 y <- getChar;
                 return (x,y)}
```

Diese Schreibweise ähnelt nun sehr der imperativen Programmierung. Die do-Notation ist jedoch nur syntaktischer Zucker, sie kann mit >>= wegkodiert werden unter Verwendung der folgenden Regeln:

```
do { x<-e; s }  = e >>= \x -> do { s }
do { e; s }     = e >> do { s }
do { e }        = e
```

Als Anmerkung sei noch erwähnt, dass wir im Folgenden die geschweifte Klammerung und die Semikolons nicht verwenden, da diese durch Einrückung der entsprechenden Kodezeilen vom Parser automatisch eingefügt werden.

Als Fazit zur Implementierung von IO in Haskell kann man sich merken, dass neben den primitiven Operationen wie `getChar` und `putChar`, die Kombinatoren `>>=` und `return` ausreichen, um genügend viele andere Operationen zu definieren.

Wir zeigen noch, wie man eine ganze Zeile Text einlesen kann, indem man `getChar` wiederholt rekursiv aufruft:

```
getLine :: IO [Char]
getLine = do c <- getChar;
             if c == '\n' then
               return []
             else
               do
                cs <- getLine
                return (c:cs)
```

5.1.3 Monaden

Bisher haben wir zwar erwähnt, dass Haskell *monadisches* IO verwendet. Wir sind jedoch noch nicht darauf eingegangen, warum dieser Begriff verwendet wird. Der Begriff *Monade* stammt aus dem Gebiet der Kategorientheorie (aus der Mathematik). Eine Monade besteht aus einem Typkonstruktor `M` und zwei Operationen:

```
(>>=)  :: M a -> (a -> M b) -> M b
return :: a -> M a
```

wobei zusätzlich die folgenden drei Gesetze gelten müssen.

(1) `return x >>= f  = f x`

(2) `m >>= return  = m`

(3)
```
m1 >>= (\x -> m2 >>= (\y -> m3))
                =
(m1 >>= (\x -> m2)) >>= (\y -> m3)
```

Man kann nachweisen, dass diese Gesetze für `M = IO` und den entsprechenden Operatoren `>>=` und `return` erfüllt sind. Da Monaden die Sequentialisierung erzwingen, ergibt sich, dass die Implementierung des IO in Haskell sequentialisierend ist, was i.A. gewünscht ist.

Eine der wichtigsten Eigenschaften des monadischen IOs in Haskell ist, dass es keinen Weg aus einer Monade heraus gibt. D.h. es gibt keine Funktion `f:: IO a -> a`, die aus einem in einer I/O-Aktion verpackten Wert nur diesen Wert extrahiert. Dies erzwingt, dass man `IO`

nur innerhalb der Monade programmieren kann und i.A. rein funktionale Teile von der I/O-Programmierung trennen sollte.

Dies ist zumindest in der Theorie so. In der Praxis stimmt obige Behauptung nicht mehr, da alle Haskell-Compiler eine Möglichkeit bieten, die IO-Monade zu „knacken“. Im folgenden Abschnitt werden wir uns mit den Gründen dafür beschäftigen und genauer erläutern, wie das „Knacken“ durchgeführt wird.

5.1.4 Verzögern innerhalb der IO-Monade

Wir betrachten ein Problem beim monadischen Programmieren. Wir schauen uns die Implementierung des `readFile` an, welches den Inhalt einer Datei ausliest. Hierfür werden intern `Handle`s benutzt. Diese sind im Grunde „intelligente“ Zeiger auf Dateien. Für `readFile` wird zunächst ein solcher Handle erzeugt (mit `openFile`), anschließend der Inhalt gelesen (mit `leseHandleAus`).

```
-- openFile :: FilePath -> IOMode -> IO Handle
-- hGetChar :: Handle -> IO Char

readFile :: FilePath -> IO String
readFile path =
 do
  handle <- openFile path ReadMode
  inhalt <- leseHandleAus handle
  return inhalt
```

Es fehlt noch die Implementierung von `leseHandleAus`. Diese Funktion soll alle Zeichen vom `Handle` lesen und anschließend diese als Liste zurückgegeben und den Handle noch schließen (mit `hClose`). Wir benutzen außerdem die vordefinierten Funktion `hIsEOF :: Handle -> IO Bool`, die testet ob das Dateiende erreicht ist und `hGetChar`, die ein Zeichen vom Handle liest.

Ein erster Versuch führt zur Implementierung:

```
leseHandleAus handle =
  do
    ende <- hIsEOF handle
    if ende then
          do
           hClose handle
           return []
     else do
           c  <- hGetChar handle
           cs <- leseHandleAus handle
           return (c:cs)
```

Diese Implementierung funktioniert, ist allerdings sehr speicherlastig, da das letzte `return` erst ausgeführt wird, nachdem auch der rekursive Aufruf durchgeführt wurde. D.h. wir lesen

die gesamte Datei aus, bevor wir irgendetwas zurückgegeben. Dies ist unabhängig davon, ob wir eigentlich nur das erste Zeichen der Datei oder alle Zeichen benutzen wollen. Für eine verzögert auswertende Programmiersprache und zum eleganten Programmieren ist dieses Verhalten nicht gewünscht. Deswegen benutzen wir die Funktion `unsafeInterleaveIO :: IO a -> IO a`, die die strenge Sequentialisierung der IO-Monade aufbricht, d.h. anstatt die IO-Aktion sofort durchzuführen wird beim Aufruf innerhalb eines do-Blocks:

```
do
 ...
 ergebnis <- unsafeInterleaveIO aktion
 weitere_Aktionen
```

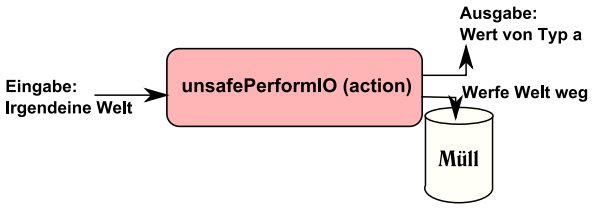
nicht die Aktion `aktion` durchgeführt (berechnet), sondern direkt mit den weiteren Aktionen weitergemacht. Die Aktion `aktion` wird erst dann ausgeführt, wenn der Wert der Variablen `ergebnis` benötigt wird.

Die Implementierung von `unsafeInterleaveIO` verwendet `unsafePerformIO`:

```
unsafeInterleaveIO a = return (unsafePerformIO a)
```

Die monadische Aktion `a` vom Typ `IO a` wird mittels `unsafePerformIO` in einen nicht-monadischen Wert vom Typ `a` konvertiert; mittels `return` wird dieser Wert dann wieder in die `IO`-Monade verpackt.

Die Funktion `unsafePerformIO` „knackt“ also die IO-Monade. Im Welt-Modell kann man sich dies so vorstellen: Es wird irgendeine Welt benutzt, um die IO-Aktion durchzuführen. Anschließend wird die neue Welt nicht weitergereicht, sondern sofort weggeworfen.



Die Implementierung von `leseHandleAus` ändern wir nun ab in:

```
leseHandleAus handle =
 do
  ende <- hIsEOF handle
  if ende then
        do
         hClose handle
         return []
   else do
         c  <- hGetChar handle
         cs <- unsafeInterleaveIO (leseHandleAus handle)
         return (c:cs)
```

Nun liefert `readFile` schon Zeichen, bevor der komplette Inhalt der Datei gelesen wurde. Beim Testen im Interpreter sieht man den Unterschied.

Mit der Version ohne `unsafeInterleaveIO`:

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
'1'
(7.09 secs, 263542820 bytes)
```

Mit Benutzung von `unsafeInterleaveIO`:

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
'1'
(0.00 secs, 0 bytes)
```

Beachte, dass beide Funktionen `unsafeInterleaveIO` und `unsafePerformIO` nicht vereinbar sind mit monadischem IO, da sie die strenge Sequentialisierung aufbrechen. Eine Rechtfertigung, die Funktionen trotzdem einzusetzen, besteht darin, dass man gut damit Bibliotheksfunktionen oder ähnliches definieren kann. Wichtig dabei ist, dass der Benutzer der Funktion sich bewusst ist, dass deren Verwendung eigentlich verboten ist und dass das Ein- / Ausgabeverhalten nicht mehr sequentiell ist. D.h. sie sollte nur verwendet werden, wenn das verzögerte I/O das Ergebnis nicht beeinträchtigt.

5.1.5 Speicherzellen

Haskell stellt primitive Speicherplätze mit dem (abstrakten) Datentypen `IORef` zur Verfügung. Abstrakt meint hier, dass die Implementierung (die Datenkonstruktoren) nicht sichtbar ist, die Konstruktoren sind in der Implementierung verborgen.

```
data IORef a = (nicht sichtbar)
```

Ein Wert vom Typ `IORef a` stellt eine Speicherzelle dar, die ein Element vom Typ `a` speichert. Es gibt drei primitive Funktionen (bzw. I/O-Aktionen) zum Erstellen und zum Zugriff auf `IORefs`. Die Funktion

```
newIORef :: a -> IO (IORef a)
```

erwartet ein Argument und erstellt anschließend eine IO-Aktion, die bei Ausführung eine Speicherzelle mit dem Argument als Inhalt erstellt.
Die Funktion

```
readIORef :: IORef a -> IO a
```

kann benutzt werden, um den Inhalt einer Speicherzelle (innerhalb der IO-Monade) auszulesen. Analog dazu schreibt die Funktion

```
writeIORef :: a -> IORef a -> IO ()
```

das erste Argument in die Speicherzelle (die als zweites Argument übergeben wird).

5.2 Concurrent Haskell

Concurrent Haskell ist eine Erweiterung von Haskell um Konstrukte zur nebenläufigen Programmierung. Diese Erweiterung wurde als notwendig empfunden, um IO-lastige Real-World-Anwendungen, wie. z.B. Graphische Benutzeroberflächen oder diverse Serverdienste (z.b. http-Server) in Haskell zu implementieren.

Die neuen Konstrukte sind

- Nebenläufige Threads und Konstrukte zum Erzeugen solcher Threads
- Konstrukte zur Kommunikation zwischen Threads und zur Synchronisation von Threads.

Insgesamt wurden dafür folgende neue primitive Operationen zu Haskell hinzugefügt, die innerhalb der `IO`-Monade verwendet werden dürfen.

Zur Erzeugung von nebenläufigen Threads existiert die Funktion

```
forkIO :: IO () -> IO ThreadId
```

Diese Funktion erwartet einen Ausdruck vom Typ `IO ()` und führt diesen in einem nebenläufigen Thread aus. Das Ergebnis ist eine eindeutige Identifikationsnummer für den erzeugten Thread. D.h. aus Sicht des Hauptthreads liefert `forkIO s` sofort ein Ergebnis zurück. Im GHC ist zusätzlich eine Funktion `killThread :: ThreadId -> IO ()` implementiert, die es erlaubt einen nebenläufigen Thread anhand seiner `ThreadId` zu beenden. Ansonsten führt das Beenden des Hauptthreads auch immer zum Beenden aller nebenläufigen Threads.

Zur Synchronisation und Kommunikation zwischen mehreren Threads wurde ein neuer Datentyp `MVar` (mutable variable) eingeführt. Hierbei handelt es sich um Speicherplätze, die im Gegensatz zum Datentyp `IORef` auch zur Synchronisation verwendet werden können.

Für den Datentyp stehen drei Basisfunktionen zur Verfügung:

- `newEmptyMVar :: IO (MVar a)` erzeugt eine leere `MVar`, die Werte vom Typ `a` speichern kann.
- `takeMVar :: MVar a -> IO a` liefert den Wert aus einer `MVar` und hinterlässt die Variable leer. Falls die entsprechende `MVar` leer ist, wird der Thread, der die `MVar` lesen möchte, solange blockiert, bis die `MVar` gefüllt ist.

  Wenn mehrere Threads ein `takeMVar` auf die gleiche (zunächst leere) Variable durchführen, so wird nachdem die Variable einen Wert hat, nur ein Thread bedient. Die anderen Threads warten weiter. Die Reihenfolge hierbei ist first-in-first-out (FIFO), d.h. jener Thread, der zuerst das `takeMVar` durchführt, wird zuerst bedient.
- `putMVar :: MVar a -> a -> IO ()` speichert den Wert des zweiten Arguments in der übergebenen Variablen, wenn diese leer ist. Falls die Variable bereits durch einen Wert belegt ist, wartet der Thread solange, bis die Variable leer ist. Genau wie bei `takeMVar` wird bei gleichzeitigem Zugriff von mehreren Threads auf die gleiche Variable mittels `putMVar` der Zugriff in FIFO-Reihenfolge abgearbeitet.

5.2.1 Einfache Verwendungen von MVars

Z.B. kann mithilfe von `MVars` auf das Beenden von nebenläufigen Threads gewartet werden:

```
main = do
        syncT1 <- newEmptyMVar
```

```
        syncT2 <- newEmptyMVar
        forkIO (thread1 >> putMVar syncT1 ())
        forkIO (thread2 >> putMVar syncT2 ())
        takeMVar syncT1
        takeMVar syncT2

thread1 = ...
thread2 = ...
```

Der Haupthread erstellt zwei leere MVars, die beiden nebenläufigen Threads befüllen die MVars nachdem sie ihre Berechnung beendet haben mit dem Wert () und der Hauptthread wartet darauf das beide MVars befüllt werden.

Da in MVars beliebige Daten abgelegt werden können, kann man mit MVars auch Daten zugriffsgeschützt ablegen. Z.B. kann ein Zähler mit atomarer Operation zum Inkrementieren durch

```
type Counter = MVar Int

atomicIncrement m = do
  r <- takeMVar m
  putMVar m (r+1)
```

implementiert werden.

Ein kritischer Abschnitt kann durch eine MVar geschützt werden, indem man vor dem Betreten die MVar entleert und diese danach wieder befüllt:

```
takeMVar mutex
... kritischer Abschnitt ...
putMVar mutex ()
```

Dabei ist mutex eine mit () gefüllte MVar vom Typ MVar ()

5.2.2 Semaphore

MVars können leicht wie binäre (und sogar starke) Semaphore benutzt werden. Das Anlegen eines Semaphores (mit 0 initialisiert) geschieht durch das Anlegen einer leeren MVar. Die signal-Operation füllt die MVar mit irgendeinem Wert und die wait-Operation versucht die MVar zu leeren. Da der Wert in der MVar nicht von Belang ist, verwenden wir das 0-Tupel (). Insgesamt ergibt sich die Implementierung von Semaphore als:

```
type Semaphore = MVar ()

newSem     :: IO Semaphore
newSem     =  newEmptyMVar

wait       :: Semaphore -> IO ()
wait sem   =  takeMVar sem
```

```
signal     :: Semaphore -> IO ()
signal sem =  putMVar sem ()
```

Beachte, dass die `wait`-Operation blockiert, wenn die `MVar` leer ist. Sobald eine `signal`-Operation die `MVar` füllt, wird die erste wartende `wait`-Operation durchgeführt und der zugehörige Prozess entblockiert. Wird eine verbotene `signal`-Operation auf einer vollen `MVar` durchgeführt, so ist das Verhalten nicht undefiniert, sondern der `signal` ausführende Prozess wird blockiert. Dieses Verhalten ist nicht durch Semaphore festgelegt, deswegen ist es eher uninteressant.

Die im Folgenden gezeigte `echoS`-Funktion liest eine Zeile von der Standardeingabe und druckt anschließend den gelesenen String auf der Standardausgabe aus. Die Funktion `zweiEchosS` erzeugt zwei nebenläufige Threads, welche beide die `echoS`-Funktion ausführen. Würde man das Einlesen und Ausgeben ungeschützt durchführen, so entsteht durch das Interleaving Chaos. Unsere Implementierung schützt jedoch den Zugriff auf die Standardeingabe und Standardausgabe durch Verwendung eines Semaphores (die durch eine `MVar` implementiert ist). Nur derjenige Thread, der ein erfolgreiches `wait` durchgeführt hat, darf passieren und auf die Standardeingabe bzw. -ausgabe zugreifen. Nachdem er dies erledigt hat, gibt er den kritischen Abschnitt wieder frei, indem er eine `signal`-Operation durchführt.

```
echoS sem i =
 do
  wait sem
  putStr $ "Eingabe fuer Thread" ++ show i ++ ":"
  line <- getLine
  signal sem
  wait sem
  putStrLn $ "Letzte Eingabe fuer Thread" ++ show i ++  ":" ++ line
  signal sem
  echoS sem i

zweiEchosS = do
               sem <- newSem
               signal sem
               forkIO (echoS sem 1)
               forkIO (echoS sem 2)
               block
```

Beachte, dass sich `MVars` in ihrem Blockierverhalten symmetrisch verhalten. Deshalb hätten wir die Implementierung von Semaphore auch andersherum gestalten können: Ein Semaphor, der mit 0 belegt ist, wird durch eine *gefüllte* `MVar` dargestellt, die `wait`-Operation führt eine `putMVar`-Operation durch und mit `takeMVar` wird signalisiert.

5.2.3 Weitere Operationen auf MVars und Threads

Die Haskell Bibliothek `Control.Concurrent.MVar` stellt noch weitere Operationen auf `MVars` zur Verfügung. Einige dieser Operationen werden im Folgenden erläutert.

- `newMVar:: a -> IO (MVar a)`
  Erzeugt eine neue `MVar`, die mit dem als erstes Argument übergebenen Ausdruck gefüllt wird.
- `readMVar :: MVar a -> IO a`
  Liest den Wert einer `MVar`, entnimmt ihn aber nicht. Ist die `MVar` leer, so blockiert der aufrufende Thread, bis die `MVar` gefüllt ist. Die Implementierung von `readMVar` ist eine Kombination von `takeMVar` und `putMVar`
- `swapMVar :: MVar a -> a -> IO a`
  Tauscht den Wert einer MVar aus, indem zunächst mit `takeMVar` der alte Wert gelesen wird, und anschließend der neue Wert mit `putMVar` in die `MVar` geschrieben wird. Die Rückgabe besteht im alten Wert der `MVar`. Beachte, dass der Austausch *nicht* atomar geschieht, d.h. falls nach dem Herausnehmen des alten Wertes ein zweiter Thread die `MVar` beschreibt, kann ein „falscher“ Wert in der `MVar` stehen.
- `tryTakeMVar :: MVar a -> IO (Maybe a)`
  `tryTakeMVar` versucht eine `takeMVar`-Operation durchzuführen. Ist die `MVar` vorher gefüllt, so wird sie entleert und der Wert der `MVar` als Ergebnis (mit `Just` verpackt) zurück geliefert. Ist die `MVar` leer, so wird *nicht* blockiert, sondern `Nothing` als Wert der I/O-Aktion zurück geliefert.
- `tryPutMVar :: MVar a -> a -> IO Bool`
  Analog zu `tryTakeMVar` ist `tryPutMVar` eine nicht-blockierende Version von `putMVar`. Das Ergebnis der Aktion ist ein Boolescher Wert: War das Füllen der `MVar` erfolgreich, so ist der Wert `True`, andernfalls `False`
- `isEmptyMVar :: MVar a -> IO Bool`
  Testet, ob eine `MVar` leer ist.
- `modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()`
  `modifyMVar_` wendet eine Funktion auf den Wert einer `MVar` an. Die Implementierung ist sicher gegenüber Exceptions: Falls ein Fehler bei der Ausführung auftritt, so stellt `modifyMVar_` sicher, dass der alte Wert der `MVar` erhalten bleibt.

Auch für Concurrent Haskell selbst gibt es weitere Operationen (definiert in `Control.Concurrent`). Wir zählen diese mit einigen bereits erwähnten Operationen auf:

- `forkIO :: IO () -> IO ThreadId`
  `forkIO` erzeugt einen nebenläufigen Thread. Dieser ist leicht-gewichtig, er wird durch das Haskell-Laufzeitsystem verwaltet.
- `forkOS :: IO () -> IO ThreadId`
  `forkOS` erzeugt einen „bound thread“, der durch das Betriebssystem verwaltet wird.
- `killThread :: ThreadId -> IO ()`
  `killTread` beendet den Thread mit der entsprechenden Identifikationsnummer. Wenn der Thread bereits vorher beendet ist, dann ist `killThread` wirkungslos.
- `yield :: IO ()`
  `yield` forciert einen Context-Switch: Der aktuelle Thread wird von aktiv auf bereit gesetzt. Ein anderer Thread wird aktiv.
- `threadDelay :: Int -> IO ()`

`threadDelay` verzögert den aufrufenden Thread um die gegebene Zahl an Mikrosekunden.

5.2.4 Erzeuger / Verbraucher-Implementierung mit 1-Platz Puffer

Mithilfe von `MVars` kann ein Puffer mit einem Speicherplatz problemlos implementiert werden. Der Puffer selbst wird durch eine `MVar` dargestellt

```
type Buffer a = MVar a
```

Das Erzeugen eines Puffers entspricht dem Erzeugen einer `MVar`.

```
newBuffer = newEmptyMVar
```

Der Erzeuger schreibt Werte in den Puffer, und blockiert, solange der Puffer voll ist. Dies entspricht genau der `putMVar`-Operation:

```
writeToBuffer = putMVar
```

Der Verbraucher entnimmt einen Wert aus dem Puffer und blockiert, falls der Puffer leer ist. Dieses Verhalten wird genau durch `takeMVar` implementiert.

```
readFromBuffer  = takeMVar
```

5.2.5 MVars zur Verwaltung von Zuständen

Wir betrachten eine Beispiel aus (Marlow, 2013). Wir wollen ein Telefonbuch so in einem Programm verwalten, dass nebenläufige Zugriffe darauf sicher und korrekt möglich sind. Das Telefonbuch besteht aus einer Zuordnung von Namen auf Telefonnummern, wofür wir eine `Map` aus der Bibliothek `Data.Map` verwenden (diese ist durch einen balancierten Suchbaum implementiert). Der aktuelle Zustand des Telefonbuchs wird in einer `MVar` gespeichert, d.h. es ergeben sich die Typsynonyme:

```
type Name               = String
type Telefonnummer      = String
type Telefonbuch        = Map.Map Name Telefonnummer
type TelefonbuchZustand = MVar Telefonbuch
```

Funktionalitäten zum Erstellen eines Telefonbuchs, Einfügen und Nachschauen von Einträgen lassen sich nun leicht implementieren:

```
neu :: IO TelefonbuchZustand
neu = newMVar Map.empty

einfuegen :: TelefonbuchZustand -> Name -> Telefonnummer -> IO ()
einfuegen m name nummer = do
 buch <- takeMVar m
 putMVar m (Map.insert name nummer buch)
```

```
nachschauen :: TelefonbuchZustand -> Name -> IO (Maybe Telefonnummer)
nachschauen m name = do
 buch <- takeMVar m
 putMVar m buch
 return (Map.lookup name buch)
```

Ein Testfunktion zum Ausprobieren ist:

```
main = do
 s <- neu
 sequence_ [einfuegen s ("name" ++ show i) (show i) | i <- [1..10000]]
 nachschauen s "name123" >>= print
 nachschauen s "unkown" >>= print
```

Eine Auswirkung der call-by-need-Auswertung von Haskell ist, dass die Ausdrücke normalerweise nicht ausgewertet werden, bevor sie in eine `MVar` geschrieben werden. So führt der Aufruf

```
   putMVar m (Map.insert name nummer buch)
```

in der `einfuegen`-Funktion dazu, dass der unausgewertete Ausdruck

```
    (Map.insert name nummer buch)
```

in die `MVar` geschrieben wird. Passiert dies oft, so wird der Ausdruck größer und größer und dies kann dann zu einem Space-Leak führen. Eine mögliche Abhilfe ist es, die strikte Auswertung zu erwingen. Mit f `$!` x wird das Argument x ausgewertet *bevor* die Funktion f mit dem Ergebnis dieser Auswertung aufgerufen wird. Daher könnten wir die Implementierung abändern zu:

```
   putMVar m $! (Map.insert name nummer buch)
```

Der Space-Leak tritt dann nicht mehr auf. Allerdings ist die `MVar` `m` nun länger gesperrt (solange bis die `Map.insert`-Operation fertig ist). Eine noch bessere Lösung ist:

```
einfuegen m name nummer = do
 buch <- takeMVar m
 let buch' = Map.insert name nummer buch
 putMVar m buch'
 seq buch' (return ())
```

Der Operator `seq` hat die Semantik, dass `seq` e_1 e_2 zuerst e_1 auswertet und dann e_2 zurückliefert[1]. Durch obiges Programm wird zuerst der unausgewertete Ausdruck in die `MVar` gelegt (dadurch ist sie nicht länger blockiert) und *danach* die `insert`-Operation ausgewertett.

[1] `f $! x` kann durch `f $! x= seq x (f x)` implementiert werden.

5.2.6 Das Problem der Speisenden Philosophen

In diesem Abschnitt betrachten wir das Problem der speisenden Philosophen und Implementierung für das Problem in Haskell. Zur Erinnerung: Es sitzen n Philosophen um einen runden Tisch und zwischen den Philosophen liegt genau je eine Gabel. Zum Essen benötigt ein Philosoph beide Gabeln (seine linke und seine rechte). Ein Philosoph denkt und isst abwechselnd. Wir stellen jede Gabel durch eine `MVar ()` dar, die Philosophen werden durch Threads implementiert. Die naive Lösung kann wie folgt implementiert werden

```
philosoph i gabeln =
 do
  let n = length gabeln                    -- Anzahl Gabeln
  takeMVar $ gabeln!!i                     -- nehme linke Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
  takeMVar $ gabeln!!(mod (i+1) n)         -- nehme rechte Gabel
  putStrLn $ "Philosoph " ++ show i ++ " isst ..."
  putMVar (gabeln!!i) ()                   -- lege linke Gabel ab
  putMVar (gabeln!!(mod (i+1) n)) ()       -- lege rechte Gabel ab
  putStrLn $ "Philosoph " ++ show i ++ " denkt ..."
  philosoph i gabeln
```

Das Hauptprogramm dazu erzeugt die Gabeln und die Philosophen:

```
philosophen n =
 do
   -- erzeuge Gabeln (n MVars):
   gabeln <- sequence $ replicate n (newMVar ())
   -- erzeuge Philosophen:
   sequence_ [forkIO (philosoph i gabeln) | i <- [0..n-1]]
   block
```

Beachte, dass `sequence_ :: [IO a] -> IO ()` eine Liste von IO-Aktionen sequentiell hintereinander ausführt, d.h. `sequence_ [`$a_1, \ldots, a_n$`]` ist äquivalent zu $a_1 \gg a_2 \gg \ldots \gg a_n$.

Diese Lösung für das Philosophen-Problem kann in einem globalen Deadlock enden, wenn alle Philosophen die linke Gabel belegen, und damit alle unendlich lange auf die rechte Gabel warten.

Eine Deadlock- und Starvationfreie Lösung besteht darin, den letzten Philosophen die Gabeln in verkehrter Reihenfolge aufnehmen zu lassen, da dann das Total-Order Theorem gilt. Die Haskell-Implementierung muss hierfür wie folgt modifiziert werden:

```
philosophAsym i gabeln =
  do
   let n = length gabeln -- Anzahl Gabeln
   if length gabeln == i+1 then -- letzter Philosoph
     do takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
        putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel ..."
```

```
      takeMVar $ gabeln!!i  -- nehme linke Gabel
      putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel und isst"
  else
   do
    takeMVar $ gabeln!!i  -- nehme linke Gabel
    putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
    takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
    putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel und isst"
 putMVar (gabeln!!i) () -- lege linke Gabel ab
 putMVar (gabeln!!(mod (i+1) n)) () -- lege rechte Gabel ab
 philosophAsym i gabeln
```

Das Hauptprogramm zum Erzeugen der Philosophenprozesse und der Gabeln bleibt dabei unverändert.

Wir hatten bereits gesehen, dass eine weitere Starvation- und Deadlockfreie Lösung des Philosophenproblems darin besteht, zu verhindern, alle Philosophen gleichzeitig an die Gabeln zu lassen. Hierfür hatten wir ein genereller Semaphor `raum` benutzt, der mit dem Wert $n-1$ initialisiert wurde.

Bisher haben wir keine Implementierung von generellen Semaphore in Haskell gesehen. Es gibt zwar Kodierungen von generellen Semaphore mithilfe von binären Semaphoren. Diese sind jedoch im Allgemeinen kompliziert. Da wir jedoch `MVars` zur Verfügung haben, ist die Implementierung von generellen Semaphore relativ einfach. In der Bibliothek `Control.Concurrent.QSem` findet man die Implementierung von generellen Semaphore. Zunächst benutzen wir diese Semaphore, um eine weitere Lösung für das Philosophen-Problem zu erstellen. Im Anschluss werden wir die Implementierung der generellen Semaphore erörtern.

Wir benutzen eine `QSem`, der mit $n-1$ initialisiert wird und dadurch verhindert, dass mehr als $n-1$ Philosophen Zugriff auf die Gabeln erhalten. Die Implementierung in Haskell für einen Philosophenprozess ist

```
philosophRaum i raum gabeln = do
  let n = length gabeln -- Anzahl Gabeln
  waitQSem raum -- genereller Semaphor
  putStrLn $ "Philosoph " ++ show i ++ " im Raum"
  takeMVar $ gabeln!!i  -- nehme linke Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat linke Gabel ..."
  takeMVar $ gabeln!!(mod (i+1) n) -- nehme rechte Gabel
  putStrLn $ "Philosoph " ++ show i ++ " hat rechte Gabel und isst"
  putMVar (gabeln!!i) () -- lege linke Gabel ab
  putMVar (gabeln!!(mod (i+1) n)) () -- lege rechte Gabel ab
  signalQSem raum
  putStrLn $ "Philosoph " ++ show i ++ " aus Raum raus"
  putStrLn $ "Philosoph " ++ show i ++ " denkt ..."
  philosophRaum i raum gabeln
```

Das Hauptprogramm muss leicht abgeändert werden, da der generelle Semaphor erzeugt werden muss:

```
philosophenRaum n =
 do
 gabeln <- sequence $ replicate n (newMVar ())
 raum <- newQSem (n-1)
 sequence [forkIO (philosophRaum i raum gabeln) | i <- [0..n-1]]
 block
```

5.2.7 Implementierung von generellen Semaphore in Haskell

Ein genereller Semaphor wird durch ein Paar vom Typ `(Int,[MVar ()])` dargestellt, die erste Komponente – die Zahl – stellt den Zähler eines Semaphores dar, die zweite Komponente ist eine Liste von `MVars`, an diesen `MVars` werden Prozesse blockiert. Die Semaphorenoperationen `wait` und `signal` modifizieren dieses Paar. Da dies eine Speicheränderung darstellt (also einen Seiteneffekt hat), wird das Paar in einem Speicherplatz abgelegt. Wir könnten hierfür eine `IORef` verwenden, müssten dann aber den Zugriff auf den Semaphor z.B. mithilfe eines binären Semaphores schützen. Da wir jedoch `MVars` zur Verfügung haben, können wir beide Aufgaben (speichern und schützen) erledigen, indem wir das Paar in einer `MVar` ablegen. Es ergibt sich somit der Typ eines generellen Semaphores `QSem` als:

```
type QSem = MVar (Int,[MVar ()])
```

Ein genereller Semaphor wird erzeugt, indem der Zähler k auf einen Initialwert gesetzt wird, und eine leere Menge von wartenden Prozessen generiert wird. In Haskell wird dies durch die Funktion `newQSem` implementiert:

```
newQSem :: Int -> IO QSem
newQSem k = do
             sem <- newMVar (k, [])
             return sem
```

Bei einer `wait`-Operation gibt es zwei Möglichkeiten: Ist der Wert k noch größer als 0, so wird dieser erniedrigt. Andernfalls muss der aufrufende Prozess blockiert werden und in die Menge der wartenden Prozesse eingefügt werden. Für Haskells `QSems` wird dies implementiert, indem das Paar zunächst aus der `MVar` entnommen wird (dadurch ist der Zugriff für andere Prozesse unmöglich) und anschließend entweder die erste Komponente erniedrigt wird (wenn $k > 0$ galt), oder eine neue, leere `MVar` erzeugt wird, die in die Liste der zweiten Komponente des Paares eingefügt wird (ganz hinten, damit eine FIFO-Reihenfolge implementiert wird). Anschließend wird das neue Paar in die `MVar` zurück geschrieben und zum Schluss darauf gewartet dass, die leere `MVar` in der Liste gefüllt wird. Der Code in Haskell für die `waitQSem`-Operation hat somit die Form:

```
waitQSem :: QSem -> IO ()
waitQSem sem = do
 (k,blocked) <- takeMVar sem
 if k > 0 then
   putMVar sem (k-1,[])
  else do
```

```
    block <- newEmptyMVar
    putMVar sem (0, blocked++[block])
    takeMVar block
```

Für die `signal`-Operation wird geprüft, ob es blockierte Prozesse gibt, indem das Paar aus der `MVar` herausgenommen wird und anschließend überprüft wird, ob die Liste der zweiten Paarkomponente leer ist. Gibt es keine blockierten Prozesse, so wird der Zähler k erhöht. Andernfalls wird die erste `MVar` der Liste aus der Liste entfernt und gefüllt (dadurch wird der an der `MVar` blockierte Prozess entblockiert). Die Implementierung in Haskell ist:

```
signalQSem :: QSem -> IO ()
signalQSem sem = do
   (k,blocked) <- takeMVar sem
   case blocked of
     []               -> putMVar sem (k+1,[])
     (block:blocked') -> do
                         putMVar sem (0,blocked')
                         putMVar block ()
```

In Haskell gibt es noch eine Erweiterung der generellen Semaphore. Bei generellen Semaphore wird der Zähler k um eins erhöht oder erniedrigt. Haskells Erweiterung besteht darin, dass man eine Zahl angegeben kann, die bei der `signal`-Operation angibt, um wieviel „Ressourcen“ erhöht werden soll, und bei der `wait`-Operation angibt, wieviel „Ressourcen“ ein Prozess benötigt, um nicht blockiert zu werden. Die Implementierung erweitert die Liste der wartenden Prozesse um eine weitere Komponente: Es wird mit abgespeichert, wieviel Ressourcen ein Prozess benötigt bevor er entblockiert werden darf. Der Typ der so genanntenQSemNs ergibt sich damit als:

```
type QSemN = MVar (Int,[(Int,MVar ())])
```

Die Funktion zum Erzeugen einer neuen `QSemN` gleicht der vorherigen Funktion zum Erzeugen einer `QSem`:

```
newQSemN :: Int -> IO QSemN
newQSemN initial = do
   sem <- newMVar (initial, [])
   return sem
```

Die `wait`-Operation darf nun nur dann zum Nichtblockieren führen, falls genügend Ressourcen vorhanden sind. Sie erhält ein weiteres Argument, welches die angeforderten Ressourcen angibt:

```
waitQSemN :: QSemN -> Int -> IO ()
waitQSemN sem sz = do
  (k,blocked) <- takeMVar sem
  if (k - sz) >= 0 then
     putMVar sem (k-sz,blocked)
   else do
```

```
    block <- newEmptyMVar
    putMVar sem (k, blocked++[(sz,block)])
    takeMVar block
```

Die `signal`-Operation ist etwas komplizierter zu definieren, da nur solche und alle solchen Prozesse entblockiert werden, für die nun genügend viele Ressourcen zur Verfügung stehen (die neuen Ressourcen werden als zusätzlicher Parameter der `signal`-Operation übergeben):

```
signalQSemN :: QSemN -> Int  -> IO ()
signalQSemN sem n = do
   (k,blocked)   <- takeMVar sem
   (k',blocked') <- free (k+n) blocked
   putMVar sem (k',blocked')
 where
   free k []    = return (k,[])
   free k ((req,block):blocked)
     | k >= req = do
        putMVar block ()
        free (k-req) blocked
     | otherwise    = do
        (k',blocked') <- free k blocked
        return (k',(req,block):blocked')
```

Man kann mithilfe dieser `QSemN`s einfach einen Barrier implementieren: Der Barrier wird dargestellt durch ein Paar: Die erste Komponente merkt sich die Anzahl der zu synchronisierenden Prozesse, die zweite Komponente ist wiederum ein Paar bestehend aus der Zahl der bisher angekommenen Prozesse und einer `QSemN` an der die Prozesse synchronisiert werden. Da wir die Zahl der angekommenen Prozesse sicher verändern wollen, wird das ganze Paar in eine `MVar` gesteckt. Das ergibt den Typ:

```
type Barrier = (Int, MVar (Int, QSemN))
```

Die Funktion `newBarrier` erwartet eine Zahl (die Anzahl der zu synchronisierenden Prozesse) und erzeugt anschließend einen entsprechenden Barrier:

```
newBarrier n =
 do
  qsem <- newQSemN 0
  mvar <- newMVar (0,qsem)
  return (n,mvar)
```

Die `synchBarrier`-Funktion wird von jedem der zu synchronisierenden Prozesse aufgerufen. Die Funktion prüft zunächst, ob die Anzahl der angekommenen Prozesse schon der maximalen Anzahl entspricht. Ist dies nicht der Fall, wird die Anzahl der angekommenen Prozesse erhöht, und der Prozess wartet mit einer Einheit an der `QSemN`.

Der letzte ankommende Prozesse merkt, dass er der letzte ist und befreit alle Prozesse, indem er eine `signalQSemN`-Aktion ausführt, die $n-1$ Ressourcen freigibt ($n-1$, da er sich selbst nicht befreien muss!).

```
synchBarrier :: Barrier -> IO ()
synchBarrier (maxP,barrier) = do
 (angekommen,qsem) <- takeMVar barrier
 if angekommen+1 < maxP then do
   putMVar barrier (angekommen+1,qsem)
   waitQSemN  qsem 1
  else do
   signalQSemN qsem (maxP-1)
   putMVar barrier (0,qsem)
```

Der Code für die einzelnen Prozesse ist dann von der Form

```
prozess_i barrier = do
  'Code f\"ur die aktuelle Phase'
  synchBarrier barrier -- warte auf Synchronisierung
  prozess_i barrier -- starte n\"achste Phase
```

5.2.8 Kanäle beliebiger Länge

Über den Puffer kann immer nur ein Wert ausgetauscht werden, das ist i.A. nicht sinnvoll, wenn z.B. der Erzeuger wesentlich schneller ist als der Verbraucher, oder wenn es mehrere Erzeuger und Verbraucher gibt. Hierfür eignen sich Kanäle, d.h. Puffer beliebiger Länge (Diese werden in Haskell als Kanäle bezeichnet). Ein (FIFO)-Kanal besteht aus einer Liste von Elementen, wobei an einem Ende Werte angehängt, am anderen Ende Elemente gelesen (und entfernt) werden können.

Als Schnittstelle sollte somit verfügbar sein:

- ein Typ für den Kanal, der polymorph über dem Elementtyp ist:
  `type Kanal a`
- eine Funktion zum Erzeugen eines neuen, leeren Kanals:
  `neuerKanal :: IO (Kanal a)`
- eine Funktion zum Anhängen eines Elements an den Kanal:
  `schreibe :: Kanal a -> a -> IO ()`
- eine Funktion zum Lesen (und Entnehmen) des zuerst eingefügten Elements: `lese :: Kanal a -> IO a`

Zudem sollen keine Fehler auftreten, wenn mehrere Threads vom Kanal lesen, oder in den Kanal schreiben.

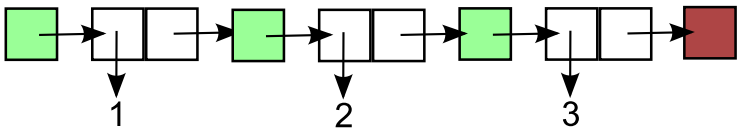
Der eigentliche Strom wird durch eine veränderliche Liste implementiert, indem jeder Tail durch eine `MVar` verkettet wird (d.h. jeder Tail ist veränderbar). Das Ende der Liste („Nil“) ist dann genau eine leere `MVar`:

```
type Strom a =  MVar (SCons a)
data SCons a =  SCons a (Strom a)
```

Z.B. kann der Strom, der 1,2 und 3 speichert, erzeugt werden mit

```
do
  ende <- newEmptyMVar
  drei <- newMVar (SCons 3 ende)
  zwei <- newMVar (SCons 2 drei)
  eins <- newMVar (SCons 1 zwei)
  return eins
```

Als Box-and-Pointer-Diagramm, wobei `MVars` graue, leere `MVars` dunkelgraue Kästchen sind, kann dieser Strom wie folgt dargestellt werden:



Ströme sind nun veränderbare Listen. Wir können z.B. zwei Ströme aneinander hängen oder einen Strom ausdrucken:

```
appendStroeme strom1 strom2 =
  do
    test <- isEmptyMVar strom2
    if test then return strom1
     else do
           kopf2 <- readMVar strom2
           ende  <- findeEnde strom1
           putMVar ende kopf2
           return strom1

findeEnde strom =
 do
  test <- isEmptyMVar strom
  if test then return strom
   else do
         SCons hd tl <- readMVar strom
         findeEnde tl

listToStrom []     = newEmptyMVar
listToStrom (x:xs) = do
                       tl <- listToStrom xs
                       v <- newMVar (SCons x tl)
                       return v

printStrom strom =
    do
     test <- isEmptyMVar strom
     if test then putStrLn "[]\n"
      else do
        SCons el tl <- readMVar strom
```

```
        putStr (show el ++ ":")
        printStrom tl
```

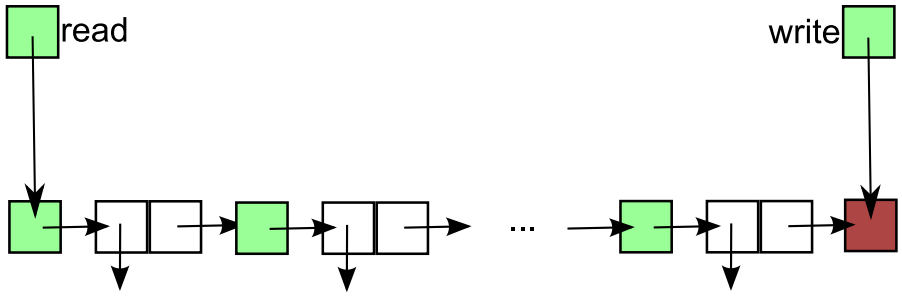
Beim Aneinanderhängen werden wirklich die benutzten Speicherplätze verändert, wie der folgende Test im Interpreter zeigt

```
*Main> f <- listToStrom [1..10]
*Main> g <- listToStrom [11..20]
*Main> printStrom f
1:2:3:4:5:6:7:8:9:10:[]
*Main> printStrom g
11:12:13:14:15:16:17:18:19:20:[]
*Main> h <- appendStroeme f g
*Main> printStrom h
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:[]
*Main> printStrom f
1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:[]
*Main> printStrom g
11:12:13:14:15:16:17:18:19:20:[]
```

Zur Definition des Kanals werden nun zwei `MVars` benutzt, die auf das Lese-Ende (links) und auf das Schreibe-Ende (rechts) eines Stroms zeigen. Die Verwendung von `MVars` sorgt dafür, dass immer nur ein Thread auf ein Ende zugreifen kann:

```
type Kanal a  = (MVar (Strom a), -- Lese-Ende
                 MVar (Strom a)) -- Schreibe-Ende
```

In Box-and-Pointer-Darstellung sieht ein Kanal somit wie folgt aus:



Eine leerer Kanal wird nun erzeugt, indem 3 `MVars` erzeugt werden: Eine für das Lese-Ende, eine für das Schreibe-Ende und eine für den leeren Strom:

```
neuerKanal :: IO (Kanal a)
neuerKanal =
  do
   hole  <- newEmptyMVar
   read  <- newMVar hole
   write <- newMVar hole
   return $ (read, write)
```

Schreiben an das Schreibe-Ende und Lesen vom Kanal kann nun wie folgt implementiert werden:

Beim Schreiben wird zunächst eine neue `MVar` erzeugt, die das neue Stromende darstellen soll, anschließend wird das alte Stromende berechnet. Ab diesem Zeitpunkt ist die `MVar write` leer, d.h. andere schreibende Threads müssen warten. Im nächsten Schritt wird das ehemalige Listenende durch das neue Element ersetzt. Im letzten Schritt erhält die `MVar write` als neuen Wert das neue Stromende.

```
schreibe :: Kanal a -> a -> IO ()
schreibe (read,write) val =
 do
  new_hole <- newEmptyMVar
  old_hole <- takeMVar write
  putMVar old_hole (SCons val new_hole)
  putMVar write new_hole
```

Beim Lesen wird im ersten Schritt die `MVar read` gelesen, sie ist somit leer, andere lesende Threads müssen warten. Im zweiten Schritt wird das erste Stromelement gelesen, anschließend die `MVar read` auf das ehemals zweite Element des Stroms gesetzt. Erst ab diesem Zeitpunkt können konkurrierende Threads fortfahren. Im letzten Schritt wird der Wert zurück gegeben.

```
lese :: Kanal a -> IO a
lese (read,write) =
  do
   kopf <- takeMVar read
   (SCons val stream) <- takeMVar kopf
   putMVar read stream
   return val
```

Man kann diese Kanäle leicht erweitern, sodass sie auch als Multicast-Kanäle verwendet werden können. Die folgende Funktion

```
dupliziere :: Kanal a -> IO (Kanal a)
dupliziere (read,write) =
  do
   hole <- readMVar write
   new_read <- newMVar hole
   return (new_read,write)
```

dupliziert einen Kanal: Alle schreibe-Operationen auf dem alten und dem neuen Kanal werden dupliziert. Der neu erstellte Kanal nimmt verwendet für das Schreibe-Ende den selben Zeiger wie der alte Kanal, aber erstellt einen neues Lese-Ende. Dadurch kann quasi zweimal gelesen werden (am alten und am neuen Kanal).

Damit die erwünschte Funktionalität aber wirklich vorhanden ist, muss das Lesen vom Kanal sorgfältiger programmiert werden und durch den Code

```
lese :: Kanal a -> IO a
```

```
lese (read,write) =
  do
   kopf <- takeMVar read
   (SCons val stream) <- readMVar kopf
   putMVar read stream
   return val
```

ersetzt werden, der anstelle des Abhängens mittel `takeMVar` nun `readMVar` verwendet.

Beachte, dass eine Operation zum rückgängig machen eine `lese`-Operation, nicht ohne Weiteres implementiert werden kann. Der Versuch

```
undoLese :: Kanal a -> a -> IO a
undoLese (read,write) val =
  do
   new_read <- newEmptyMVar
   hole <- takeMVar read
   putMVar new_read (SCons val hole)
   putMVar read new_read
```

funktioniert, wenn ein einzelner Thread arbeitet, aber er scheitert, wenn mehrere Threads arbeiten: Wenn der Kanal leer ist und eine Lese-Operation bereits ausgeführt wird, die blockiert ist und auf Elemente wartet. Dann wird eine undoLese-Operation ebenfalls blockieren (da die zu read zugehörige MVar leer ist) und ein Deadlock tritt ein.

Schließlich erwähnen wir, dass die hier vorgestellten Kanäle in Haskell vordefiniert im Modul `Control.Concurrent.Chan` sind, wobei der Typ für den Kanal `Chan a` heißt, und die Funktion zum Erzeugen, Lesen und Schreiben `newChan`, `readChan` und `writeChan` heißen.

5.2.9 Nichtdeterministisches Mischen

Mithilfe des Kanals können wir zwei Listen nichtdeterministisch mischen, indem wir zunächst einen Kanal erzeugen, anschließend für beide Listen einen nebenläufigen Thread erzeugen, der jeweils die Elemente der Liste auf den Kanal schreibt (mittels schreibeListeAufKanal), und im Hauptthread die Elemente vom Kanal lesen und in eine Liste einfügen (mittels leseKanal). Um zu erkennen, wann alle Elemente in den Kanal geschrieben wurden, ist die tatsächliche Implementierung etwas komplizierter: Wir schreiben nicht nur die Elemente in den Kanal, sondern für jedes Listenelement e das Paar (`False`,e) in den Kanal. Sobald das letzte Element geschrieben wurde (das Ende der Liste ist erreicht), schreiben wir (`True`,`bot`) auf den Kanal. Der Leser weiß also, dass keine weiteren Elemente mehr erscheinen, sobald er zweimal (`True`,`_`) gelesen hat. Zum Bewerkstelligen des verzögerten Auslesens des Kanals benutzen wir `unsafeInterleaveIO`.

```
ndMerge xs ys =
  do
   chan <- neuerKanal
   id_s <- forkIO (schreibeListeAufKanal chan xs)
```

```
   id_t <- forkIO (schreibeListeAufKanal chan ys)
   leseKanal chan False

leseKanal chan flag =
    do
       (flag1,el) <- lese chan

       if flag1 && flag then return [] else
        do
         rest <- unsafeInterleaveIO (leseKanal chan (flag || flag1))
         if flag1 then return (rest) else return (el:rest)

schreibeListeAufKanal chan []     = schreibe chan (True,undefined)
schreibeListeAufKanal chan (x:xs) =
 do
  schreibe chan (False,x)
  yield
  schreibeListeAufKanal chan xs
```

Eine solche Mischfunktion ist z.B. sehr nützlich, wenn Ereignisse von verschiedenen Erzeugern geliefert werden (z.B. Tastaturereignisse, Mausereignisse, usw.) und diese nacheinander vom Verbraucher (z.B. dem Betriebssystem) verarbeitet werden sollen.

Eine weitere Eigenschaft dieser Mischfunktion ist, dass sie zum Einen für unendliche Listen definiert ist und zum Anderen auch dann noch etwas liefert, wenn eine der beiden Eingabelisten nicht definiert ist, wie die folgenden Test im Interpreter zeigen:

```
*Main> ndMerge (1:2:(let bot = bot in bot) ) [3..] >>= print
[1,3,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25...
*Main> ndMerge [3..] (1:2:(let bot = bot in bot) ) >>= print
[3,1,4,2,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
```

Aufbauend auf Kanälen können weitere Konstrukte der nebenläufigen Programmierung implementiert werden, die wir jedoch nicht weiter betrachten werden.

5.2.10 Kodierung nichtdeterministischer Operationen

Mithilfe von Concurrent Haskell ist es möglich nichtdeterministische Operationen innerhalb der IO-Monade zu definieren.

5.2.10.1 Paralleles Oder

Während in einer sequentiellen Programmiersprache die Implementierung des parallelen Oders nicht möglich ist, kann dies mit den nebenläufigen Konstrukten von Concurrent Haskell programmiert werden.

Wir betrachten zunächst das Verhalten des (sequentiellen) Oders in Haskell, wobei s ein Ausdruck aus $\{\texttt{True}, \texttt{False}, \bot\}$ sei.

a	b	a || b
False	s	s
True	s	True
$\bot$	s	$\bot$

Ein *paralleles Oder* stellt die Anforderung das $a \vee b$ zu True auswertet, sofern a oder b zu True auswerten. Diese ist in der letzten Zeile verletzt, da `bot || True` nicht terminiert.

Die Implementierung des parallelen Oder mittels Concurrent Haskell basiert auf der folgenden Idee: Es werden beide Argumente des Oder nebenläufig ausgewertet und sobald eines der beiden zu True ausgewertet ist, wird dieser Wert als Resultat des parallelen Oders übernommen. Falls eine Argumentauswertung mit False endet, ist das Ergebnis gerade das andere Argument. Zur Synchronisation der beiden nebenläufigen Auswertungen benutzen wir eine MVar. Diese wird zunächst leer erzeugt, anschließend werden die beiden nebenläufigen Auswertungen gestartet, die ihr Ergebnis in die MVar schreiben. Der Hauptthread liest nun die MVar (d.h. er wartet solange, bis einer der beiden Threads etwas in die MVar geschrieben hat) und liefert das erhaltene Resultat als Ergebnis zurück. Vorher beendet er noch beide Threads, damit keine unnötigen Threads mehr laufen. Insgesamt ergibt das den Code:

```
por :: Bool -> Bool -> IO Bool
por s t = do
  ergebnisMVar <- newEmptyMVar
  id_s <- forkIO (if s then (putMVar ergebnisMVar True) else t)
  id_t <- forkIO (if t then (putMVar ergebnisMVar True) else s)
  ergebnis <- takeMVar ergebnisMVar
  killThread id_s
  killThread id_t
  return ergebnis
```

Das Verhalten von por lässt sich im Interpreter testen:

```
*Main> por False False >>= print
False
*Main> por False True >>= print
True
*Main> por True False >>= print
True
*Main> por True True >>= print
True
*Main> por True (let bot = bot in bot) >>= print
True
*Main> por (let bot = bot in bot) True >>= print
```

Die Implementierung hat somit das gewünschte Verhalten. Man beachte, dass der Wert von por $s\ t$ nur vom Wert der Argumente s und t abhängt, d.h. eigentlich ist das parallele Oder kein echter nichtdeterministischer Operator. Solche Operatoren werden auch als *schwach nichtdeterministisch* bezeichnet. Es es gibt keinen echten Grund das parallele Oder innerhalb der IO-Monade zu definieren, d.h. die referentielle Transparenz ist durch die Definition

```
safePor :: Bool -> Bool -> Bool
safePor s t = unsafePerformIO $ por s t
```

nicht verletzt. Allerdings geben die GHC-Entwickler keine Garantie, dass `unsafePerformIO` zusammen mit Concurrent Haskell wirklich wie erwartet funktioniert.

5.2.10.2 McCarthy's amb

Wir betrachten nun einen (*stark*) nichtdeterministischen Operator. Der vom Lisp-Erfinder John McCarthy vorgeschlagene Operator `amb` (abgeleitet von „ambigious choice“) erwartet zwei Argumente, wertet diese parallel (oder nebenläufig) aus. Wenn eine der beiden Auswertungen mit einem Wert endet, wird dieser als Gesamtergebnis übernommen.

Die Implementierung mittels Concurrent Haskell als Operator in der `IO`-Monade ist ähnlich zur Implementierung des parallelen Oders:

```
amb :: a -> a -> IO a
amb s t = do
 ergebnisMVar <- newEmptyMVar
 id_s <- forkIO (let x = s in seq x (putMVar ergebnisMVar x))
 id_t <- forkIO (let x = t in seq x (putMVar ergebnisMVar x))
 ergebnis <- takeMVar ergebnisMVar
 killThread id_s
 killThread id_t
 return ergebnis
```

Man beachte, dass `seq` die Auswertung zur WHNF erzwingt. Dies ist notwendig, da ansonsten der unausgewertete Ausdruck in die `MVar` geschrieben würde.

Das Ergebnis eines Aufrufs `amb` $s\ t$ hängt nicht alleinig von den Werten der Argumente s und t ab, denn falls beide Argumente zu einem „echten“ Wert auswerten können, kann sowohl der Wert von s oder aber auch der Wert von t als Gesamtresultat gewählt werden. Die Entscheidung hängt dann lediglich davon ab, welcher der beiden nebenläufigen Threads schneller fertig ist, was wiederum vom Scheduling und der damit verbundenen Ressourcenverteilung abhängig ist. Semantisch lässt sich das Ergebnis der Auswertung des `amb`-Konstruktes beschreiben als:

$$\texttt{amb}\ s\ t = \begin{cases} t, \text{ wenn } s \text{ nicht terminiert} \\ s, \text{ wenn } t \text{ nicht terminiert} \\ s \text{ oder } t, \text{ wenn } s \text{ und } t \text{ terminieren} \end{cases}$$

Der Operator `amb` ist aus verschieden Sichtweisen interessant. Er ermöglicht die Kodierung verschiedener nichtdeterministischer Operatoren, z.B. kann das parallele Oder mithilfe von `amb` definiert werden:

```
por2 :: Bool -> Bool -> IO Bool
por2 s t = amb (if s then True else t) (if t then True else s)
```

Außerdem kann ein `choice`, welches willkürlich zwischen seinen beiden Argumenten wählt mittels `amb` definiert werden:

```
choice :: a -> a -> IO a
choice s t = do res <- (amb (\x -> s) (\x -> t))
                return (res ())
```

Hierbei zeigt sich jedoch beim Testen im GHCi, dass meist das erste Argument gewählt wird, da anscheinend der zuerst erzeugte Thread auch zuerst Ressourcen erhält.

Wir können den `amb`-Operator auch auf eine Liste von beliebig vielen Argumenten erweitern:

```
ambList :: [a] -> IO a
ambList [x] = return x
ambList (x:xs) = do
                  l <- ambList xs
                  amb x l
```

Diese Implementierung funktioniert jedoch noch nur auf endlichen Listen, da durch die Verwendung der `IO`-Monade das Erzeugen der nebenläufigen Threads sequentialisiert wird. Für eine Liste $a : as$ wird der Ausdruck `amb a l` erst erzeugt, nachdem die durch l zu erzeugenden Threads wirklich erzeugt wurden.

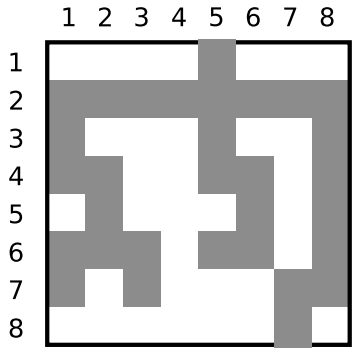
Mithilfe von `unsafeInterleaveIO :: IO a -> IO a` können wir dies (genau wie bei `readFile`) verhindern:

```
ambList :: [a] -> IO a
ambList [x] = return x
ambList (x:xs) = do
                  l <- unsafeInterleaveIO (ambList xs)
                  amb x l
```

Nun wird das `amb` sofort erzeugt, da `l` sofort einen Wert liefert. Diese Implementierung funktioniert nun auch für unendliche Listen. Z.B. liefert der folgende Aufruf ein Ergebnis im Interpreter:

```
ambList ([let bot = bot in bot] ++ [2..]) >>= print
2
```

Abschließend zum `amb` betrachten wir noch als Beispiel eine „parallelisierte“ Breitensuche, die sich mithilfe des `amb` sehr deklarativ programmieren lässt. Gegeben sei das folgende Labyrinth:



Das Ziel ist vom Eingang (oben) einen Weg zum Ausgang (unten) zu finden. Wir modellieren das Labyrinth mithilfe eines (n-ären) Suchbaums, wobei die Markierung eines Knotens genau den Koordinaten eines Feldes entspricht:

```
data Suchbaum a = Ziel a | Knoten a [Suchbaum a]

labyrinth =
 let
    kn51 = Knoten (5,1) [kn52]
    kn52 = Knoten (5,2) [kn42,kn53,kn62]
    ...
    kn78 = Ziel (7,8)
 in kn51
```

Für die eigentliche Breitensuche implementieren wir die Funktion `ndBFsearch`, die als erstes Argument die Liste der Pfadmarkierungen bis zum aktuellen Knoten und als zweites Argument den aktuellen Knoten erhält.

```
suche =
 do
  result <- ndBFsearch [] labyrinth
  print result

ndBFsearch res (Ziel a)      =
 return (reverse $ a:res)

ndBFsearch res (Knoten a []) =
 do
  yield
  ndBFsearch res (Knoten a [])

ndBFsearch res (Knoten a nf) =
 do
  nf' <- mapM (unsafeInterleaveIO . ndBFsearch (a:res)) nf
  ambList nf'
```

Zur Erläuterung von `ndBFsearch`: Wenn die Suche am Ziel ist, dann werden die Pfadmarkierungen als Liste zurück gegeben. Wenn wir an einem Knoten sind, der keine Nachfolger hat, so blockieren wir diese Suche indem wir in eine Endlosschleife gehen (das zusätzliche `yield` ist eine kleine Optimierung: Es erzwingt einen Kontextwechsel, d.h. ein anderer Thread ist erstmal dran). An einem Nichtzielknoten, der noch Nachfolger hat rufen wir rekursiv die Breitensuche für alle Nachfolger auf und verknüpfen die Suchergebnisse mittels `ambList`, d.h. die zuerst erfolgreiche Suche wird als Ergebnis gewählt. Wiederum benutzen wir `unsafeInterleavIO`, um die rekursiven Such-Aktionen zu erzeugen, aber noch nicht auszuwerten.

Der Aufruf im Interpreter ergibt den gesuchten Weg:

```
*Main> suche
```

```
[(5,1),(5,2),(6,2),(7,2),(8,2),(8,3),(8,4),(8,5),
 (8,6),(8,7),(7,7),(7,8)]
```

5.2.11 Futures

Futures sind Variablen deren Wert am Anfang unbekannt ist, aber in der Zukunft (daher der Name) verfügbar wird, sobald die zur Future zugehörige Berechnung beendet ist. In Haskell ist eigentlich jede Variable eine Future, da verzögert ausgewertet wird. Wir betrachten in diesem Abschnitt jedoch *nebenläufige Futures*, d.h. der Wert der Future wird durch eine nebenläufige Berechnung ermittelt. Man kann zwischen *expliziten* und *impliziten* Futures unterscheiden: Bei expliziten Futures muss der Wert einer Future explizit angefordert werden. D.h. wenn ein Thread den Wert einer Future benötigt muss er eine Operation auf der Future-Variablen ausführen, die solange wartet, bis der Wert der Future berechnet wurde. Bei impliziten Futures ist diese Operation unnötig, da die Auswertung automatisch den Wert der Future bestimmt, wenn dieser benötigt wird.

Der Vorteil von (impliziten) Futures liegt darin, dass man manche Anwendungen relativ einfach programmieren kann, da die Synchronisation automatisch geschieht.

Mithilfe von `forkIO` und `MVars` kann main explizite Futures wie folgt implementieren:

```
type EFuture a = MVar a

efuture :: IO a -> IO (EFuture a)
efuture act =
 do  ack <- newEmptyMVar
     forkIO (act >>= putMVar ack)
     return ack

force :: EFuture a -> IO a
force = readMVar
```

Eine explizite Future wird dabei durch eine `MVar` dargestellt. Das Erstellen einer expliziten Future wird durch die Funktion `efuture` durchgeführt. Diese erwartet eine IO-Aktion (die den Wert der Future berechnet) und liefert (eine Referenz auf) die Future. Zunächst wird eine leere `MVar` erstellt, anschließend wird eine nebenläufige Auswertung angestoßen: Diese führt die übergebene IO-Aktion aus und schreibt das Ergebnis in die `MVar` Da beim Aufruf von `forkIO` sofort weitergerechnet werden kann, wird sofort die letzte Zeile ausgeführt: Die `MVar` wird zurück gegeben (diese stellt die explizite Future dar).

Wenn der Wert der Future benötigt wird, muss der entsprechende Thread die Funktion `force` ausführen. Diese versucht mittels `readMVar` den Wert der Future zu lesen. Ist dieser noch nicht fertig berechnet, so wartet der aufrufende Thread bis der Wert der Future verfügbar ist.

Ein Beispiel zur Verwendung von Futures ist die parallele Berechnung der Summe der Knoten eines Baumes

```
data BTree a =
   Leaf a
 | Node a (BTree a) (BTree a)
```

```
treeSum (Leaf a)     = return a
treeSum (Node a l r) =
 do
  futl <- efuture (treeSum l)
  futr <- efuture (treeSum r)
  resl <- force futl
  resr <- force futr
  let result = (a + resl + resr)
   in seq result (return result)
```

Für jeden inneren Knoten des Baumes werden für den linken und rechten Teilbaum zwei Futures angelegt, die rekursiv deren Baumsummen berechnen. Anschließend wird auf den Wert beider Futures gewartet, zum Schluss wird addiert. Hierbei wird `seq` verwendet, welches die verzögerte Auswertung in Haskell aushebelt, damit das Resultat wirklich ausgewertet wird, bevor es zurück gegeben wird. Ohne `seq` hätten wir parallel die unausgewertete Summe erzeugt.

Die Programmierung mit expliziten Futures ist nicht wirklich komfortabel, da der Wert der Futures explizit angefordert werden muss. Im Beispiel wird dies auch noch sequentiell durchgeführt: Zuerst wird gewartet, dass der Wert der linken Teilsumme danach der Wert der rechten Teilsumme verfügbar ist. Es wäre eventuell effizienter zunächst `resl + a` als Zwischenergebnis zu berechnen und danach erst den Wert von `futr` anzufordern.

Es wäre besser, wenn wir auf das `force`-Kommando verzichten könnten und direkt schreiben könnten: `result = (a + futl + futr)`. Dies leisten die expliziten Futures allerdings nicht.

Mithilfe von `unsafeInterleaveIO` ist es jedoch möglich implizite Futures zu implementieren:

```
future :: IO a -> IO a
future code = do ack <-newEmptyMVar
                 thread <- forkIO (code >>= putMVar ack)
                 unsafeInterleaveIO (do result <- takeMVar ack
                                        killThread thread
                                        return result)
```

Mit `future` wird eine implizite Future erzeugt: Zunächst wird eine leere `MVar` erzeugt, anschließend wird (wie vorher) nebenläufig die übergebene IO-Aktion ausgeführt und das Resultat in die `MVar` geschrieben. Der letzte Schritt besteht darin, das Resultat aus der `MVar` zu lesen, den nebenläufigen Thread zu beenden und das Ergebnis zurück zu liefern. Würden wir dies ohne den umgebenen `unsafeInterleaveIO`-Aufruf durchführen, würde ein `future e` Aufruf solange blockieren, bis der Wert der Future ermittelt ist. Durch `unsafeInterleave` wird jedoch sofort ein Ergebnis geliefert (da die Sequentialisierung der `IO-Monade` aufgebrochen wird).

Beachte auch, dass es wenig sinnvoll wäre `unsafeInterleaveIO` um den gesamten Code zu schreiben, da dann der nebenläufige Thread nicht sofort gestartet würde.

Die parallele Baumsumme kann nun wie folgt berechnet werden

```
  treeSum (Leaf a)     = return a
  treeSum (Node a l r) = do
                           futl <- future (treeSum l)
                           futr <- future (treeSum r)
                           let result = (a + futl + futr)
                            in seq result (return result)
```

Wir sehen, dass die Implementierung nun einfach wurde, die Werte der Futures werden implizit durch die Auswertung angefordert (da `(+)` beide Argumentwerte benötigt).

5.2.12 Die Async-Bibliothek

Die Bibliothek Control.Concurrent.Async stellt einen ähnlichen Mechanismus zur Verfügung, wie die im letzten Abschnitt vorgestellten expliziten Futures. Dabei wird nicht der Begriff „Future“ verwendet, sondern Async für asynchrone Berechnungen. Zusätzlich ist das Ergebnis einer solchen Berechnung durch einen zusätzlichen Konstruktor verpackt. D.h. eine asynchrone Berechnung mit Ergebnis vom Typ `a` ist durch ein `Async a` repräsentiert und wird mit der Funktion `async` erzeugt. Das Warten auf ein asynchron berechnetes Ergebnis geschieht mit der Funktion `wait` (die analog zum `force` für explizite Futures ist. Das ergibt als Interface:

```
data Async a = Asynch (MVar a)

async :: IO a -> IO (ASync a)
async action = do
 var <- newEmptyMVar
 forkIO (action >>= \r -> putMVar var r)
 return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

Ein Beispiel aus (Marlow, 2013) ist der nebenläufige Download zweier Webseiten (Quellcodes z.B. das Modul `GetURL` zu (Marlow, 2013) sind unter https://hackage.haskell.org/package/parconc-examples-0.1 verfügbar):

```
import Control.Concurrent.Async
import Control.Concurrent
import GetURL
import qualified Data.ByteString  as B
main = do
 a1 <- async (getURL "http://www2.tcs.ifi.lmu.de/~letz/informationen.shtml")
 a2 <- async (getURL "http://www.ifi.lmu.de")
 r1 <- wait a1
 r2 <- wait a2
 print (B.length r1,B.length r2)
```

In (Marlow, 2013) sind viele weitere Beispiele und auch Erweiterung der Async-Implementierung erläutert, die auch Behandlung von Ausnahmen (d.h. Exceptions) miteinschließt, sodass nebenläufige auftretende Fehler auch als Resultate der Berechnungen auftreten und behandelt werden können. Wir gehen hier nicht tiefer darauf ein, und verweisen auf das Buch.

5.3 Software Transactional Memory in Haskell

Software Transactional Memory ist als Bibliothek in Haskell implementiert. Die Konstrukte sind in der Bibliothek `Control.Concurrent.STM` verfügbar. STM wurde 2005 von Tim Harris, Simon Marlow, Simon L. Peyton Jones und Maurice Herlihy in (Harris et al., 2005) für Haskell vorgeschlagen. Bemerkenswert ist, dass dieser Artikel der erste war, der die Konstrukte `retry` und `orElse` einführte. Eine der Grundideen von Haskells STM liegt darin, Transaktionen von normalen Ein- und Ausgabeoperationen zu trennen. Für diese Trennung wird Haskells Typsystem verwendet. Transaktionen sind vom Typ `STM a`, wobei `STM` genau wie der `IO`-Typ eine Monade ist.

Werte vom Typ `STM a` sind Transaktionen, die jedoch noch nicht ausgeführt sind. Zum Ausführen stellt Haskell STM die Funktion `atomically :: STM a -> IO a` zur Verfügung. Diese überführt eine STM-Aktion in eine IO-Operation, welche die entsprechende Transaktion atomar ausführt. Eine umgekehrte Operation, die IO-Aktionen in STM-Aktionen überführt ist von der Theorie her gesehen verboten[2]

Speicherplätze die durch Transaktionen verändert werden dürfen, werden durch *Transaktionsvariablen* dargestellt. Diese sind als abstrakter Datentyp `data TVar a = ...` primitiv implementiert. Die folgenden Operationen auf Transaktionsvariablen stehen zur Verfügung:

- `newTVar :: a -> STM (TVar a)`
  Erzeugt eine neue `TVar` mit Inhalt
- `readTVar :: TVar a -> STM a`
  Liest den den momentanen Wert einer `TVar`
- `writeTVar :: TVar a -> a -> STM ()`
  Schreibt einen neuen Wert in die `TVar`
- `newTVarIO :: TVar a -> a -> IO (TVar a)`
  Erzeugen einer TVar in der IO Monade

Innerhalb von STM-Transaktionen können damit folgende Konstrukte verwendet werden: Operationen auf TVars und rein funktionale Berechnungen. Da `STM` eine Monade ist, kann die do-Notation, der `return`-Operator und die Sequenzkombinatoren » und »= zur Komposition von STM-Aktionen verwendet werden. Insgesamt lässt sich das erlaubte Programmieren einer Transaktion wie folgt zusammenfassen:

```
atomically (
 do
  ... Zugriffe auf TVars und pure
```

[2] Allerdings stellt die STM-Bibliothek auch hierfür eine unsichere Funktion zur Verfügung (`unsafeIOToSTM :: IO a -> STM a`). Durch Verwenden dieser Funktion wird die strikte Trennung der IO-Aktionen von STM-Transaktionen verletzt, sie sollte deshalb nur in Ausnahmefällen verwendet werden.

```
      funktionale Berechnungen aber kein IO ...
 )
```

5.3.1 Die retry-Operation

Haskells STM stellt den `retry`-Operator zur Verfügung, er ist als `retry :: STM a` getypt. Die Semantik von `retry` besteht darin, dass die laufende Transaktion bei Aufruf von `retry` abgebrochen wird und die Transaktion neu gestartet wird. Nehmen wir an, dass eine Transaktion mehrere Operationen auf Transaktionsvariablen ausführt und anschließend ein `retry` ausführt. Wenn die Werte der Transaktionsvariablen nicht durch andere nebenläufige Transaktionen verändert werden, wird beim nächsten Durchlauf der Transaktion wieder das `retry`-Statement erreicht werden, was zum erneuten Neustart führt. Dieses Verhalten verbraucht unnötige Rechenzeit und ist auch nicht sinnvoll, da keine Seiteneffekte in Haskell erlaubt sind. Deshalb führt Haskells STM-Implementierung erst dann einen Neustart der Transaktion durch, wenn sich die Belegung der benutzten Transaktionsvariablen ändert. D.h. solange die Belegung der TVars unverändert ist, wird der `retry`-aufrufende Prozess blockiert. Durch dieses Verhalten lassen sich viele der klassischen Algorithmen für nebenläufige Prozesse auf das STM-Modell übertragen.

Tatsächlich ist die `retry`-Primitive nützlicher, als man denkt. Betrachte z.B. die Situation, dass viele Prozesse auf einen Zähler zugreifen und diesen verändern. Aufgabe ist es nun, einen Prozess zu implementieren, der den Zählerstand anzeigt, diesen aber nicht ständig aktualisiert (wegen des Flimmerns), sondern nur dann, wenn sich der Wert um mindestens 1000 nach dem letzten Aktualisieren geändert hat. Eine Implementierung des Szenarios ist

```
import Control.Concurrent
import Control.Concurrent.STM
import System.Random

main = do
 -- TVar fuer den Zaehler
 tv <- (newTVarIO 0)::IO (TVar Integer)
 -- 10 worker-Threads erzeugen
 mapM_ forkIO (replicate 10 (worker tv))
 -- aktualisieren der Anzeige starten
 updateDisplay tv

worker tv =
 do z <- randomRIO (-100,100)
    threadDelay 10000
    atomically $ do
        old <- readTVar tv
        writeTVar tv (z+old)
    worker tv
```

In der `main`-Funktion wird eine `TVar` erzeugt, die als Zähler dient, und es werden 10 Threads erzeugt, wobei jeder dieser Threads eine Transaktion ausführt, die zufällig den Zähler um eine Zahl im Intervall $[-100, 100]$ verändert.

Es fehlt noch die Implementierung der Funktion `updateDisplay`, welche die Anzeige des Zählers übernimmt. Diese kann wie folgt programmiert werden:

```
updateDisplay tv =
  do b <- readTVarIO tv
     printNext b -- einmal anzeigen
     loop b      -- in Schleife einsteigen
 where
  loop b =
    do next <- atomically $
                do c <- readTVar tv
                   if abs (b-c) < 1000 then retry else return c
       printNext next
       loop next

-- Hilfsfunktion zum Anzeigen
printNext i = putStr $ "\r" ++ show i ++ "        "
```

Die Funktion `updateDisplay` ruft `retry` auf, wenn die Änderung des Zählers weniger als 1000 beträgt. Dadurch blockiert der Thread, bis sich der Zähler ändert und prüft erst bei nächsten Änderung erneut. Dieses eher künstliche Beispiel kann z.B. hilfreich sein, wenn `updateDisplay` eine Rendering-Methode eines Fenstermanagers implementiert, die nur dann aktiv werden soll, wenn sich die Fensterposition ändert.

5.3.2 Beispiele

In diesem Abschnitt demonstrieren wir einige Beispiele, wie man klassische Konstrukte bzw. Probleme der nebenläufigen Programmierung als Transaktionen implementieren kann. Wir betrachten zunächst die Implementierung binärer Semaphore als STM-Transaktionen. Ein Semaphor wird durch eine `TVar` mit Booleschem Inhalt dargestellt, wobei der Inhalt `True` ist, wenn noch Ressourcen vorhanden sind (also k=1 für einen binären Semaphor gilt).

```
type Semaphore = TVar Bool
```

Erzeugen eines Semaphores kommt dem Erzeugen einer `TVar` gleich:

```
newSem :: Bool -> IO Semaphore
newSem k = newTVarIO k -- k True/False
```

Die `wait` und die `signal`-Operation können nun als Transaktionen definiert werden, wobei `wait` bei belegtem Semaphor ein `retry` zum Neustart der Transaktion durchführt:

```
wait :: Semaphore -> STM ()
wait sem = do
           b <- readTVar sem
           if b
            then writeTVar sem False
```

```
            else retry

signal :: Semaphore -> STM ()
signal sem = writeTVar sem True
```

Beachte, dass `wait` bei belegtem Semaphor zum Blockieren des aufrufenden Prozesses führt, da das `retry`-Statement erst dann einen Neustart durchführt, wenn sich die Belegung der Transaktionsvariablen (und damit des Semaphores) ändert.

Als nächstes Beispiel betrachten wir die Implementierung eines FIFO-Buffers mit unbegrenzter Länge. Hierfür speichern wir eine Liste in einer Transaktionsvariablen ab, d.h. der Typ eines Pufferspeichers ergibt sich als:

```
type Buffer a = TVar [a]
```

Zum Erzeugen eines neuen (d.h. leeren) Pufferspeichers wird eine TVar angelegt, die die leere Liste enthält:

```
newBuffer :: IO (Buffer a)
newBuffer = newTVarIO []
```

Das Einfügen in den Pufferspeicher wird durch die `put`-Operation realisiert, die eine Transaktion darstellt, die ein Element an die Liste anhängt:

```
put :: Buffer a -> a -> STM ()
put buffer item = do ls <- readTVar buffer
                     writeTVar buffer (ls ++ [item])
```

Die `get`-Operation entfernt das erste Element aus dem Pufferspeicher. Ist dieser leer, so blockiert die Operation, indem die Transaktion mittels `retry` abgebrochen wird:

```
get :: Buffer a -> STM a
get buffer = do ls <- readTVar buffer
                case ls of
                  [] -> retry
                  (item:rest) -> do writeTVar buffer rest
                                    return item
```

Als weiteres Beispiel betrachten wir die Implementierung von `MVars` innerhalb der STM-Monade. Eine `MVar` wird dabei wie folgt dargestellt:

```
type MVar a = TVar (Maybe a)
```

D.h. eine `MVar` ist eine `TVar`, die einen durch `Maybe` verpackten Elementtypen hat. Dies dient dazu zwischen einen leeren `MVar` (in diesem Fall hält die `TVar` den Wert `Nothing`) und einer gefüllten `MVar` zu unterscheiden. Eine leere `MVar` kann wie folgt erzeugt werden:

```
newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing
```

Die takeMVar-Operation überprüft zunächst, ob die MVar leer ist, d.h. ob die TVar den Wert Nothing enthält. Ist dies der Fall, so blockiert die Operation durch Abbrechen der Transaktion mittels retry. Andernfalls wird die MVar geleert:

```
takeMVar :: MVar a -> STM a
takeMVar mv = do
 v <- readTVar mv
 case v of
   Nothing -> retry
   Just val -> do
               writeTVar mv Nothing
               return val
```

Die putMVar-Operation blockiert (mithilfe von retry), falls die MVar gefüllt ist, anderenfalls wird der Wert in die MVar geschrieben:

```
putMVar :: MVar a -> a -> STM ()
putMVar mv val = do
 v <- readTVar mv
 case v of
   Nothing -> writeTVar mv (Just val)
   Just val -> retry
```

5.3.3 Transaktionen kombinieren

Wir haben bereits gesehen, dass größere Transaktionen aus kleineren komponiert werden können, indem die monadischen Operationen wie »= und » bzw. die do-Notation benutzt wird. Diese Kombinatoren erstellen aus Transaktionen zusammengesetzte *Sequenzen*. Z.B. kann man damit eine *atomare* Operation swapMVar definieren, die den Inhalt einer MVar austauscht und den alten Wert der MVar zurück liefert:

```
swapMVar :: MVar a -> a -> STM a
swapMVar mv val = do
  res <- takeMVar mv
  putMVar val
  return res
```

Hierbei sei anzumerken, dass zwischen takeMVar und putMVar kein anderer Thread eine Race Condition verursachen kann, da die zusammengesetzte Transaktion atomar ausgeführt wird.

Als weitere Möglichkeit der Komposition bietet Haskells STM den orElse-Kombinator an. Er hat den Typ orElse :: STM a -> STM a -> STM a und kombiniert zwei Transaktionen zu einer neuen Transaktion: Wenn die erste Transaktion erfolgreich ist, dann ist auch die kombinierte Transaktion erfolgreich. Wenn die erste Transaktion ein retry durchführt, dann wird die zweite Transaktion durchgeführt. Wenn beide Transaktionen ein retry durchführen, so wird die gesamte Transaktion neu gestartet.

Mithilfe von orElse kann beispielsweise ganz einfach eine nicht blockierende Version von putMVar implementiert werden.

```
tryPutMVar :: MVar a -> a -> STM Bool
tryPutMVar mv val =
 ( do
     putMVar mv val
     return True
  )
   `orElse`
   (return False)
```

Aufbauend auf dem binären `orElse` kann man problemlos neue Kombinatoren definieren. Z.B. kann man eine Operation definieren, die eine beliebige Liste von STM-Transaktionen erhält und diese Transaktionen alle mit `orElse` verknüpft, d.h. die erste erfolgreiche Transaktion führt zum Erfolg der gesamten Transaktion:

```
mergeSTM :: [STM a] -> STM a
mergeSTM transactions = foldl1 orElse transactions

foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

foldl1 f (x:xs) = foldl f x xs
```

Hierbei ergibt `mergeSTM [t1,...,tn]` gerade die links verschachtelte Verkettung mittels `orElse`: `(...((t1 `orElse` t2) `orElse` t3) `orElse` ... )`

Als abschließendes Beispiel zu Haskells STM Implementierung erläutern wir eine Implementierung des Problems der Speisenden Philosophen. Die Implementierung stammt von Josef Svenningsson[3]

Wir betrachten zunächst das Programm zur Simulation, d.h. die Erzeugung der nebenläufigen Prozesse, der Gabeln als Semaphore und die Ein- und Ausgabe des Programms. Für die Gabeln verwenden wir die vorher vorgestellte Semaphoreimplementierung in Haskells STM. Für die Ausgabe verwenden wir einen Pufferspeicher. In diesen werden die Philosophenprozesse Zeilenweise Strings einfügen, die durch das Hauptprogramm ausgelesen und ausgegeben werden. Beachte, dass die Philosophenprozesse innerhalb der Transaktionen nicht direkt auf ausdrucken können, da solche IO-Aktionen nicht innerhalb von Transaktionen erlaubt sind.

Die Funktion `simulation` erzeugt die `n` Gabeln (als Semaphore), einen Pufferspeicher für die Ausgabe und schließlich `n` Philosophenprozesse. Die dabei verwendete Funktion `philosoph` werden wir später erläutern:

```
simulation n = do
 forks <- sequence (replicate n (newSem True))
 outputBuffer <- newBuffer
 sequence_ [forkIO (philosoph
                        i
                        outputBuffer
```

[3] http://computationalthoughts.blogspot.com/2008/03/some-examples-of-software-transactional.html

```
                        (forks!!i)
                        (forks!!((i+1)`mod`n)))
                  | i <- [0..n-1]]
 output outputBuffer
```

Die als letztes aufgerufene Funktion `output` ist wie folgt definiert:

```
output buffer = do
 str <- atomically $ get buffer
 putStrLn str
 output buffer
```

Sie liest einen Eintrag aus dem Pufferspeicher (atomar als Transaktion mittels `get`), druckt diesen Eintrag aus und macht anschließend rekursiv weiter.

Das Programm für einen Philosophen erhält als Eingaben die Nummer des Philosophen, den Pufferspeicher für die Ausgabe, und zwei Gabeln. Neben einigen Ausgaben führt ein Philosoph das Aufnehmen der beiden Gabeln als atomare Transaktion durch:

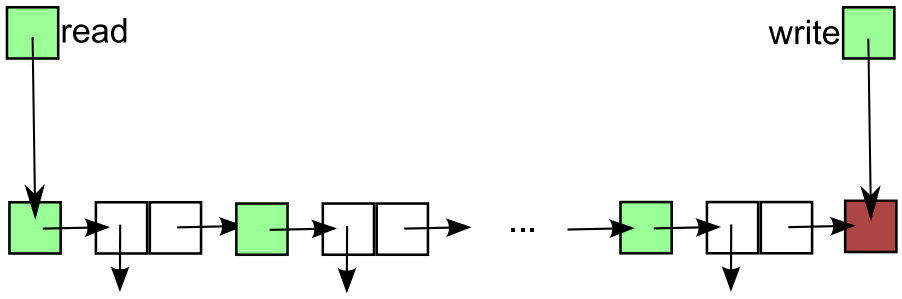
```
philosoph :: Int -> Buffer String -> Semaphore -> Semaphore -> IO ()
philosoph i out fork1 fork2 = do
 atomically $ put out ("Philosoph " ++ show i ++ " denkt.")
 atomically $ do wait fork1
                 wait fork2
 atomically $ put out ("Philosoph " ++ show i ++ " isst.")
 atomically $ do signal fork1
                 signal fork2
 philosoph i out fork1 fork2
```

5.3.4 Kanäle mit STM

Die in Abschnitt 5.2.8 mithilfe von `MVars` konstruierten Kanäle, lassen sich analog mithilfe von `TVars` und die Operationen als STM-Transaktionen implementieren. Da `TVars` nicht zwischen leer und voll unterscheiden, verwenden wir den Konstruktor `TNil`, um „leer“ von „voll“ zu unterscheiden:

```
type TKanal a    = (TVar (TVarListe a),TVar (TVarListe a))
type TVarListe a = TVar (TListe a)
data TListe a    = TNil | TCons a (TVarListe a)
```

Das ergibt das schon bekannte Bild:



wobei eine rote Box eine `TVar` gefüllt mit `TNil` entspricht und, grünen Boxen, `TVars` entsprechen, die entweder mit `TCons h t` (weiße Doppelbox) gefüllt sind, wobei h auf das Element und t auf den Rest zeigt, oder für das `read` und `write`-Paar auf `TVars` zeigen, die dem Anfang bzw. dem Ende des Kanals entsprechen.

Die Operationen zum Erzeugen eines Kanals, zum Schreiben auf den Kanal und zum Lesen und Entfernen eines Elements vom Kanal können nun als STM-Transaktionen formuliert werden:

```
neuerTKanal ::  STM (TKanal a)
neuerTKanal = do
   hole  <- newTVar TNil
   read  <- newTVar hole
   write <- newTVar hole
   return $ (read, write)


schreibe :: TKanal a -> a -> STM ()
schreibe (read,write) val = do
  newEnd <- newTVar TNil
  oldEnd <- readTVar write
  writeTVar write newEnd
  writeTVar oldEnd (TCons val newEnd)

lese :: TKanal a -> STM a
lese (read,write) =
  do
   listHead <- readTVar read
   tryHead <- readTVar listHead
   case tryHead of
     TNil -> retry
     TCons val rest -> do
                          writeTVar read rest
                          return val
```

Auch die Funktionalität zum Duplizieren des Kanals (d.h. alle Schreiboperationen nach dem Duplizieren können doppel (vom alten Kanal und vom neuen Kanal) gelesen werden), kann wie vorher, angepasst an die `TVars`, implementiert werden.

```
dupliziere :: TKanal a -> STM (TKanal a)
dupliziere (read,write) =
  do
   hole     <- readTVar write
   new_read <- newTVar hole
   return (new_read,write)
```

Während es mit der Kanalimplementierung mit `MVars` nicht möglich scheint, eine Lese-Operation rückgängig zu machen, ist das mit der STM-Implementierung kein Problem:

```
undoLese :: TKanal a -> a -> STM ()
undoLese (read,write) val =
   do
    listHead <- readTVar read
    newHead <- newTVar (TCons val listHead)
    writeTVar read newHead
```

Die atomare Ausführung der Transaktionen sichert die Korrektheit der Operation zu, auch dann, wenn der Kanal leer ist und eine Lese-Operation bereits ausgeführt wird, die blockiert ist und auf Elemente wartet: Diese ist durch `retry` blockiert. Die `undoLese`-Operation wird durchlaufen und danach kann die blockierte Lese-Operation durchgeführt werden.

Ebenso kann man leicht eine Operation implementieren, die prüft, ob eine Kanal leer ist (in der Kanalimplementierung mit `MVars` ist dies ebenfalls nicht (einfach) möglich:

```
istLeer :: TKanal a -> STM Bool
istLeer (read,write) =
 do
  listHead <- readTVar read
  tryHead <- readTVar listHead
  case tryHead of
   TNil       -> return True
   TCons _ _ ->  return False
```

5.3.5 Alternative Kanalimplementierung

Da mit STM auf beliebige Bedingungen gewartet werden kann (mittels `retry`) können wir eine einfachere (mehr funktionale) Implementierung von Kanälen angeben, welche eine Liste der Elemente innerhalb ein `TVar` verwaltet:

```
type TList a = TVar [a]

neueTList :: STM (TList a)
neueTList =  newTVar []

schreibeTList :: TList a -> a -> STM ()
schreibeTList l val =
    do
      xs <- readTVar l
      writeTVar l (xs ++ [val])

leseTList :: TList a -> STM a
leseTList l =
    do
      xs <- readTVar l
      case xs of
         [] -> retry
```

```
        (val:xs) -> do

                        writeTVar l xs
                        return val
```

Beachte, dass ein analoges Vorgehen bei `MVars` nicht so leicht möglich ist, da das Blockieren bei leerem Kanal nicht ohne weiteres implementiert werden kann.

Ebenso ist einerseits zu beachten, dass diese Implementierung die Operationen zum Duplizieren nicht unterstützt und dass sie in dieser Form ineffizient ist, da sie beim Schreiben das Element mit ++ anhängt, welches lineare Laufzeit hat. Eine bessere Implementierung kann mit folgendem Trick erzielt werden: Statt einer Liste xs, merkt man sich zwei Listen as und bs, wobei stets gelten soll, dass die eigentliche Liste xs durch as++(`reverse`bs) berechnet werden kann. Dadurch kann man in den allgemeinen Fällen schnell lesen (erstes Element von as) und schnell hinten anfügen (füge Elemente vorne an bs an). Der Spezialfall tritt ein, wenn man Lesen möchte, aber as leer ist: Dann muss man an das erste Element von (`reverse`bs). Dies benötigt lineare Laufzeit in der Länge von bs. Aber: Man kann anschließend as durch die berechnet Liste ersetzen und bs durch die leere Liste `[]`. Daher fällt der Aufwand für die `reverse`-Operation nur nach n-maligem Einfügen und Lesen an, sodass man *amortisiert* konstante Laufzeiten für Lesen und Schreiben hat.

Wir speichen die beiden Listen as und bs in separaten `TVars`, um weniger Konflikte zu erhalten. Die Implementierung mit diesem Trick ist dann:

```
type TListF a = (TVar [a],TVar [a])

neueTListF :: STM (TListF a)
neueTListF =
    do
     l <- newTVar []
     s <- newTVar []
     return (l,s)

schreibeTListF :: TListF a -> a -> STM ()
schreibeTListF (l,s) val =
    do
      bs <- readTVar s
      writeTVar s (val:bs)

leseTListF :: TListF a -> STM a
leseTListF (l,s) =
    do
      as <- readTVar l
      case as of
         (val:as') -> do
                        writeTVar l as'
                        return val
         [] -> do
```

```
bs <- readTVar s
case bs of
 [] -> retry
 _  -> do
        let (val:as) = reverse bs
        writeTVar l as
        writeTVar s []
        return val
```

5.3.6 Transaktionsmanagement

Wir erläutern, wie Haskells STM nebenläufig ablaufende Transaktionen verwaltet und wie Konflikte erkannt werden. D.h. wir beschreiben die Implementierung von STM Haskell im Glasgow Haskell Compiler. Wird eine Transaktion ausgeführt, so wird dafür ein Transaktions-Log angelegt (es genügt ein solches Log pro Thread anzulegen, da ein Thread nicht mehr als eine Transaktion gleichzeitig ausführt). Dieses stellt eine Tabelle dar, die alle TVars erfasst, die die Transaktion gelesen, geschrieben oder erzeugt hat. Für jede dieser TVars wird gespeichert, welche Werte die Transaktion gelesen hat und welche Werte in die TVars geschrieben werden sollen. Am Ende der Transaktion (wenn die Transaktion committen soll), wird geprüft, ob das Transaktions-Log *gültig* ist: Dazu wird geprüft, ob die gelesenen Werte gleich sind zu den aktuellen Werten. Ist dies der Fall, so ist das Transaktions-Log gültig und die lokalen Werte aus dem Transaktions-Log werden in die TVars geschrieben. Diese Schritte geschehen atomar, d.h. während des Prüfens und eventuellen Schreibens werden die TVars gesperrt. Ist das Transaktions-Log nicht gültig, so wird das Log verworfen und die Transaktion neu gestartet.

Wir betrachten das folgende Beispiel bestehend aus zwei nebenläufig ausführenden Transaktionen. Wir verwenden $\rightarrow$ um uns die Programmzeiger der beiden Transaktionen zu merken, also die Operation, die als nächstes ausgeführt wird. Mit $\Rightarrow$ markieren wir die Operation die gerade ausgeführt wird.

Zu Beginn befinden sich beide Transaktionen am Anfang. Für beide Transaktionen wird ein Transaktions-Log angelegt:

```
saldo::TVar Int
saldo = 7
```

Transaktion 1

```
    atomically (
→     do
        local <- readTVar saldo
        writeTVar saldo (local + 1)
         )
```

Transaktions-Log:

TVar	gelesen	geschrieben

Transaktion 2

```
    atomically (
→     do
        local <- readTVar saldo
        writeTVar saldo (local - 3)
         )
```

Transaktions-Log:

TVar	gelesen	geschrieben

Wir nehmen nun an, dass die zweite Transaktion einen Schritt machen darf. Sie liest den Wert der TVar saldo und schreibt den gelesenen Wert in ihr Transaktions-Log:

```
saldo::TVar Int
   saldo = 7
```

Transaktion 1

```
    atomically (
→      do
         local <- readTVar saldo
         writeTVar saldo (local + 1)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben

Transaktion 2

```
    atomically (
⇒      do
         local <- readTVar saldo
         writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	

Als nächsten Schritt nehmen wir an, dass Transaktion 1 die Lese-Operation durchführt:

```
saldo::TVar Int
   saldo = 7
```

Transaktion 1

```
    atomically (
       do
⇒        local <- readTVar saldo
         writeTVar saldo (local + 1)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
→        writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	

Im nächsten Schritt nehmen wir an, dass Transaktion 1 eine Operation durchführen darf. Da sie die TVar beschreiben möchte wird dies im lokalen Transaktions-Log durchgeführt:

```
saldo::TVar Int
   saldo = 7
```

Transaktion 1

```
    atomically (
       do
         local <- readTVar saldo
⇒        writeTVar saldo (local + 1)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	8

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
→        writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	

Wir nehmen, dass nun Transaktion 2 an der Reihe ist, und dementsprechend in ihr Transaktions-Log schreibt:

```
saldo::TVar Int
saldo = 7
```

Transaktion 1

```
    atomically (
       do
         local <- readTVar saldo
         writeTVar saldo (local + 1)
→       )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	8

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
⇒        writeTVar saldo (local - 3)
         )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	4

Nun sei Transaktion 1 wieder an der Reihe. Diese Transaktion möchte nun committen. Hierfür vergleicht sie die gelesenen Werte (in diesem Fall ist das nur der Wert von `saldo`) mit dem aktuellen Wert der TVar saldo. Da diese Werte identisch sind (beide 7), darf die Transaktion erfolgreich abschließen, der Wert von `saldo` wird auf 8 gesetzt, d.h. die Werte aus dem Transaktions-Log werden für die TVars übernommen, das Transaktions-Log nach dem Beenden gelöscht:

```
saldo::TVar Int
saldo = 8
```

Transaktion 1

```
    atomically (
       do
         local <- readTVar saldo
         writeTVar saldo (local + 1)
⇒       )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	8

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
         writeTVar saldo (local - 3)
→       )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	4

Nun ist nur noch Transaktion 2 aktiv. Diese Transaktion möchte nun auch committen. Da der gelesene Wert von `saldo` 7 ist, die TVar aber nun den Wert 8 besitzt, darf Transaktion 2 nicht committen, sondern muss abbrechen und neu starten.

```
saldo::TVar Int
saldo = 8
```

Transaktion 1

```
  atomically (
     do
       local <- readTVar saldo
       writeTVar saldo (local + 1)
       )
```

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
         writeTVar saldo (local - 3)
⇒       )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	7	4

Abort & Restart!

Hierfür wird das Transaktions-Log zurück gesetzt und der Programmzeiger springt auf den Anfang:

```
saldo::TVar Int
   saldo = 8
```

Transaktion 1

```
atomically (
   do
     local <- readTVar saldo
     writeTVar saldo (local + 1)
      )
```

Transaktion 2

```
    atomically (
→      do
         local <- readTVar saldo
         writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben

Nun liest Transaktion 2 die TVar saldo erneut:

```
saldo::TVar Int
   saldo = 8
```

Transaktion 1

```
atomically (
   do
     local <- readTVar saldo
     writeTVar saldo (local + 1)
      )
```

Transaktion 2

```
    atomically (
       do
⇒        local <- readTVar saldo
         writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	8	

Transaktion 2 führt die Schreiboperation durch:

```
saldo::TVar Int
   saldo = 8
```

Transaktion 1

```
atomically (
   do
     local <- readTVar saldo
     writeTVar saldo (local + 1)
      )
```

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
⇒        writeTVar saldo (local - 3)
          )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	8	5

Da der gelesene Wert von saldo mit dem aktuellen Wert übereinstimmt, darf Transaktion 2 committen und der Wert 5 wird für saldo übernommen:

```
saldo::TVar Int
   saldo = 5
```

Transaktion 1

```
atomically (
   do
     local <- readTVar saldo
     writeTVar saldo (local + 1)
      )
```

Transaktion 2

```
    atomically (
       do
         local <- readTVar saldo
         writeTVar saldo (local - 3)
⇒         )
```

Transaktions-Log:

TVar	gelesen	geschrieben
saldo	8	5

Dieses Beispiel verdeutlicht die Vorgehensweise: Erst nach dem Committen einer Transaktion sind die geänderten Werte nach außen sichtbar. Ebenso sieht man, dass Transaktionen immer erfolgreich sind (es sei denn sie führen ein `retry` durch), wenn sie alleine ausgeführt werden.

Die Implementierung im Glasgow Haskell Compiler ist noch etwas komplizierter: Jede `TVar` hat zusätzlich zu ihrem Inhalt noch eine assozierte Warteschlange von Threads. Diese dient dazu, dass Threads beim `retry` blockieren und erst dann wieder neu starten, wenn sich der Speicherinhalt einer der verwendeten `TVars` ändert: Wird `retry` ausgeführt, so hängt sich der entsprechende Thread in die Wartelisten aller gelesenen `TVars` ein und blockiert. Wenn ein anderer Thread erfolgreich committed, so entblockiert er alle Threads in den Warteschlangen der `TVars`, die er beschreibt. Diese prüfen, ob sich die Werte gegenüber den gelesenen nun geändert haben und starten dann neu, oder warten erneut, wenn die Werte noch gleich sind.

Um `orElse` zu implementieren, wird anstelle der oben genannten Tabelle mit alten und neuen Werten der `TVars`, ein Stack solcher Tabellen verwendet für die zu schreibenden Werte, während die gelesenen Werte nicht gestackt werden. Die Ausführung von `orElse` erzeugt eine neue Ebene des Stacks. Die Idee dabei ist: Wird für die Berechnung von `orElse` T_1 T_2 die Transaktion T_1 abgebrochen, so wird die oberste Ebene des Stacks gelöscht (die neuen zu schreibenden Werte), bevor T_2 gestartet wird. Die gelesenen Werte müssen aufgehoben werden, da sie für die Gültigkeit des Transaktions-Log auch geprüft werden.

Schließlich bemerken wir noch, dass das Transaktions-Log in regelmäßigen Abständen auf Gültigkeit geprüft wird: Ist das Log nicht mehr gültig, so wird die Transaktion abgebrochen und neu gestartet. Dies ist nicht ausschließlich eine Optimierung, sondern auch notwendig, um (bedingt) nicht-terminierende Transaktionen zu stoppen und neu zu starten.

Betrachte dazu den folgenden Haskell-Code:

```
i1 tv = atomically (
  do
    c <- readTVar tv
    if c then
     let loop i = do loop (i+1) in loop 0
     else return ()
 )

i2 tv = atomically (writeTVar tv False)

main = do tv <- atomically (newTVar True)
          s <- newEmptyMVar
          forkIO (i1 tv >> putMVar s ())
          forkIO (i2 tv >> putMVar s ())
          takeMVar s >> takeMVar s
```

Die von `i1` ausgeführte Transaktion läuft in die Nichtterminierung, solange die von `i2` ausgeführte Transaktion noch nicht committed ist. Allerdings ist die von `i1` ausgeführte Transaktion erfolgreich durchführbar, wenn `i2` vorher erfolgreich ist. Daher ist es semantisch richtig, dass die von `i1` ausgeführte Transaktion abgebrochen und neu gestartet wird und nicht für immer in der Endlosschleife verharrt. Das regelmäßige Prüfen des Transaktions-Log sichert dies zu.

5.3.7 Eigenschaften von Haskells STM

Wir betrachten abschließend einige Eigenschaften der STM-Implementierung in Haskell:

Isolation: Haskells STM implementiert so genannte „Strong Isolation“, da außerhalb von Transaktionen der (veränderliche) Zugriff auf Transaktionsvariablen unmöglich ist. Dies wird durch das Typsystem sichergestellt.

Granularität: Die Granularität von Haskells STM ist Wort-basiert, da Logging und Konflikterkennung pro Transaktionsvariable erfolgt.

Update: Das Update des gemeinsamen Speichers ist *verzögert* (im Gegensatz zu direktem Update), da neue Werte zunächst ins Transaktions-Log geschrieben werden und erst zum Commit-Zeitpunkt in den Speicher übernommen werden.

Concurrency Control: Die Nebenläufigkeitskontrolle kann als *optimistisch* bezeichnet werden, da keine Locks auf die gemeinsamen Speicherplätze verwendet werden. Dies passt zum verzögerten Update: Zunächst wird alles lokal gespeichert, in der Hoffnung, dass am Ende kein Konflikt auftritt und die Transaktion committen kann.

Synchronisation: Die Synchronisation von nebenläufigen Threads kann *blockierend* stattfinden, da `retry` zum Blockieren benutzt werden kann.

Conflict Detection: Die Konflikterkennung findet *spät* (engl. late) statt, da erst beim commit überprüft wird, ob es Konflikte zwischen lokalem Transaktions-Log und dem aktuellen Speicherzustand gibt.

Nested Transactions: Haskell unterstützt keine verschachtelten Transaktionen. Vielmehr sind diese unmöglich zu definieren, da das Typsystem diese verhindert: `atomically` hat den Typ `STM a -> IO a`, also `IO a` als Ausgabetyp. Genau solche IO-Aktionen können jedoch nicht innerhalb von Transaktionen verwendet werden.

Bezüglich der Korrektheitseigenschaften von STM-Systemen, erfüllt die Haskell-Implementierung von STM keine Opazität, da abbrechende Transaktionen inkonsistente Werte lesen können. Betrachte z.B. die Transaktion

```
trans tvar1 tvar2 =
  do
   t1 <- readTVar tvar1
   t2 <- readTVar tvar2
   if t1 /= t2 then error "stop" else
   writeTVar tvar1 (t1 +1)
   writeTVar tvar2 (t2 +1)
```

Wenn mehrere dieser Transaktionen nebenläufig laufen für die selben TVars, welche am Anfang beide mit 0 initialsiert sind. Dann sind nach jeder erfolgreichen Transaktion die Werte der TVars gleich, d.h. in einem STM-System, dass die Opazität erfüllt, tritt der Fehler nicht auf. In Haskell jedoch kann es sein, dass beim Lesen von tvar1 z.B. 5 gelesen wird, dann eine andere Transaktion committed, sodass beim Lesen von tvar2 nun 6 gelesen wird. Die Transaktion hat dann zwar einen Konflikt, aber der Konflikt wird erst später (beim committen oder einem temporären Prüfen des Log) entdeckt. Dadurch kann die Transaktion in den error-Fall laufen.

5.4 Quellennachweis

Die Darstellung von Haskells monadischen IO in Abschnitt 5.1 richtet sich im Wesentlichen nach (Peyton Jones, 2001). Ein gutes Buch zu Concurrent Haskell und parallelem Programmieren in Haskell, welches zum Teil auch als Vorlage für diese Kapitel verwendet wurde ist (Marlow, 2013). Auch STM-Haskell wird dort behandelt. Der Abschnitt 5.2 über Concurrent Haskell greift auch auf den Originalartikel (Peyton Jones et al., 1996) zurück, zwischenzeitlich wurden jedoch einige Funktionalitäten geändert, die der Dokumentation der entsprechenden Programmbibliotheken zu entnehmen sind. Weitere Einführungen zu Concurrent Haskell sind in (Peyton Jones & Singh, 2009) und (O'Sullivan et al., 2008) zu finden. Haskells STM Implementierung wird in (Harris et al., 2005), (Peyton-Jones, 2007, Kapitel 24) und in (Marlow, 2013) beschrieben. Auch in (Larus & Rajwar, 2006) findet sich ein Abschnitt über Haskells STM.

6

Semantische Modelle nebenläufiger Programmiersprachen

In diesem Kapitel betrachten wir Modelle für nebenläufige Programmiersprachen und deren Semantik. Solche *Kalküle* bestehen im Wesentlichen aus der *Syntax*, die festlegt, welche Ausdrücke in der Sprache gebildet werden dürfen und der *Semantik*, die angibt, welche Bedeutung die Ausdrücke haben. Diese Kalküle kann man als abgespeckte Programmiersprachen auffassen. „Normale“ Programmiersprachen werden oft während des Compilierens in solche einfacheren Sprachen übersetzt, da diese besser überschaubar sind und sie u.a. besser mathematisch handhabbar sind. Mithilfe einer Semantik kann man (mathematisch korrekte) Aussagen über Programme und deren Eigenschaften nachweisen. Das Gebiet der „Formale Semantiken“ unterscheidet im Wesentlichen drei Ansätze:

- Eine *axiomatische Semantik* beschreibt Eigenschaften von Programmen mithilfe logischer Axiome bzw. Schlussregeln. Weitere Eigenschaften von Programmen können dann mithilfe von logischen Schlussregeln hergeleitet werden. Im Allgemeinen werden von axiomatischen Semantiken nicht alle Eigenschaften, sondern nur einzelne Eigenschaften der Programme erfasst. Ein prominentes Beispiel für eine axiomatische Semantik ist der Hoare-Kalkül (Hoare, 1969).
- *Denotationale Semantiken* verwenden mathematische Räume, um die Bedeutung von Programmen festzulegen. Eine *semantische Funktion* bildet Programme in den entsprechenden mathematischen Raum ab. Für funktionale Programmiersprachen werden als mathematische Räume oft „Domains“, d.h. partiell geordnete Mengen, verwendet. Für nichtdeterministische Sprachen wird dieser Formalismus oft erweitert auf so genannte „Power-Domains“, d.h. Programme werden selbst auf Mengen abgebildet. Denotationale Semantiken sind mathematisch elegant, allerdings für umfangreichere Sprachen oft schwer zu definieren. Dies wird weiterhin erheblich schwieriger, wenn nebenläufige (bzw. nichtdeterministische) Sprachen verwendet werden.
- Eine *operationale Semantik* legt die Bedeutung von Programmen fest, indem sie definiert, wie Programme *ausgewertet* werden. Es gibt verschiedene Möglichkeiten operationale Semantiken zu definieren: *Zustandsübergangssysteme* geben an, wie sich der Zustand der Maschine (des Speichers) beim Auswerten verändert, *Abstrakte Maschine* geben ein Maschinenmodell zur Auswertung von Programmen an und *Ersetzungssysteme* definieren, wie der Wert von Programmen durch Term-Ersetzungen „ausgerechnet“ werden kann. Man kann operationale Semantiken noch in *Big-Step*- und *Small-Step*-Semantiken unterscheiden. Während Small-Step-Semantiken die Auswertung in kleinen Schritten festlegt, d.h. Programme werden schrittweise ausgewertet, legen Big-Step-Semantiken diese Auswertung in größeren (meist einem) Schritt fest. Oft erlauben Big-Step-Semantiken Freiheit bei der Implementierung eines darauf basierenden Interpreters, während bei Small-

Step-Semantiken meist ein Interpreter direkt aus den Regeln ablesbar ist.

Wir werden operationale Semantiken betrachten, die als Ersetzungssysteme aufgefasst werden können und Small-Step-Semantiken sind. Im ersten Abschnitt dieses Kapitels betrachten wir den Lambda-Kalkül, der *kein* Modell für nebenläufige Programmiersprachen ist, aber ein weit verbreitetes Modell für *sequentielle* Programmiersprachen. Der Grund für dessen Betrachtung ist, Konzepte, Definitionen und Eigenschaften am Lambda-Kalkül zu demonstrieren, um sie später auch für nebenläufige Modelle anzuwenden bzw. zu vergleichen.

Anschließend betrachten wir den π-Kalkül, der ein Modell für Message-Passing-Nebenläufigkeit ist, d.h. Prozesse kommunizieren ausschließlich über den Austausch von Nachrichten.

Danach betrachten wir den CHF-Kalkül der als Kernsprache von Concurrent Haskell gesehen werden kann (erweitert um Futures), als eine Erweiterung des Lambda-Kalküls aufgefasst werden kann und ein Shared-Memory-Modell ist, d.h. Prozesse kommunizieren über gemeinsamen Speicher.

6.1 Der Lambda-Kalkül

In diesem Abschnitt betrachten wir den Lambda-Kalkül als Modell für (insbesondere funktionale) sequentielle Programmiersprachen. Wir werden hierbei verschiedene Auswertungsstrategien (also operationale Semantiken) darstellen. Der Lambda-Kalkül wurde von Alonzo Church in den 1930er Jahren eingeführt (Church, 1941). Ausdrücke des Lambda-Kalküls können mit dem Nichtterminal **Expr** mithilfe der folgenden kontextfreien Grammatik gebildet werden:

$$\textbf{Expr} ::= V \mid \lambda V.\textbf{Expr} \mid (\textbf{Expr Expr})$$

Hierbei ist V ein Nichtterminal, welches eine Variable (aus einer unendlichen Menge von Variablennamen) generiert. $\lambda x.s$ wird als *Abstraktion* bezeichnet. Durch den Lambda-Binder λx wird die Variable x innerhalb des Unterausdrucks s gebunden, d.h. umgekehrt: Der Gültigkeitsbereich von x ist s. Eine Abstraktion wirkt wie eine anonyme Funktion, d.h. wie eine Funktion ohne Namen. Z.B. kann die Identitätsfunktion $id(x) = x$ im Lambda-Kalkül durch die Abstraktion $\lambda x.x$ dargestellt werden.

Das Konstrukt $(s\ t)$ wird als *Anwendung* (oder auch *Applikation*) bezeichnet. Mithilfe von Anwendungen können Funktionen auf Argumente angewendet werden. Hierbei darf sowohl die Funktion (der Ausdruck s) als auch das Argument t ein beliebiger Ausdruck des Lambda-Kalküls sein. D.h. insbesondere auch, dass Abstraktionen selbst Argumente von Anwendungen sein dürfen. Deshalb spricht man auch vom Lambda-Kalkül höherer Ordnung (bzw. higher-order Lambda-Kalkül). Z.B. kann man die Identitätsfunktion auf sich selbst anwenden, d.h. die Anwendung $id(id)$ kann in Lambda-Notation als $(\lambda x.x)\ (\lambda x.x)$ geschrieben werden.

Um die Gültigkeitsbereiche von Variablen formal festzuhalten, definieren wir die Funktionen FV und BV. Für einen Ausdruck t ist $BV(t)$ die Menge seiner *gebundenen Variablen* und $FV(t)$ die Menge seiner *freien Variablen*, wobei diese (induktiv) durch die folgenden Regeln definiert sind:

$$\begin{array}{lcllcl} FV(x) & = & x & BV(x) & = & \emptyset \\ FV(\lambda x.s) & = & FV(s) \setminus \{x\} & BV(\lambda x.s) & = & BV(s) \cup \{x\} \\ FV(s\ t) & = & FV(s) \cup FV(t) & BV(s\ t) & = & BV(s) \cup BV(t) \end{array}$$

Gilt $FV(t) = \emptyset$ für einen Ausdruck t, so sagen wir t ist *geschlossen* oder auch t ist ein *Programm*. Ist ein Ausdruck t nicht geschlossen, so nennen wir t *offen*.

Um die operationale Semantik des Lambda-Kalküls zu definieren, benötigen wir den Begriff der *Substitution*: Wir schreiben $s[t/x]$ für den Ausdruck, der entsteht, indem alle freien Vorkommen der Variable x in s durch den Ausdruck t ersetzt werden. Um Namenskonflikte bei dieser Ersetzung zu vermeiden, nehmen wir an, dass $x \notin BV(s)$ gilt. Unter dieser Annahme kann man die Substitution formal durch die folgenden Gleichungen definieren:

$$\begin{array}{lcl} x[t/x] & = & t \\ y[t/x] & = & y, \text{ falls } x \neq y \\ (\lambda y.s)[t/x] & = & \lambda y.(s[t/x]) \\ (s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x]) \end{array}$$

Z.B. ergibt $(\lambda x.z\ x)[(\lambda y.y)/z]$ den Ausdruck $(\lambda x.((\lambda y.y)\ x))$.

Kontexte C sind Ausdrücke, wobei genau ein Unterausdruck durch ein Loch (dargestellt mit $[\cdot]$) ersetzt ist. Man kann Kontexte auch mit der folgenden kontextfreien Grammatik definieren:

$$\mathbf{C} = [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{C})$$

In Kontexte kann man Ausdrücke *einsetzen*, um so einen neuen Ausdruck zu erhalten. Sei C ein Kontext und s ein Ausdruck. Dann ist $C[s]$ der Ausdruck, der entsteht, indem man in C anstelle des Loches den Ausdruck s einsetzt. Diese Einsetzung kann Variablen einfangen. Betrachte als Beispiel den Kontext $C = \lambda x.[\cdot]$. Dann ist $C[\lambda y.x]$ der Ausdruck $\lambda x.(\lambda y.x)$. Die freie Variable x in $\lambda y.x$ wird beim Einsetzen eingefangen.

Nun können wir α-Umbenennung definieren: Ein α-Umbenennungsschritt hat die Form:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ falls } y \notin BV(\lambda x.s) \cup FV(\lambda x.s)$$

Die reflexiv-transitive Hülle solcher α-Umbenennungen heißt *α-Äquivalenz*. Wir unterscheiden α-äquivalente Ausdrücke nicht. Vielmehr nehmen wir an, dass in einem Ausdruck alle gebundenen Variablen unterschiedliche Namen haben und dass Namen gebundener Variablen stets verschieden sind von Namen freier Variablen. Mithilfe von α-Umbenennung kann diese Konvention stets eingehalten werden.

Die klassische Reduktionsregel des Lambda-Kalküls ist die β-Reduktion, die die Funktionsanwendung auswertet:

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

Wenn $s_0 = (\lambda x.s)\ t \xrightarrow{\beta} s[t/x] = t_0$, so sagt man auch s_0 reduziert unmittelbar zu t_0. Allerdings reicht es nicht aus, nur β-Reduktionen auf oberster Ebene durchzuführen, man könnte dann z.B. den Ausdruck $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$ nicht weiter reduzieren. Zur Festlegung der operationalen Semantik muss man deshalb noch definieren, wo im Ausdruck die β-Reduktionen angewendet werden sollen. Betrachte z.B. den Ausdruck $((\lambda x.x\ x)((\lambda y.y)(\lambda z.z)))$. Er enthält zwei verschiedene Positionen, die man reduzieren könnte (die entsprechenden Unterausdrücke nennt man *Redex*): $\underline{((\lambda x.x\ x)((\lambda y.y)(\lambda z.z)))} \to ((\lambda y.y)(\lambda z.z))\ ((\lambda y.y)(\lambda z.z))$ oder $((\lambda x.x\ x)\underline{((\lambda y.y)(\lambda z.z)))} \to ((\lambda x.x\ x)(\lambda z.z))$.

Diese Festlegung nennt man auch Reduktionsstrategie. Wir führen mit der Call-by-Name- und der Call-by-Value-Auswertung zwei verschiedene Strategien ein.

6.1.1 Call-by-Name Auswertung

Die Call-by-Name-Auswertung sucht immer den am weitesten oben und am weitesten links stehenden Redex. Formal kann man die Call-by-Name-Auswertung mithilfe von Reduktionskontexten definieren.

Definition 6.1.1. *Call-by-Name-Reduktionskontexte R werden durch die folgende Grammatik definiert:*

$$R ::= [\cdot] \mid (R\ Expr)$$

Wenn $s_0 \xrightarrow{\beta} t_0$, dann ist $R[s_0] \xrightarrow{name} R[t_0]$ ein Call-by-Name-Reduktionsschritt.

Eine Eigenschaft der Call-by-Name-Reduktion ist, dass diese stets *deterministisch* ist, d.h. für einen Ausdruck s gibt es höchstens einen Ausdruck t, so dass $s \xrightarrow{name} t$. Es gibt auch Ausdrücke, für die keine Reduktion möglich ist. Dies sind zum einen Ausdrücke, bei denen die Auswertung auf eine freie Variable stößt (z.B. ist $(x\ (\lambda y.y))$ nicht reduzibel). Zum anderen sind dies Werte: Für den Call-by-Name-Lambda-Kalkül sind dies Abstraktionen. D.h. sobald wir eine Abstraktion mithilfe von $\xrightarrow{name}$-Reduktionen erreicht haben, ist die Auswertung beendet. Ausdrücke, die man so in eine Abstraktion überführen kann, *konvergieren* (oder *terminieren*). Wir definieren dies formal, wobei $\xrightarrow{name,+}$ die transitive Hülle von $\xrightarrow{name}$ und $\xrightarrow{name,*}$ die reflexiv-transitive Hülle von $\xrightarrow{name}$ sei[1].

Definition 6.1.2. *Ein Ausdruck s (call-by-name)* konvergiert *genau dann, wenn es eine Folge von Call-by-Name-Reduktionen gibt, die s in eine Abstraktion überführt, wir schreiben dann $s \downarrow_{name}$. D.h. $s \downarrow_{name}$ gdw. $\exists$Abstraktion $v : s \xrightarrow{name,*} v$. Falls s nicht konvergiert, so schreiben wir $s \uparrow_{name}$ und sagen s* divergiert.

Die Programmiersprache Haskell hat den Call-by-Name-Lambda-Kalkül als semantische Grundlage, wobei Implementierungen die verbesserte Strategie der Call-by-Need-Auswertung verwenden, die Doppelauswertungen von Argumenten vermeidet. Für die Call-by-Name-Strategie gilt die folgende Eigenschaft:

Satz 6.1.3. *Sei s ein Lambda-Ausdruck und s kann mit beliebigen β-Reduktionen (an beliebigen Positionen) in eine Abstraktion v überführt werden. Dann gilt $s \downarrow_{name}$.*

Diese Aussage zeigt, dass die Call-by-Name-Auswertung bezüglich der Terminierung eine optimale Strategie ist.

[1] Formal kann man definieren:

$$\begin{aligned}
\xrightarrow{name,0} &:= \{(s,s) \mid s \text{ ist Ausdruck des Lambda-Kalküls}\},\\
\xrightarrow{name,i+1} &:= \{(s,t) \mid s \xrightarrow{name} t' \text{ und } t' \xrightarrow{name,i} t\}\\
\xrightarrow{name,*} &:= \bigcup_{i\in\mathbb{N}_0} \xrightarrow{name,i}\\
\xrightarrow{name,+} &:= \bigcup_{i\in\mathbb{N},i>0} \xrightarrow{name,i}
\end{aligned}$$

6.1.2 Call-by-Value Auswertung

Wir betrachten eine weitere wichtige Auswertungstrategie für den Lambda-Kalkül. Die Call-by-Value-Auswertungsstrategie verlangt, dass eine β-Reduktion nur dann durchgeführt werden darf, wenn das Argument eine Abstraktion ist. Wir definieren hierfür die β_{cbv}-Reduktion als:

$$(\beta_{cbv}) \qquad (\lambda x.s)\ v \to s[v/x], \text{ wobei } v \text{ eine Abstraktion oder Variable}$$

Jede β_{cbv}-Reduktion ist auch eine β-Reduktion. Die Umkehrung gilt nicht. Wie bei der Call-by-Name-Reduktion, definieren wir einen Call-by-Value-Reduktionsschritt mithilfe von Reduktionskontexten. Diese müssen jedoch angepasst werden, damit Argumente vor dem Einsetzen ausgewertet werden.

Definition 6.1.4. *Call-by-value Reduktionsskontexte E werden durch die folgende Grammatik generiert:*

$$E ::= [\cdot] \mid (E\ Expr) \mid ((\lambda V.Expr)\ E)$$

Wenn $s_0 \xrightarrow{\beta_{cbv}} t_0$, dann ist $E[s_0] \xrightarrow{value} E[t_0]$ ein Call-by-Value Reduktionsschritt.

Auch die Call-by-Value-Auswertung ist deterministisch, also eindeutig. Werte sind ebenfalls Abstraktionen. Der Begriff der Konvergenz ergibt sich wie folgt:

Definition 6.1.5. *Ein Ausdruck s (call-by-value)* konvergiert *genau dann, wenn es eine Folge von Call-by-Value-Reduktionen gibt, die s in eine Abstraktion überführt, wir schreiben dann $s \downarrow_{value}$. D.h. $s \downarrow_{value}$ gdw. $\exists$Abstraktion $v : s \xrightarrow{value,*} v$. Falls s nicht konvergiert, so schreiben wir $s \uparrow_{value}$ und sagen s* divergiert.

Satz 6.1.3 zeigt sofort: $s \downarrow_{value} \implies s \downarrow_{name}$. Die Umkehrung gilt nicht, da es Ausdrücke gibt, die bei Call-by-Value-Auswertung divergieren, bei Call-by-Name-Auswertung jedoch konvergieren:

Beispiel 6.1.6. *Betrachte den Ausdruck $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$. Es lässt sich leicht nachprüfen, dass $\Omega \xrightarrow{name} \Omega$ als auch $\Omega \xrightarrow{value} \Omega$. Daraus folgt sofort, dass $\Omega \uparrow_{name}$ und $\Omega \uparrow_{value}$. Betrachte nun den Ausdruck $t := ((\lambda x.(\lambda y.y))\ \Omega)$. Dann gilt $t \xrightarrow{name} \lambda y.y$, d.h. $t \downarrow_{name}$. Da die Call-by-Value-Auswertung jedoch zunächst das Argument Ω auswerten muss, gilt $t \uparrow_{value}$.*

Trotz dieser eher schlechten Eigenschaft der Call-by-Value-Auswertung, spielt sie eine wichtige Rolle für Programmiersprachen. Durch die festgelegte Reihenfolge der Auswertung kann man direkte Seiteneffekte in Sprachen mit Call-by-Value-Auswertung zulassen. Einige wichtige Programmiersprachen mit Call-by-Value-Auswertung sind die strikten funktionalen Programmiersprachen ML (mit den Dialekten SML, OCaml), Scheme und Microsofts F#.

6.1.3 Gleichheit von Programmen

Bisher haben wir den Call-by-Name- und den Call-by-Value-Lambda-Kalkül vorgestellt, die Syntax und die jeweilige operationale Semantik definiert. Wir können damit Programme ausführen (d.h. auswerten) allerdings fehlt noch ein Begriff der Gleichheit von Programmen. Dieser

ist z.B. notwendig um nachzuprüfen, ob ein Compiler korrekt optimiert, d.h. ob ein ursprüngliches Programm und ein optimiertes Programm gleich sind.

Nach dem Leibnitzschen Prinzip sind zwei Dinge gleich, wenn sie die gleichen Eigenschaften bezüglich aller Eigenschaften besitzen. Dann sind die Dinge beliebig in jedem Kontext austauschbar. Für Programmkalküle kann man das so fassen: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt. Formaler ausgedrückt sind s und t gleich, wenn für alle Kontexte C: gilt $C[s]$ und $C[t]$ verhalten sich gleich. Hierbei fehlt noch ein Begriff dafür, welches Verhalten wir beobachten möchten. Für deterministische Sprachen reicht die Beobachtung der Terminierung, die wir bereits als Konvergenz für beide Kalküle definiert haben.

Wir definieren nach diesem Muster die *Kontextuelle Gleichheit*, wobei man zunächst eine Approximation definieren kann und die Gleichheit dann die Symmetrisierung der Approximation darstellt.

Definition 6.1.7 (Kontextuelle Approximation und Gleichheit)**.** *Für den Call-by-Name-Lambda-Kalkül definieren wir die* Kontextuelle Approximation $\leq_{c,name}$ *und die* Kontextuelle Gleichheit $\sim_{c,name}$ *als:*

- $s \leq_{c,name} t$ *gdw.* $\forall C : C[s]\downarrow_{name} \implies C[t]\downarrow_{name}$
- $s \sim_{c,name} t$ *gdw.* $s \leq_{c,name} t$ *und* $t \leq_{c,name} s$

Analog definieren wir für den Call-by-Value-Lambda-Kalkül die Kontextuelle Approximation $\leq_{c,value}$ *und die* Kontextuelle Gleichheit $\sim_{c,value}$ *als:*

- $s \leq_{c,value} t$ *gdw.* $\forall C : C[s]\downarrow_{value} \implies C[t]\downarrow_{value}$
- $s \sim_{c,value} t$ *gdw.* $s \leq_{c,value} t$ *und* $t \leq_{c,value} s$

Die kontextuelle Gleichheit kann als gröbste Gleichheit betrachtet werden (d.h. möglichst viele Programme sind gleich), die offensichtlich unterschiedliche Programme unterscheidet. Eine wichtige Eigenschaft der kontextuellen Gleichheit ist die folgende:

Satz 6.1.8. $\sim_{c,name}$ *und* $\sim_{c,value}$ *sind* Kongruenzen*, d.h. sie sind Äquivalenzelationen und kompatibel mit Kontexten, d.h.* $s \sim t \implies C[s] \sim C[t]$.

Beweis. Wir betrachten hier exemplarisch nur den Call-by-Name-Lambda-Kalkül. Reflexivität $s \sim_{c,name} s$ und Symmetrie $s \sim_{c,name} t \implies t \sim_{c,name} s$ gilt offensichtlich. Für die Transitivität sei $s \sim_{c,name} t$ und $t \sim_{c,name} r$. Sei C ein Kontext, sodass $C[s]\downarrow_{name}$. Dann folgt aus $s \sim_{c,name} t$ zunächst $C[t]\downarrow_{name}$ und aus $t \sim_{c,name} r$ folgt $C[r]\downarrow_{name}$. Genauso kann man schließen, dass aus $C[r]\downarrow_{name}$ auch $C[s]\downarrow_{name}$ folgt. Damit gilt $s \sim_{c,name} r$.

Für die Kongruenzeigenschaft, nehme $s \sim_{c,name} t$ an und sei C ein Kontext. Wir müssen $C[s] \sim_{c,name} C[t]$ zeigen. Sei C' ein Kontext, sodass $C'[C[s]]\downarrow_{name}$, dann folgt aus $s \sim_{c,name} t$ auch $C'[C[t]]\downarrow_{name}$ (benutze den Kontext $C'[C[\cdot]]$ in der Definition von $\sim_{c,name}$). Umgekehrt kann man auch $C'[C[t]]\downarrow_{name} \implies C'[C[s]]\downarrow_{name}$ genauso folgern. Da dies für jeden Kontext C' gilt, folgt $C[s] \sim_{c,name} C[t]$. □

Die Kongruenzeigenschaft erlaubt es u.a. Unterprogramme eines großen Programms zu transformieren (bzw. optimieren) und dabei ein gleiches Gesamtprogramm zu erhalten (unter der Voraussetzung, dass die lokale Transformation die kontextuelle Gleichheit erhält).

Die kontextuelle Gleichheit liefert einen allgemein anerkannten Begriff der Programmgleichheit. Ein Nachteil der Gleichheit ist, dass Gleichheitsbeweise im Allgemeinen schwierig sind, da man alle Kontexte betrachten muss. Im Allgemeinen ist kontextuelle Gleichheit unentscheidbar, da man mit der Frage, ob $s \sim_c \Omega$ gilt, dass Halteproblem lösen würde. Wir gehen nicht genauer auf die Beweistechniken ein. Es ist jedoch noch erwähnenswert, dass Ungleichheit i.A. leichter nachzuweisen ist, da man in diesem Fall „nur“ einen Kontext angeben muss, der Ausdrücke bezüglich ihrer Konvergenz unterscheidet.

Es lässt sich nachweisen, dass die (β)-Reduktion die Gleichheit im Call-by-Name-Lambda-Kalkül erhält, d.h.es gilt $(\beta) \subseteq \sim_{c,name}$. Für den Call-by-Value-Lambda-Kalkül gilt $(\beta_{cbv}) \subseteq \sim_{c,value}$ aber $(\beta) \not\subseteq \sim_{c,value}$, denn z.B. konvergiert $((\lambda x.(\lambda y.y))\ \Omega)$ im leeren Kontext unter Call-by-Value-Auswertung nicht, während $\lambda y.y$ sofort konvergiert.

Bezüglich der Gleichheiten in beiden Kalkülen gilt keinerlei Beziehung, d.h. $\sim_{c,name} \not\subseteq \sim_{c,value}$ und $\sim_{c,value} \not\subseteq \sim_{c,name}$. Die erste Aussage folgt sofort aufgrund der Korrektheit der (β)-Reduktion. Für die zweite Aussage kann man als Beispiel nachweisen, dass $((\lambda x.(\lambda y.y))\ \Omega) \sim_{c,value} \Omega$ während $((\lambda x.(\lambda y.y))\ \Omega) \not\sim_{c,name} \Omega$.

6.2 Ein Message-Passing-Modell: Der π-Kalkül

Während im Lambda-Kalkül sämtliche Funktionalitäten eines Programms, d.h. sowohl Kontrollstrukturen als auch Daten durch Abstraktionen und Anwendungen dargestellt werden, wird im π-Kalkül alles als Prozess dargestellt, wobei die einzige Operation die Kommunikation von Prozessen darstellt. Der π-Kalkül wurde von Robin Milner, Joachim Parrow und David Walker entwickelt und wird im Grunde nur als theoretisches Modell gesehen. D.h. zum echten Programmieren wird der π-Kalkül nie benutzt. Der π-Kalkül ist als Modell jedoch sehr erfolgreich, es gibt viele Varianten des Kalküls, die u.a. in der Bioinformatik Anwendungen haben.

6.2.1 Der synchrone π-Kalkül

Wir führen zunächst die Syntax des synchronen π-Kalküls ein, anschließend werden wir den asynchronen π-Kalkül beschreiben, der eine eingeschränktere Syntax besitzt. Wir schreiben ab jetzt einfach „π-Kalkül“, wenn aus dem Zusammenhang eindeutig ist, welche der beiden Varianten gemeint ist.

6.2.1.1 Syntax des synchronen π-Kalküls

Wir beschreiben eine Variante des synchronen π-Kalküls. Es gibt verschiedene andere Varianten, die beispielsweise Summen von Prozessen oder rekursive Prozessdefinitionen enthalten (der Leser sei hierfür auf die weiterführende Literatur verwiesen).

Sei $\mathcal{N}$ eine abzählbar unendliche Menge von *Namen*. Im π-Kalkül können Prozesse verschiedene *Aktionen* durchführen, die durch einen so genannten *Action-Präfix* π notiert werden:

$$\pi ::= x(y) \mid \overline{x}y \qquad \text{wobei } x, y \in \mathcal{N}$$

Die Syntax von π-Kalkül-Prozessen P ist gegeben durch die Grammatik:

$$P ::= \pi.P \mid P_1 \mid P_2 \mid !\,P \mid \mathbf{0} \mid \nu x.P \qquad \text{für } x \in \mathcal{N}$$

Wir beschreiben zunächst die Aktionen für Prozesse: Der Prozess $x(y).P$ *empfängt* den Namen y über den Kanal namens x und verhält sich anschließend wie P, nur dass y durch den empfangenen Namen in P ersetzt wird. Der Prozess $\overline{x}y.P$ *sendet* den Namen y über den Kanal namens x und verhält sich anschließend wie P.

Die weiteren Konstrukte für Prozesse lassen sich informal wie folgt erläutern:

- $\mathbf{0}$ ist der *inaktive Prozess*, der nichts mehr tut.
- $P_1 \mid P_2$ ist die *parallele Komposition* der beiden Prozesse P_1 und P_2. Die beiden Prozesse laufen nebenläufig ab und können über gemeinsame Namen (bzw. Kanäle) kommunizieren.
- $\nu z.P$ ist die *Restriktion*, die den Geltungsbereich des Namens z auf P beschränkt.
- $!\,P$ ist die *Replikation*, der Prozess P wird dadurch beliebig oft vervielfacht, d.h. man kann $!\,P$ als Abkürzung für $\underbrace{P \mid P \mid \ldots \mid P}_{\text{unendlich oft}}$ ansehen.

Damit gibt es im π-Kalkül zwei Konstrukte die Namen *binden* und damit auch *Geltungsbereiche* von Namen festlegen: Die Restriktion $\nu z.P$ bindet den Namen z mit Geltungsbereich P und für den Prozess mit Input-Präfix $x(y).P$ wird der Name y mit Geltungsbereich P eingeführt. Beachte, dass ein Output-Präfix $\overline{x}y$ weder x noch y bindet. Namen, die in keinem Geltungsbereich stehen werden als *freie Namen* bezeichnet, andere Namen sind *gebunden*. Wir definieren dies formal für Prozesse:

Für einen Prozess P ist die *Menge seiner freien Namen* $\mathsf{fn}(P)$ induktiv definiert durch die Gleichungen:

$$\begin{aligned}
\mathsf{fn}(x(y).P) &= \{x\} \cup (\mathsf{fn}(P) \setminus \{y\}) \\
\mathsf{fn}(\overline{x}y.P) &= \{x, y\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(P_1 \mid P_2) &= \mathsf{fn}(P_1) \cup \mathsf{fn}(P_2) \\
\mathsf{fn}(\mathbf{0}) &= \emptyset \\
\mathsf{fn}(\nu x.P) &= \mathsf{fn}(P) \setminus \{x\} \\
\mathsf{fn}(!\,P) &= \mathsf{fn}(P)
\end{aligned}$$

Die *Menge der gebundenen Namen* $\mathsf{bn}(P)$ eines Prozesses P ist induktiv definiert durch die Gleichungen:

$$\begin{aligned}
\mathsf{bn}(x(y).P) &= \{y\} \cup \mathsf{bn}(P) \\
\mathsf{bn}(\overline{x}y.P) &= \mathsf{bn}(P) \\
\mathsf{bn}(P_1 \mid P_2) &= \mathsf{bn}(P_1) \cup \mathsf{bn}(P_2) \\
\mathsf{bn}(\mathbf{0}) &= \emptyset \\
\mathsf{bn}(\nu x.P) &= \{x\} \cup \mathsf{bn}(P) \\
\mathsf{bn}(!\,P) &= \mathsf{bn}(P)
\end{aligned}$$

Mit $\mathsf{n}(P)$ bezeichnen wir alle Namen eines Prozesses (also $\mathsf{n}(P) := \mathsf{fn}(P) \cup \mathsf{bn}(P)$). Das Vorkommen eines Namens in einem Prozess ist *gebunden*, wenn er im Geltungsbereich des entsprechendes Binders steht, anderenfalls ist das Vorkommen *frei*. Für einen Prozess P ist $P[x/y]$ der Prozess, der aus P entsteht, indem alle freien Vorkommen des Namens y durch den Namen x ersetzt werden, wobei kein ungewolltes Einfangen von Namen passieren darf, d.h. ein vorher freies Vorkommen von y durch ein gebundenes Vorkommen von x ersetzt wird.

Das Einfangen von Namen kann verhindert werden, indem gebundene Namen *umbenannt* werden. Vor der Definition der Umbenennung definieren wir jedoch *Kontexte*: Ein *Prozesskontext* D ist wie ein Prozess, der an einer Position anstelle eines Prozesses ein Loch (welches wir mit $[\cdot]$ notieren) hat. Man kann Prozesskontexte auch formal mit der folgenden Grammatik definieren, wobei π und P wie oben definiert sind.

$$D ::= [\cdot] \mid \pi.D \mid D \mid P \mid P \mid D \mid \mathbf{!}\, D \mid \nu x.D \qquad \text{für } x \in \mathcal{N}$$

Definition 6.2.1 (Alpha-Äquivalenz)**.** *Eine* Umbenennung gebundener Namen *eines Prozesses P ist ein α-Umbenennungsschritt, der definiert ist als:*

$$\begin{array}{lcl} D[x(y).P] & \xrightarrow{\alpha} & D[x(z).P[z/y]] \textit{ falls } z \notin n(x(y).s) \\ D[\nu y.P] & \xrightarrow{\alpha} & D[\nu z.P'[z/y]] \textit{ falls } z \notin n(\nu y.P) \end{array}$$

Die Relation $=_\alpha$ ist die reflexiv-transitive Hülle *von $\xrightarrow{\alpha}$. Prozesse P_1, P_2 heißen* α-äquivalent*, wenn sie durch Umbenennungen gebundener Namen gleich gemacht werden können.*

Wir unterscheiden α-äquivalente Prozesse nicht, vielmehr verlangen wir das stets folgende Konvention gilt:

Konvention 6.2.2. *Für einen Prozess P nehmen wir stets an, dass seine gebundenen Namen paarweise verschieden sind und dass gebundene und freie Namen disjunkt sind.*

Es ist einfach diese Konvention einzuhalten, da sie mithilfe von Umbenennungen gebundener Namen stets erfüllt werden kann, und α-äquivalente Prozesse nicht unterschieden werden.

Im Folgenden definieren wir die Relation $\equiv$, die *strukturelle Kongruenz* für Prozesse des π-Kalküls.

α-äquivalente Kontexte werden *nicht* als gleich angesehen, d.h. z.B. sind $\nu x.[\cdot]$ und $\nu y.[\cdot]$ echt verschiedene Kontexte.

Eine binäre Relation $\mathcal{R}$ auf Prozessen ist *kompatibel mit Kontexten*, wenn für alle $(a, b) \in \mathcal{R}$ und alle Kontexte D gilt $(D[a], D[b]) \in \mathcal{R}$. Eine Äquivalenzrelation auf Prozessen, die kompatibel mit Kontexten ist, heißt *Kongruenz*.

Definition 6.2.3. *Die* strukturelle Kongruenz $\equiv$ *ist definiert als die kleinste Kongruenz auf Prozessen, die die folgenden Axiome erfüllt.*

$$\begin{array}{rcl} P & \equiv & Q, \textit{ falls } P \textit{ und } Q\ \alpha\textit{-äquivalent sind} \\ P_1 \mid (P_2 \mid P_3) & \equiv & (P_1 \mid P_2) \mid P_3 \\ P_1 \mid P_2 & \equiv & P_2 \mid P_1 \\ P \mid \mathbf{0} & \equiv & P \\ \nu z.\nu w.P & \equiv & \nu w.\nu z.P \\ \nu z.\mathbf{0} & \equiv & \mathbf{0} \\ \nu z.(P_1 \mid P_2) & \equiv & P_1 \mid \nu z.P_2, \textit{ falls } z \notin fn(P_1) \\ \mathbf{!}\, P & \equiv & P \mid \mathbf{!}\, P \end{array}$$

Umgekehrt ausgedrückt: Seien P_1 und P_2 Prozesse, die mithilfe der Axiome (angewendet auf Unterprozesse) und α-Umbenennungen gleich gemacht werden können, dann sind P_1 und P_2 strukturell kongruent (d.h. $P_1 \equiv P_2$)

Ein interessanter Fakt bezüglich der so definierten strukturellen Kongruenz ist, dass es bis heute nicht geklärt ist, ob diese entscheidbar ist, d.h. das Entscheidungsproblem ist, ob $P \equiv Q$ für gegebenes P und Q gilt. Es ist bekannt, dass das Problem zumindest EXPSPACE-schwer ist (siehe z.B. (Engelfriet & Gelsema, 2007; Schmidt-Schauß et al., 2013)).

6.2.1.2 Operationale Semantik des synchronen π-Kalküls

Wir werden im Folgenden die Reduktionssemantik für den π-Kalkül definieren. Diese ist als small-step Semantik anzusehen. Die Auswertungsregeln sind hierbei:

(Interact)	$x(y).P \mid \overline{x}v.Q \to P[v/y] \mid Q$
(Par)	$P \mid Q \to P' \mid Q$, falls $P \to P'$
(New)	$\nu x.P \to \nu x.P'$, falls $P \to P'$
(StructCongr)	$P \to P'$, falls $Q \to Q'$ und $P \equiv Q$ und $P' \equiv Q'$

Die wesentliche Reduktionsregel ist (Interact), da sie die Kommunikation zwischen zwei Prozessen realisiert. Die Regeln (Par) und (New) ermöglichen es, „unter“ der parallelen Komposition und unter ν-Bindern zu reduzieren. Die vierte Regel (StructCongr) erlaubt es den Prozess vor und nach der Reduktion mithilfe der strukturellen Kongruenz umzuformen. Man beachte, dass nicht unterhalb der Replikation oder unterhalb von Action-Präfixen reduziert wird. Das bedeutet z.B. das weder der Prozess $x(y).(u(v).\mathbf{0} \mid \overline{u}w.\mathbf{0})$ noch der Prozess $!\,(u(v).\mathbf{0} \mid \overline{u}w.\mathbf{0})$ direkt reduziert werden können. Allerdings kann der zweite Prozesse unter Zuhilfenahme der strukturellen Kongruenz reduziert werden.
Wir betrachten einige Beispiele:

Beispiel 6.2.4. *Der Prozess*

$$P \equiv \underbrace{x(y).\overline{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\overline{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$$

lässt sich wie folgt reduzieren. Zunächst kommunizieren P_1 und P_2 über den Kanal namens x, wobei P_2 an P_1 den Namen z verschickt:

$$P \to \underbrace{\overline{z}z.\mathbf{0}}_{P_1'} \mid \underbrace{\mathbf{0}}_{P_2'} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

Nun können P_1' und P_3 über den Kanal namens z kommunizieren, wobei P_1' den Namen z verschickt:

$$P' \to \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv P''$$

Der entstandene Prozess ist strukturell kongruent zu $\mathbf{0}$ und kann nicht weiter reduziert werden.

Beispiel 6.2.5. *Die Reduktion ist* nicht-deterministisch*, z.B. sind für den Prozess*

$$P \equiv x(y).\mathbf{0} \mid \overline{x}v.\mathbf{0} \mid x(z).\overline{z}w.\mathbf{0}$$

zwei Reduktionen möglich. Die ersten beiden Prozesse können interagieren:

$$P \to \mathbf{0} \mid \mathbf{0} \mid x(z).\overline{z}w.\mathbf{0},$$

aber auch die letzten beiden:

$$P \to x(y).\mathbf{0} \mid \mathbf{0} \mid \overline{v}w.\mathbf{0}$$

Beispiel 6.2.6. *ν-Binder verhindern Kommunikationsmöglichkeiten und ermöglichen dadurch lokale Kommunikation, die durch den Eingriff von außen geschützt ist. Z.B. ist für*

$$P \equiv \nu x.(x(y).\mathbf{0} \mid \overline{x}v.\mathbf{0}) \mid x(z).\overline{z}w.\mathbf{0}$$

nur eine Reduktion möglich, da der Name x im letzten Prozess nicht im Bindungsbereich des ν-Binders ist und damit als verschieden vom Namen x in den vorderen Prozessen anzusehen ist. Eine korrekte α-Umbenennung von P, die zur Erfüllung von Konvention 6.2.2 führt, macht das deutlich:

$$P \equiv \nu x'.(x'(y).\mathbf{0} \mid \overline{x'}v.\mathbf{0}) \mid x(z).\overline{z}w.\mathbf{0}$$

Nun ist offensichtlich, dass der letzte Prozess nicht mit den anderen Prozessen kommunizieren kann.

*Es sei aber auch zu erwähnen, dass lokale Namen ihren Bindungsbereich durch Kommunikation verlassen können. Z.B. sei $P \equiv x(y).P_1 \mid \nu z.\overline{x}z.P_2$, wobei z nicht in P_1 vorkommt. Durch Anwenden von (*Interact*) wird der scheinbar nur für P_2 bekannte Namen z auch für P_1 bekannt, denn $P \to \nu z.(P_1[z/y] \mid P_2)$. Dieses Phänomen wird auch als Extrusion bezeichnet.*

Beispiel 6.2.7. *Es gibt Prozesse die unendlich oft kommunizieren können. Ein einfaches Beispiel ist der Prozess*

$$P = \mathbf{!}\,(\overline{x}v.\mathbf{0}) \mid \mathbf{!}\,(x(z).\mathbf{0})$$

Wir zeigen den Anfang der unendlich langen Reduktionsfolge, wobei wir einige Umformungen bezüglich struktureller Kongruenz explizit machen:

$$\begin{aligned} P &\equiv \mathbf{!}\,(\overline{x}v.\mathbf{0}) \mid \overline{x}v.\mathbf{0} \mid x(z).\mathbf{0} \mid \mathbf{!}\,(x(z).\mathbf{0}) \\ &\to \mathbf{!}\,(\overline{x}v.\mathbf{0}) \mid \mathbf{0} \mid \mathbf{0} \mid \mathbf{!}\,(x(z).\mathbf{0}) \equiv \mathbf{!}\,\overline{x}v.\mathbf{0} \mid \mathbf{!}\,(x(z).\mathbf{0}) \equiv P \end{aligned}$$

6.2.1.3 Turing-Mächtigkeit des synchronen π-Kalküls

In diesem Abschnitt illustrieren wir, dass der synchrone π-Kalkül Turing-mächtig ist, indem wir zeigen, wie der Lambda-Kalkül in den π-Kalkül kodiert werden kann. Wir verzichten auf den vollständigen Beweis der Korrektheit dieser Kodierung (dieser ist in (Milner, 1992) nachzulesen).

Der Call-by-Name-Lambda-Kalkül wird in den π-Kalkül mithilfe der Abbildung $[\![\cdot]\!]$ kodiert, wobei für einen Lambda-Ausdruck s, die Übersetzung $[\![s]\!]$ noch kein π-Kalkül Prozess ist, sondern eine Funktion, die einen Namen erwartet und dann einen Prozess liefert, d.h. z.B. ist $[\![s]\!]u$ für einen Namen u ein Prozess. Der Name u ist dabei der Kanalname über den der zu s zugehörige Prozess seine Argumente empfängt.

Die folgenden Gleichungen geben an, wie alle Konstrukte des Lambda-Kalküls zu übersetzen sind. D.h. es sind drei Fälle zu definieren: Die Übersetzung von Variablen, von Abstraktionen und die Übersetzung von Anwendungen.

Eine Abstraktion $\lambda x.s$ wird so übersetzt, dass sie über den zusätzlichen Namen (also u) zwei Namen empfängt: Als erstes den Namen seines Arguments also x und anschließend den Namen, den s benutzen soll, um Argumente zu empfangen:

$$[\![\lambda x.s]\!]u := u(x).u(v).[\![s]\!]v$$

Eine Variable x wird übersetzt in einen Prozess, der über den Kanal namens x den durch die Übersetzung eingeführten Namen u versendet:

$$[\![x]\!]u := \overline{x}u.\mathbf{0}$$

Die Anwendung $(s\ t)$ wird so übersetzt, dass das Argument t quasi wie eine Bindung $x = t$ als eigener Prozess erzeugt wird. Die Anforderung an das Argument geschieht dann über x. Jedes Mal, wenn x angefordert wird, erhält das Argument einen Kanalnamen über welchen er seine Argumente erhält. Da das Argument t evtl. öfter innerhalb von s benötigt wird, wird dieser Prozess repliziert. Wir definieren hier für die Abkürzung:

$$\langle x = t\rangle := \mathbf{!}\,(x(w).[\![t]\!]w)$$

Die Anwendung $(s\ t)$ wird nun folgendermaßen übersetzt, wobei x ein neuer Name ist, d.h. nicht in t vorkommt:

$$[\![s\ t]\!]u := \nu v.([\![s]\!]v \mid \nu x.\overline{v}x.\overline{v}u.\langle x = t\rangle)$$

Die Übersetzung ist nicht ganz einfach im Detail nachzuvollziehen, da einige Kommunikation über Kanäle statt findet. Wir demonstrieren die Übersetzung anhand einiger Beispiele:

Betrachte den Lambda-Ausdruck $(\lambda x.x)\ t$. Die Übersetzung (wobei u frei gewählt werden kann) ergibt:

$$\begin{array}{rcl} [\![(\lambda x.x)\ t]\!]u & = & \nu v.([\![(\lambda x.x)]\!]v \mid \nu y.\overline{v}y.\overline{v}u.\langle y = t\rangle) \\ & = & \nu v.(v(x).v(w).[\![x]\!]w \mid \nu y.\overline{v}y.\overline{v}u.\langle y = t\rangle) \\ & = & \nu v.(v(x).v(w).\overline{x}w.\mathbf{0} \mid \nu y.\overline{v}y.\overline{v}u.\langle y = t\rangle) \end{array}$$

Jetzt kann man wie folgt reduzieren:

$$\begin{array}{rl} & \nu v.(v(x).v(w).\overline{x}w.\mathbf{0} \mid \nu y.\overline{v}y.\overline{v}u.\langle y = t\rangle) \\ \equiv & \nu y.\nu v.(v(x).v(w).\overline{x}w.\mathbf{0} \mid \overline{v}y.\overline{v}u.\langle y = t\rangle) \\ \to & \nu y.\nu v.(v(w).\overline{y}w.\mathbf{0} \mid \overline{v}u.\langle y = t\rangle) \\ \to & \nu y.\nu v.(\overline{y}u.\mathbf{0} \mid \langle y = t\rangle) \end{array}$$

Nach Entfaltung der Abkürzung für $\langle y = t\rangle$ und einer weiteren Reduktion ergibt dies:

$$\begin{array}{rcl} \nu y.\nu v.(\overline{y}u.\mathbf{0} \mid \langle y = t\rangle) & = & \nu y.\nu v.(\overline{y}u.\mathbf{0} \mid \mathbf{!}\,(y(w).[\![t]\!]w)) \\ & \equiv & \nu y.\nu v.(\overline{y}u.\mathbf{0} \mid (y(w).[\![t]\!]w) \mid \mathbf{!}\,(y(w).[\![t]\!]w)) \\ & \to & \nu y.\nu v.(\mathbf{0} \mid ([\![t]\!]u) \mid \mathbf{!}\,(y(w).[\![t]\!]w)) \\ & \equiv & [\![t]\!]u \mid \nu y.(\mathbf{!}\,(y(w).[\![t]\!]w)) \end{array}$$

Interpretiert man den Teil-Prozess $\nu y.(\mathbf{!}\,(y(w).[\![t]\!]w))$, so dass er entfernt werden kann, da kein anderer Prozess mehr mit ihm kommunizieren kann (y ist unbekannt für t). So ist das Ergebnis gerade das vom Call-by-Name-Lambda-Kalkül erwartete, denn $(\lambda x.x)\ t \xrightarrow{\beta} t$.

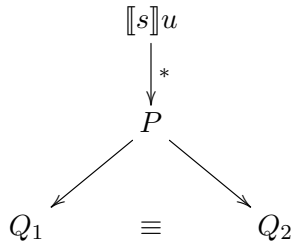
In (Milner, 1992) wurde gezeigt:

Satz 6.2.8. *Sei s ein geschlossener Lambda-Ausdruck, dann gilt eine der folgenden Bedingungen:*

1. $s \downarrow_{name}$ *und* $[\![s]\!]u \xrightarrow{*} P$*, sodass* P *irreduzibel ist.*
2. $s \uparrow_{name}$ *und es gibt kein* P*, so dass* $[\![s]\!]u \xrightarrow{*} P$*, wobei* P *irreduzibel ist.*

Dieser Satz zeigt, dass die Call-by-Name-Auswertung von Lambda-Ausdrücken im π-Kalkül bezüglich der Übersetzung $[\![\cdot]\!]$ simuliert werden kann. Die Übersetzung $[\![\cdot]\!]$ erfüllt noch eine weitere Bedingung:

Lemma 6.2.9. *Sei* s *ein geschlossener Lambda-Ausdruck, dann ist* $[\![s]\!]u$ deterministisch bzgl. $\equiv$*, d.h. falls* $[\![s]\!]u \xrightarrow{*} P$ *und* $P \to Q_1$ *als auch* $P \to Q_2$*, dann gilt stets* $Q_1 \equiv Q_2$*. Illustriert werden kann die Aussage durch das folgende Diagramm*



Dieses Lemma verstärkt Satz 6.2.8, da es besagt, dass Lambda-Ausdrücke in *deterministische* π-Kalkül Prozesse übersetzt werden. Da der Lambda-Kalkül Turing-mächtig ist, folgt:

Theorem 6.2.10. *Der synchrone* π*-Kalkül ist Turing-mächtig.*

Beachte, dass dies noch nichts über die Gleichheitstheorien des Lambda-Kalküls und des π-Kalküls aussagt. In (Milner, 1992) wird hier ein stärkerer Zusammenhang bewiesen. Wir verzichten darauf, da wir zum einen noch keinen Gleichheitsbegriff für den π-Kalkül definiert haben und es zum anderen verschiedene Gleichheitsbegriffe für den π-Kalkül gibt, auf die wir erster später eingehen werden.

6.2.2 Der asynchrone π-Kalkül

Im zuvor vorgestellten synchronen π-Kalkül findet sämtliche Kommunikation synchron statt, da der Nachrichtenaustausch (das Empfangen und Senden des Namens) direkt zwischen dem sendenden und empfangenden Prozess in einem Schritt vollzogen wird. Im asynchronen π-Kalkül wird dies geändert. Der wesentliche syntaktische Unterschied zwischen dem synchronen π-Kalkül und dem asynchronen π-Kalkül besteht darin, dass im asynchronen π-Kalkül dem Output-Präfix nur der inaktive Prozess $\mathbf{0}$ folgen darf, d.h. die Syntax von Prozessen P des asynchronen π-Kalküls ist durch die folgende Grammatik definiert:

$$P \quad ::= \quad \mathbf{0} \mid \overline{x}y.\mathbf{0} \mid x(y).P \mid P_1 \mid P_2 \mid \nu z.P \mid !P$$

Der asynchrone Charakter entsteht nun durch die Sichtweise, dass sämtliche Prozesse der Form $\overline{x}y.\mathbf{0}$ als gesendete Nachrichten aufzufassen sind und im Raum herumschwirren und nach beliebig langer Wartezeit von einem empfangenden Prozess empfangen werden können. Die Reduktionsregeln werden nicht verändert. Alleinig die Sichtweise, dass für die Regel (Interact) zwei Prozesse miteinander kommunizieren, wird verworfen, da der „sendende Prozess“ kein

beliebiger Prozess mehr sein kann, sondern ein Prozess, der nur aus der zu sendenden Nachricht und dem zugehörigen Kanal besteht (u.a. deswegen werden oft die inaktiven Prozesse $\mathbf{0}$ nicht explizit notiert. Man findet oft die Schreibweise $\overline{x}y$ anstelle von $\overline{x}y.\mathbf{0}$). Diese Sichtweise lässt auch eine Trennung der Prozesse in versendete Nachrichten (die sich gerade im Nachrichtenmedium befinden) und den restlichen Prozessen zu.

6.2.2.1 Kodierbarkeit des synchronen π-Kalküls in den asynchronen π-Kalkül

Um den synchronen π-Kalkül in den asynchronen zu kodieren, muss synchrone Kommunikation im asynchronen π-Kalkül simuliert werden. Die Idee dabei ist, dass der Sender nach dem Empfang seiner verschickten Nachricht eine Bestätigung vom Empfänger erhält. Sei $P' \mid Q' \equiv (\overline{x}z.P \mid x(y).Q)$ ein Prozess des synchronen π-Kalküls. Wir nennen P' den Sender, Q' den Empfänger. Dann sind P und Q solange blockiert, bis die Interaktion zwischen beiden Prozessen stattgefunden hat. Ein erster Ansatz $\overline{x}z.P$ in den asynchronen π-Kalkül zu übersetzen, ist $\overline{x}z.\mathbf{0} \mid P$. Allerdings kann nun P reduziert werden, bevor die Kommunikation mit Q' stattgefunden hat. Eine nächste Idee ist es, den Sender P zu schützen, so dass er eine Bestätigung von Q erwartet, bevor er weiterrechnen darf. Analog dazu muss der Empfänger Q diese Bestätigung verschicken, nachdem er die eigentliche Nachricht empfangen hat. Diese ergibt als Kodierungen:

$$\text{Sender} \quad S = \overline{x}z.\mathbf{0} \mid u(v).P \quad \text{Empfänger} \quad E = x(y).(\overline{u}v.\mathbf{0} \mid Q)$$

wobei v nicht frei in P vorkommt, und u nicht frei in Q vorkommt.

Wir betrachten die Auswertung von $S \mid E$ im asynchronen π-Kalkül:

$$(\overline{x}z.\mathbf{0} \mid u(v).P) \mid x(y).(\overline{u}v.\mathbf{0} \mid Q) \to u(v).P \mid \overline{u}v.\mathbf{0} \mid Q[z/y] \to P \mid Q[z/x]$$

Diese Lösung hat allerdings immer noch Schwächen: Die Kodierung ist nicht unabhängig vom Kontext: Wenn die Übersetzung auf einen Prozess $D[P' \mid Q']$ angewendet wird, ist nicht sichergestellt, dass andere Prozesse aus D, über den Kanal u mit dem Sender oder dem Empfänger kommunizieren, und somit die Synchronisation stören. Als Beispiel sei $D = u(w).\mathbf{0} \mid [\cdot]$, dann kann $D[S \mid E]$ wie folgt reduzieren:

$$\begin{aligned} & u(w).\mathbf{0} \mid (\overline{x}z.\mathbf{0} \mid u(v).P) \mid x(y).(\overline{u}v.\mathbf{0} \mid Q) \\ \to \quad & u(w).\mathbf{0} \mid (u(v).P) \mid \overline{u}v.\mathbf{0} \mid Q[z/y] \\ \to \quad & u(v).P \mid Q[z/y] \end{aligned}$$

In diesem Fall ist P immer noch blockiert, da die von E gesendete Bestätigung vom falschen Prozess empfangen wurde. Deshalb sollte zuerst ein *privater* Kanalname zwischen Sender und Empfänger ausgetauscht werden, über den alleinig diese beiden Prozesse kommunizieren können. Betrachtet man einen Sender der Form $\nu u.(\overline{x}u.\mathbf{0} \mid u(v)P)$, so kann man leicht nachprüfen, dass der Unterprozess $u(v)P$ erst dann weiter reduzieren kann, wenn die Nachricht u über x aus $\overline{x}u.\mathbf{0}$ versendet wurde, denn vorher ist der Kanalname u nicht sichtbar für andere Prozesse. Hier wird das Phänomen der Extrusion genutzt, um u nach außen sichtbar zu machen. Hierbei ist offensichtlich, dass nur der Prozess, der u empfängt, mit P über u kommunizieren kann, da nur dieser Prozess den Namen u kennen kann.

Die Übersetzung $[\![\cdot]\!]_\pi$ von synchronen in asynchrone π-Kalkül Prozesse ist angelehnt an (Boudol, 1992) und wie folgt definiert:

$$\begin{array}{lcl}
[\![\mathbf{0}]\!]_\pi & = & \mathbf{0} \\
[\![\overline{x}y.P]\!]_\pi & = & \nu u.(\overline{x}u.\mathbf{0} \mid u(v).(\overline{v}y.\mathbf{0} \mid [\![P]\!]_\pi)), \text{ wobei } u, v \notin \mathsf{fn}(P) \\
[\![x(y).P]\!]_\pi & = & x(u).\nu v.(\overline{u}v.\mathbf{0} \mid v(y).[\![P]\!]_\pi), \text{ wobei } u, v \notin \mathsf{fn}(P) \\
[\![P \mid Q]\!]_\pi & = & [\![P]\!]_\pi \mid [\![Q]\!]_\pi \\
[\![!\, P]\!]_\pi & = & !\, [\![P]\!]_\pi \\
[\![\nu x.P]\!]_\pi & = & \nu x.[\![P]\!]_\pi
\end{array}$$

Wir betrachten nun die Auswertung der synchronen Kommunikation, d.h. die Auswertung von $[\![\overline{x}z.P \mid x(y).Q]\!]_\pi$:

$$\begin{array}{ll}
 & [\![\overline{x}z.P \mid x(y).Q]\!]_\pi \\
= & \nu u.(\overline{x}u.\mathbf{0} \mid u(v).(\overline{v}z.\mathbf{0} \mid [\![P]\!]_\pi)) \mid x(u').\nu v'.(\overline{u'}v'.\mathbf{0} \mid v'(y).[\![Q]\!]_\pi) \\
\to & \nu u.(\mathbf{0} \mid u(v).(\overline{v}z.\mathbf{0} \mid [\![P]\!]_\pi) \mid \nu v'.(\overline{u}v'.\mathbf{0} \mid v'(y).[\![Q]\!]_\pi)) \\
\to & \nu v'.(\nu u.(\mathbf{0} \mid (\overline{v'}z.\mathbf{0}) \mid [\![P]\!]_\pi \mid \mathbf{0} \mid v'(y).[\![Q]\!]_\pi)) \\
\to & \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid [\![P]\!]_\pi \mid \mathbf{0} \mid [\![Q]\!]_\pi[z/y])) \\
\equiv & [\![P]\!]_\pi \mid [\![Q]\!]_\pi[z/y]
\end{array}$$

Beachte, dass der Sender weiter rechnen könnte, nachdem er vom Empfänger die Bestätigung erhalten hat, dass er mit ihm kommunizieren möchte, aber bevor der Name z wirklich verschickt wurde. Dies macht aber nichts, da zu diesem Zeitpunkt die Verbindung der beiden Prozesse stattgefunden hat, d.h. es ist sichergestellt, wer mit wem kommuniziert. Kein anderer Prozess als der Empfänger kann das z empfangen.

In (Boudol, 1992) wurde u.a. gezeigt, dass die Kodierung $[\![\cdot]\!]_\pi$ Konvergenz-erhaltend ist. Hierfür müssen wir jedoch zunächst definieren, welchen Konvergenzbegriff Boudol verwendet (für den π-Kalkül gibt es auch hierfür verschiedene Definitionen). Boudol betrachtet im Wesentlichen nur geschlossene Prozesse, wobei er auch hierfür einen speziellen Begriff verwendet:

Definition 6.2.11. *Ein* Ausgabe-geschlossener *Prozess des synchronen bzw. asynchronen π-Kalküls ist ein Prozess, so dass für jeden Output-Präfix $\overline{x}y$ gilt: Die Namen x und y sind durch Binder gebunden.*

Die Antworten bzw. Ergebnisse einer Reduktion sind so genannte Eingabe-Antworten:

Definition 6.2.12. *Ein Prozess P ist eine* Eingabe-Antwort*, wenn gilt* $P \equiv \nu x_1 \ldots \nu x_n.x(y).P' \mid Q$

Eingabe-Antworten sind also Prozesse, die noch eine Eingabe von „außen“ durchführen könnten. Dies entspricht der Intuition einer Abstraktion im Lambda-Kalkül, da diese noch Eingaben (Argumente) verarbeiten kann.

Nun können wir Boudol's Resultat für die Konvergenz formulieren:

Satz 6.2.13. *Sei P ein Ausgabe-geschlossener Prozess des synchronen π-Kalküls. Dann gilt:*

$$\begin{array}{c}
\exists \textit{ Eingabe-Antwort } P' \textit{: } P \xrightarrow{*} P' \\
\textit{genau dann, wenn} \\
\exists \textit{ Eingabe-Antwort } P'' \textit{: } [\![P]\!]_\pi \xrightarrow{*} P''
\end{array}$$

Das Resultat besagt somit, dass das Erreichen einer Eingabe-Antwort (das ist der von Boudol verwendete Konvergenzbegriff!) bezüglich der Übersetzung $[\![\cdot]\!]_\pi$ im synchronen und asynchronen π-Kalkül (für Ausgabe-geschlossene) Prozesse zusammenfällt.

Bemerkung 6.2.14. *Beachte, dass dieses Resultat, also die Kodierbarkeit des synchronen π-Kalküls in den asynchronen π-Kalkül, nicht mehr stimmt, wenn man zum synchronen π-Kalkül Summen hinzufügt (siehe nächster Abschnitt) und Anforderungen an die Übersetzung stellt:*

- *Die Übersetzung sollte kompositional sein und die parallele Komposition als parallele Komposition übersetzen, d.h. $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$*
- *Die Übersetzung sollte wohl-verhaltend bezüglich Umbenennungen sein, d.h. für Umbenennungen σ soll gelten $[\![\sigma(P)]\!] = \sigma([\![P]\!])$.*

Unter diesen Annahmen findet man keine sinnvolle (d.h. Semantik-erhaltende) Übersetzung (siehe (Palamidessi, 1997; Palamidessi, 2003)).

6.2.3 Erweiterungen und Varianten des π-Kalküls

In diesem Abschnitt stellen wir einige Varianten und Erweiterungen des synchronen π-Kalküls vor.

6.2.3.1 Nichtdeterministische Auswahl

In vielen Varianten des π-Kalküls wird die nicht-deterministische Auswahl als zusätzliches Konstrukt hinzugefügt. $P + Q$ entspricht hierbei gerade einem Prozess, der sich entweder wie Prozess P oder wie Prozess Q verhalten kann. Oft ist diese nichtdeterministische Auswahl jedoch beschränkt, so dass die Wahl zwischen P und Q gleichzeitig mit dem Empfangen oder Senden einer Nachricht geschehen muss. Wir folgen (Milner, 1999) und erlauben als Summanden der Summe nur Prozesse mit Präfix: Die Syntax von Prozessen des *synchronen π-Kalküls mit Summen* ist durch die folgende Grammatik festgelegt, wobei es einen neuen Präfix gibt: τ, er steht für die nicht-beobachtbare Aktion.

$$\begin{array}{lcll} \pi & ::= & x(y) \mid \overline{x}y \mid \tau & \text{wobei } x, y \in \mathcal{N} \\ P & ::= & \pi.P \mid P_1 \mid P_2 \mid \mathbf{!}\, P \mid \mathbf{0} \mid \nu x.P \mid \sum_i \pi.P_i & \text{für } x \in \mathcal{N} \end{array}$$

Wir verwenden das Summenzeichen und $+$ je nach belieben. Die Definitionen für gebundene und freie Namen und für die strukturelle Kongruenz müssen entsprechend angepasst werden. Dabei werden Summen als assoziativ und kommutativ angesehen. Zusätzlich gelte die Regel $P + P = P$.

Die Reduktionsregeln müssen entsprechend der Summen und bzgl. des neues Präfixes τ angepasst werden:

(Silent)	$\tau.P \to P$
(SilentSum)	$\tau.P + M \to P$
(InteractSum)	$x(y).P + M \mid \overline{x}v.Q + N \to P[v/y] \mid Q$
(Interact)	$x(y).P \mid \overline{x}v.Q \to P[v/y] \mid Q$
(Par)	$P \mid Q \to P' \mid Q$, falls $P \to P'$
(New)	$\nu x.P \to \nu x.P'$, falls $P \to P'$
(StructCongr)	$P \to P'$, falls $Q \to Q'$ und $P \equiv Q$ und $P' \equiv Q'$

Beachte, dass innerhalb einer Summe nicht reduziert wird. Z.B. kann man $(x(y).P + \overline{x}z.Q)$ nicht reduzieren, da die Summanden ja Alternativen darstellen.

Der τ-Präfix fügt im Grunde eine nicht-beobachtbare Aktion hinzu. Er wird durch die Reduktion quasi entfernt. Er wurde hier nur aus dem Grund in die Syntax aufgenommen, um ihn als Präfix in der Summe verwenden zu können, denn z.B. ist $(\mathbf{0} + \pi.P)$ ein syntaktisch nicht erlaubtes Konstrukt, wohin gegen $(\tau.\mathbf{0} + \pi.P)$ erlaubt ist (wenn P syntaktisch korrekt ist).

Unter Verwendung von τ als Präfix kann auch eine beliebige nichtdeterministische Auswahl kodiert werden: Sei $P \oplus Q := \tau.P + \tau.Q$. Dann verhält sich die Auswertung von $P \oplus Q$ so, dass sie sich zwischen P und Q nichtdeterministisch entscheidet, d.h. $P \oplus Q \to P$ als auch $P \oplus Q \to Q$.

6.2.3.2 Polyadischer π-Kalkül

Die bisher vorgestellten Varianten des π-Kalküls heißen auch *monadischer* π-Kalkül. Im Gegensatz zum monadischen π-Kalkül erlaubt der *polyadische* π-Kalkül das Versenden und Empfangen *mehrerer* Namen gleichzeitig, d.h. anstelle eines Namens wird ein Tupel von Namen ausgetauscht. Sei $\overrightarrow{x}$ ein Tupel von Namen (d.h. $\overrightarrow{x} = (x_1, \ldots, x_n)$) dann bezeichnen wir mit $|\overrightarrow{x}|$ die Länge des Tupels (also $|\overrightarrow{x}| = n$), falls $\overrightarrow{x} = (x_1, \ldots, x_n)$.

Die Action-Präfixe π im *polyadischen* π-Kalkül sind durch die Grammatik

$$\pi ::= x(\overrightarrow{y}) \mid \overline{x}\,\overrightarrow{z} \text{ wobei } x, y_i, z_i \in \mathcal{N}$$

definiert (auch hier kann man für den polyadischen Kalkül mit Summen, den τ-Präfix noch hinzufügen).

Die Interact-Regel für die Definition der Reduktion wird wie folgt angepasst:

$$\text{(Interact)}\ \ x(\overrightarrow{y}).P \mid \overline{x}\,\overrightarrow{z}.Q \to P[\overrightarrow{z}/\overrightarrow{y}] \mid Q \text{ falls } |\overrightarrow{y}| = |\overrightarrow{z}|$$

D.h. anstelle eines Namens wird ein ganzes Tupel versendet. Die Schreibweise $P[\overrightarrow{z}/\overrightarrow{y}]$ meint hierbei den Prozess $P[z_1/y_1, \ldots, z_n, y_n]$ wenn n die Länge der beiden Tupel ist.

Die Interact-Regel fordert, dass das Ein- und Ausgabetupel gleiche Länge haben. Damit wäre es immer noch möglich Prozess zu formen, bei denen der gleiche Kanal für verschiedene Längen benutzt wird. Deshalb wird dies normalerweise durch so genannte *Sorten* verboten. Dies wirkt ähnlich wie ein Typsystem und schreibt für jeden Kanalnamen fest, für welche Tupellänge der Kanal ausschließlich benutzt werden kann. Wir gehen darauf nicht weiter ein, nehmen jedoch im Folgenden an, dass über jeden Kanal stets nur Tupel gleicher Länge versendet werden.

Man kann den polyadischen π-Kalkül in den monadischen π-Kalkül übersetzen. Die Idee dabei ist es, die Namen aus dem versendeten Tupel $(z_1, \ldots, z_n)$ nacheinander (also sequentiell statt parallel) zu verschicken. D.h. ein Prozess mit einem (polyadischen) Input-Präfix $x(y_1, \ldots, y_n).P$ wird übersetzt als $x(y_1). \ldots x(y_n).P'$ wobei P' die Übersetzung von P ist, und ein polyadischer Prozess mit Output-Präfix $\overline{x}(z_1, \ldots, z_n).Q$ wird übersetzt in $\overline{x}z_1. \ldots \overline{x}z_n.Q'$, wobei Q' die Übersetzung von Q ist.

Leider funktioniert diese Übersetzung jedoch nicht, wenn mehr als zwei Prozesse den gleichen Kommunikationskanal benutzen, wie das folgende Beispiel zeigt. Sei $P := \overline{x}(z_1, z_2).\mathbf{0} \mid \overline{x}(z_3, z_4).\mathbf{0} \mid x(y_1, y_2).\overline{y_1}y_2.\mathbf{0}$. Dann gibt es zwei Möglichkeiten P zu reduzieren:

- $P \to \mathbf{0} \mid \overline{x}(z_3, z_4).\mathbf{0} \mid \overline{z_1}z_2.\mathbf{0}$
- $P \to \overline{x}(z_1, z_2).\mathbf{0} \mid \overline{z_3}z_4.\mathbf{0}$

Übersetzt man den Prozess P mit obiger Idee, dann erhält man $Q := \overline{x}z_1.\overline{x}z_2.\mathbf{0} \mid \overline{x}z_3.\overline{x}z_4.\mathbf{0} \mid x(y_1).x(y_2).\overline{y_1}y_2.\mathbf{0}$. Für den übersetzten Prozess gibt es vier Resultate nach der Reduktion:

- $Q \xrightarrow{*} \mathbf{0} \mid \overline{x}z_3.\overline{x}z_4.\mathbf{0} \mid \overline{z_1}z_2.\mathbf{0}$
- $Q \xrightarrow{*} \overline{x}z_1.\overline{x}z_2.\mathbf{0} \mid \mathbf{0} \mid \overline{z_3}z_4.\mathbf{0}$
- $Q \xrightarrow{*} \overline{x}z_2.\mathbf{0} \mid \overline{x}z_4.\mathbf{0} \mid \overline{z_1}z_3.\mathbf{0}$
- $Q \xrightarrow{*} \overline{x}z_2.\mathbf{0} \mid \overline{x}z_4.\mathbf{0} \mid \overline{z_3}z_1.\mathbf{0}$

Die letzten beiden Resultate entstehen dadurch, dass die sequentialisierten Operationen durchmischt wurden. Die Übersetzung kann dadurch völlig anderes Verhalten einführen, als es der ursprüngliche Prozess hatte. Der Fallstrick bei der Übersetzung ist, dass die Übertragung der einzelnen Namen des Tupels nicht alle beim *selben* Empfänger ankommen müssen bzw. nicht alle vom *selben* Sender stammen. Dies kann dadurch korrigiert werden, dass Sender und Empfänger zunächst einen privaten Kanalnamen austauschen über den die eigentlich Kommunikation läuft. Nachdem der Name ausgetauscht wurde, kann kein anderer Prozess bei der Kommunikation dazwischen funken.

Die Übersetzung $[\![\cdot]\!]_{poly}$ des polyadischen in den monadischen π-Kalkül ist definiert als.

$$\begin{array}{lcl}
[\![x(z_1, \ldots, z_n).P]\!]_{poly} & = & x(w).w(z_1). \ldots .w(z_n).[\![P]\!]_{poly} \\
[\![\overline{x}(z_1, \ldots, z_n).P]\!]_{poly} & = & \nu v.\overline{x}v.\overline{v}z_1. \ldots .\overline{v}z_n.[\![P]\!]_{poly} \\
[\![\nu x.P]\!]_{poly} & = & \nu x.[\![P]\!]_{poly} \\
[\![P \mid Q]\!]_{poly} & = & [\![P]\!]_{poly} \mid [\![Q]\!]_{poly} \\
[\![!\, P]\!]_{poly} & = & !\, [\![P]\!]_{poly} \\
[\![\mathbf{0}]\!]_{poly} & = & \mathbf{0}
\end{array}$$

Diese Übersetzung verhält sich korrekt.

6.2.3.3 Rekursive Definitionen

Oft werden anstelle der Replikation rekursive Prozessdefinitionen verwendet, d.h. die Syntax für Prozesse wird geändert: Replikation wird gestrichen und so genannten *Konstantenanwendungen* werden hinzugefügt:

$$P := \ldots \mid \cancel{!P} \mid \ldots \mid A(x_1, \ldots, x_n) \text{ wobei } x_i \in \mathcal{N}$$

Hierbei muss A (außerhalb des Programms) definiert sein als: $A(x_1, \ldots, x_n) = P$ wobei $\mathsf{fn}(P) \subseteq \{x_1, \ldots, x_n\}$.

Als zusätzliche Reduktionsregel wird aufgenommen:

$$\begin{array}{ll}
\text{(Const)} & A(y_1, \ldots, y_n) \to P[y_1/x_1, \ldots, y_n/x_n] \\
& \quad \text{falls } A(x_1, \ldots, x_n) = P \text{ die Definition von } A \text{ ist}
\end{array}$$

Wir beschreiben kurz, warum Replikation und Rekursion gegeneinander austauschbar sind. D.h. man kann Replikation mithilfe von Rekursion definieren und umgekehrt.

Wir betrachten hierfür den polyadischen π-Kalkül, da dies die Definitionen und Übersetzungen einfacher macht. Dies lässt sich jedoch auch auf den monadischen π-Kalkül übertragen, da wir bereits gesehen haben, wie sich der polyadische π-Kalkül in den monadischen π-Kalkül übersetzen lässt.

Wir betrachten zunächst die Übersetzung des π-Kalküls mit Rekursion in den π-Kalkül mit Replikation. Seien $\{A_1, \dots A_n\}$ alle definierten Konstanten, wobei A_i definiert ist als $A_i(\overrightarrow{x_i}) = P_i$. Die Übersetzung führt neue Kanalnamen $a_1, \dots, a_n$ ein. Jedes Vorkommen einer Konstantenanwendung $A_i(\overrightarrow{y_i})$ wird durch den Prozess $\langle A_i(\overrightarrow{y_i})\rangle := \overline{a_i}\overrightarrow{y_i}.0$ ersetzt. Wobei die Übersetzung $\langle\cdot\rangle$ alle anderen Konstrukte homomorph übersetzt.

Die Definitionen der Konstanten A_i werden ebenfalls durch Prozesse dargestellt. Wir definieren hierfür die Prozesse $A_i' := \,!\, a_i(\overrightarrow{x_i}).\langle P_i\rangle$. Die Prozesse A_i' werden dem Programm hinzugefügt, so dass ein Prozess Q mit Rekursion in den folgenden Prozess mit Replikation übersetzt wird:

$$[\![Q]\!]_{rek} := \nu a_1 \dots \nu a_n.(\langle Q\rangle \mid A_1' \mid \dots \mid A_n')$$

Beachte, dass diese Übersetzung nicht kompositional ist, da (neben der Abhängigkeit der definierten Namen) beispielsweise nicht gilt: $[\![Q_1 \mid Q_2]\!]_{rek} = [\![Q_1]\!]_{rek} \mid [\![Q_2]\!]_{rek}$.

Man kann zeigen, dass die Übersetzung korrekt ist. Wir gehen hierbei nicht genauer auf den Begriff der Korrektheit ein, meinen aber damit, dass sich die Prozesse im Wesentlichen gleich verhalten.

Für die umgekehrte Übersetzung $[\![\cdot]\!]_{repl}$, also die Übersetzung der Replikation in die Rekursion betrachten wir einen Prozess $!\, P$. Sei $\{x_1, \dots, x_n\} = \mathsf{fn}(P)$. Dann wähle eine neue Konstante A_P und übersetze wie folgt:

$$[\![!\, P]\!]_{repl} := A_P(x_1, \dots, x_n)$$

Alle anderen Konstrukte werden homomorph übersetzt. Es fehlt noch die Definition für die Konstante A_P. Diese lautet:

$$A_P(x_1, \dots, x_n) = ([\![P]\!]_{repl} \mid A_P(x_1, \dots, x_n))$$

Auch diese Übersetzung simuliert das Verhalten korrekt.

Abschließend betrachten wir als Beispiel den Prozess $Q = \,!\, \overline{x}z.\mathbf{0} \mid x(y).\mathbf{0} \mid x(w).\mathbf{0}$. Dann erhalten wir $[\![Q]\!]_{repl} = A(x,z) \mid x(y).\mathbf{0} \mid x(w).\mathbf{0}$. wobei $A(x,z) = (\overline{x}z \mid A(x,z))$.

Q reduziert in zwei Schritten zu $!\, \overline{x}z.\mathbf{0}$. Wir betrachten eine Auswertung von $[\![Q]\!]_{repl}$:

$$\begin{aligned}
[\![Q]\!]_{repl} &= A(x,z) \mid x(y).\mathbf{0} \mid x(w).\mathbf{0} \\
&\to (\overline{x}z \mid A(x,z)) \mid x(y).\mathbf{0} \mid x(w).\mathbf{0} \\
&\to A(x,z) \mid \mathbf{0} \mid x(w).\mathbf{0} \\
&\to (\overline{x}z \mid A(x,z)) \mid \mathbf{0} \mid x(w).\mathbf{0} \\
&\to A(x,z) \mid \mathbf{0} \mid \mathbf{0}
\end{aligned}$$

6.2.4 Prozess-Gleichheit im π-Kalkül

Während im Lambda-Kalkül mit der kontextuellen Gleichheit ein allgemein anerkannter und für „richtig“ empfundener Gleichheitsbegriff existiert, gibt es für den π-Kalkül verschiedene Gleichheitsbegriffe, die je nach Anwendung benutzt werden. Um einen Begriff der kontextu-

ellen Gleichheit zu definieren, wäre es nötig, einen Begriff der Terminierung, d.h. Konvergenz für π-Kalkül Prozesse zu definieren. Dies wird jedoch von vielen Forschern abgelehnt, da die Terminierung von Prozessen als nicht zentral angesehen wird, da verteilte Systeme oft in Endlosschleifen laufen und somit nicht terminieren.

Ein Ansatz zur Definition einer Gleichheit im π-Kalkül besteht darin, die Ein- und Ausgabemöglichkeiten eines Prozesses als Begriff für sein Verhalten zu benutzen. Hierfür wird oft ein so genanntes „labeled transition system“ (markiertes Übergangssystem) anstelle der Reduktion verwendet. Dieses kann man sich wie markierte Reduktionen vorstellen, die jedoch mehr „dürfen“ als die bisher definierte Reduktion. Die Markierungen stellen gerade die Ein- und Ausgabemöglichkeiten des Prozesses dar. Hierbei wird unterschieden zwischen *interner Kommunikation* – also die Kommunikation zwischen zwei Prozessen, und der *externen Kommunikation* – also die Kommunikation mit der Außenwelt. Ein andere Sichtweise für die externe Kommunikation ist: Wie kann ein Prozess kommunizieren, wenn man einen Kommunikationspartner als zusätzlichen Prozess hinzufügen würde?

Im Wesentlichen kann man hier zwei Arten der externen Kommunikation unterscheiden: Ein Prozess der Form $P_0 \equiv \nu z_1 \ldots \nu z_n . x(y).P$ mit $x, y \neq z_i$ kann eine Eingabe empfangen, d.h. man müsste einen Prozess der Form $\overline{x}z$ hinzufügen, damit eine Kommunikation stattfinden kann, wobei P_0 der Empfänger wäre. Ein Prozess der Form $P_0 \equiv \nu z_1 \ldots \nu z_n . \overline{x}y.P$ mit $x, y \neq z_i$ kann eine Ausgabe durchführen, d.h. man müsste ein Prozess $x(z).Q$ parallel zu P_0 hinzufügen, damit eine Kommunikation stattfinden kann, P_0 wäre in diesem Fall der Sender. Eine etwas andere Variante ist die Ausgabe eines gebundenen Namens, d.h. für $P_0 = \nu z_1 \ldots \nu z_n . \nu y . . \overline{x}y.P$ mit $x, y \neq z_i$: Zwar findet hier auch eine Kommunikation statt, falls man einen Prozess der Form $x(z).Q$ parallel zu P_0 hinzufügt, aber als „neuer“ Effekt tritt auf, dass durch die Kommunikation der ν-Binder für y nach außen geschoben wird. Dies ergibt insgesamt drei Arten der externen Kommunikation und zusätzlich eine Art für die interne Kommunikation. Deshalb gibt es vier unterschiedliche Markierungen im markierten Übergangssystem, die durch die folgende Grammatik definiert werden.

$$\alpha := \tau \mid x(y) \mid \overline{x}y \mid \overline{x}(y)$$

Die Markierung τ meint hierbei eine interne (stille) Aktion, $x(y)$ zeigt an, dass der Prozess eine Eingabe durchgeführt hat, $\overline{x}y$ zeigt an, dass der Prozesse eine Ausgabe durchgeführt hat (wobei y nicht gebunden war), und $\overline{x}(y)$ zeigt an, dass der Prozess eine gebundene Ausgabe durchgeführt hat. Die Definition der freien, gebunden und aller Namen wird wie folgt auf solche Markierungen übertragen:

$$\begin{array}{rr} \mathsf{fn}(x(y)) = \{x, y\} & \mathsf{bn}(x(y)) = \emptyset \\ \mathsf{fn}(\overline{x}y) = \{x, y\} & \mathsf{bn}(\overline{x}y) = \emptyset \\ \mathsf{fn}(\overline{x}(y)) = \{x\} & \mathsf{bn}(\overline{x}(y)) = \{y\} \\ \mathsf{fn}(\tau) = \emptyset & \mathsf{bn} \end{array}$$

Das markierte Übergangssystem für den synchronen π-Kalkül ist nun durch die folgenden Regeln definiert:

(In)	$x(y).P \xrightarrow{x(z)} P[z/y]$
(Out)	$\overline{x}y.P \xrightarrow{\overline{x}y} P$
(Open)	$\nu y.P \xrightarrow{\overline{x}(y)} Q$, falls $P \xrightarrow{\overline{x}y} Q$ und $x \neq y$.
(Interact)	$x(y).P \mid \overline{x}v.Q \xrightarrow{\tau} P[v/y] \mid Q$
(Par)	$P \mid Q \xrightarrow{\alpha} P' \mid Q$, falls $P \xrightarrow{\alpha} P'$ und $\mathsf{n}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$
(New)	$\nu x.P \xrightarrow{\alpha} \nu x.P'$, falls $P \xrightarrow{\alpha} P'$ und $x \notin \mathsf{n}(\alpha)$
(StructCongr)	$P \xrightarrow{\alpha} P'$, falls $Q \xrightarrow{\alpha} Q'$ und $P \equiv Q$ und $P' \equiv Q'$

Im Folgenden werden in den Beispielen auch Summen verwendet und τ-Präfixe verwendet. Das markierte Zustandsübergangssystem müsste dementsprechend angepasst werden. Wir verzichten an dieser Stelle auf eine formale Definition.

Eine wichtige Aussage bezüglich des Übergangssystems ist, dass τ-Übergänge gerade den Reduktionen entsprechen:

Lemma 6.2.15 (Harmony Lemma). *$P \to Q$ gdw. $P \xrightarrow{\tau} Q'$ wobei $Q' \equiv Q$*

Die für den π-Kalkül definierten so genannten *Bisimulationen* vergleichen das Verhalten zweier Prozesse anhand der Markierungen.

6.2.4.1 Starke Bisimulation

Die starke Bisimulation ist erfüllt, wenn Prozesse genau das gleiche Verhalten bezüglich des markierten Übergangssystems aufweisen:

Definition 6.2.16. *Eine binäre Relation $\mathcal{R}$ auf Prozessen ist eine* starke Bisimulation*, wenn für alle $(P, Q) \in \mathcal{R}$ gilt*

- *Falls $P \xrightarrow{\alpha} P'$, dann gibt es einen Prozess Q', so dass gilt $Q \xrightarrow{\alpha} Q'$ und $(P', Q') \in \mathcal{R}$*
- *Falls $Q \xrightarrow{\alpha} Q'$, dann gibt es einen Prozess P', so dass gilt $P \xrightarrow{\alpha} P'$ und $(P', Q') \in \mathcal{R}$.*

Zwei Prozesse P, Q sind stark bisimilar (geschrieben $P \sim_{b,strong} Q$), falls es eine starke Bisimulation R mit $(P, Q) \in R$ gibt. Die Relation $\sim_{b,strong}$ heißt starke Bisimilarity.

Lemma 6.2.17. *$\sim_{b,strong}$ ist die* größte *starke Bisimulation.*

Beweis. Die Definition der starken Bisimilarity impliziert, dass $\sim_{b,strong}$ die Vereinigung aller starken Bisimulationen ist, d.h. $\sim_{b,strong} = \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ ist starke Bisimulation }\}$. Damit folgt sofort: Die größte starke Bisimulation ist in $\sim_{b,strong}$ enthalten. Es fehlt noch zu zeigen, dass $\sim_{b,strong}$ selbst eine starke Bisimulation ist. Das folgt jedoch daraus, dass die Vereinigung $\mathcal{R}_1 \cup \mathcal{R}_2$ zweier starken Bisimulationen $\mathcal{R}_1, \mathcal{R}_2$ stets wiederum eine starke Bisimulation ist. □

Bemerkung 6.2.18. *Analog zu obiger Definition ist das folgende Vorgehen. Definiere die Funktion auf Relationen von Prozessen, die sich aus obiger Definition ergibt, d.h.: Sei $Proc$ die Menge aller Prozesse des π-Kalküls. Sei $F : (Proc \times Proc) \to (Proc \times Proc)$ die folgende Funktion auf binären Relationen von Prozessen: Sei $\eta \subseteq (Proc \times Proc)$ eine binäre Relation auf Prozessen, dann gilt $(P, Q) \in F(\eta)$ genau dann, wenn:*

- *Falls $P \xrightarrow{\alpha} P'$, dann gibt es einen Prozess Q', so dass gilt $Q \xrightarrow{\alpha} Q'$ und $(P', Q') \in \eta$*

- *Falls $Q \xrightarrow{\alpha} Q'$, dann gibt es einen Prozess P', so dass gilt $P \xrightarrow{\alpha} P'$ und $(P', Q') \in \eta$.*

Starke Bisimilarity ist dann gerade der größte Fixpunkt *der Funktion F. Das* Prinzip der Coinduktion *besagt dann gerade, dass jede Relation η, die* dicht *bezüglich F ist, auch im größten Fixpunkt enthalten ist. Eine Relation η ist F-*dicht*, wenn $\eta \subseteq F(\eta)$ gilt, d.h. die Anwendung von F verkleinert die Relation nicht. In Definition 6.2.16 entsprechen die F-dichten Relationen gerade den Relationen, die eine starke Bisimulation sind.*

Die starke Bisimilarity ist zwar eine Äquivalenzrelation, jedoch keine Kongruenz:

Beispiel 6.2.19. *Sei $P = \overline{z}b.\mathbf{0} \mid a(c).\mathbf{0}$ und $Q = \overline{z}b.a(c).\mathbf{0} + a(c).\overline{z}b.\mathbf{0}$. Es gilt $P \sim_{b,strong} Q$, denn $\mathcal{R} = \{(P,Q), (a(c).\mathbf{0}, a(c).\mathbf{0}), (\overline{z}b.\mathbf{0}, \overline{z}b.\mathbf{0}), (\mathbf{0},\mathbf{0})\}$ ist eine starke Bisimulation (wie sich leicht nachrechnen lässt). Allerdings gilt $x(z).P \not\sim_{b,strong} x(z).Q$ (das zeigen wir gleich), und damit gilt nicht $P \sim_{b,strong} Q \implies C[P] \sim_{b,strong} C[Q]$ für jeden Kontext C, d.h. $\sim_{b,strong}$ ist keine Kongruenz. Um $x(z).P \not\sim_{b,strong} x(z).Q$ nachzuweisen, betrachten wir die Aktion, in der der Name a empfangen wird. Dann gilt: $x(z).P \xrightarrow{x(a)} P[a/z]$ und $x(z).Q \xrightarrow{x(a)} Q[a/z]$. Es genügt nun zu zeigen, dass $P[a/z] \not\sim_{strong} Q[a/z]$ gilt. Dies folgt jedoch sofort, da $P[a/z] = \overline{a}b.\mathbf{0} \mid a(c).\mathbf{0} \xrightarrow{\tau} \mathbf{0}$, aber für $Q[a/z] = \overline{a}b.a(c).\mathbf{0} + a(c).\overline{a}b.\mathbf{0}$ keine τ-Transition möglich ist.*

Ein sinnvoller Begriff einer Programmgleichheit sollte stets eine Kongruenz sein, da man die Gleichheit auch auf Unterprogramme anwenden möchte. Da starke Bisimilarity keine Kongruenz ist, ist sie als Gleichheitsbegriff abzulehnen. Eine Verbesserung erhält man mit der starken vollen Bisimilarity:

Definition 6.2.20. *Zwei Prozesse P, Q sind stark voll bisimilar (Notation $P \sim_{b,strong,full} Q$), wenn für alle Substitutionen σ, die Namen für Namen ersetzen gilt: $\sigma(P) \sim_{b,strong} \sigma(Q)$.*

Man kann nachweisen, dass $\sim_{b,strong,full}$ eine Kongruenz ist.

6.2.4.2 Schwache Bisimulation

Die starke volle Bisimulation wird oft als zu diskriminierend angesehen, d.h. sie unterscheidet zu viele Prozesse. Z.B. sind die Prozesse $P_1 := \nu x.(x(y).\mathbf{0} \mid \overline{x}z.\mathbf{0})$ und $P_2 := \mathbf{0}$ nicht stark bisimilar, da P_1 noch eine interne Aktion durchführen kann, P_2 jedoch nicht. Andererseits ist dieses interne Verhalten von außen gesehen nicht beobachtbar, und daher sollte man P_1 und P_2 nicht unbedingt unterscheiden.

Um solche Unterschiede in den internen Aktionen in der Gleichheit zu ignorieren, wird die sogenannte schwache Bisimulation definiert, die oft auch einfach nur „Bisimulation“ heißt. Übertragen auf die Markierungen bezüglich des Übergangssystems bedeutet das ignorieren der internen Aktionen, dass τ-Transitionen keine Rolle für die Gleichheit spielen sollen. Sei $\stackrel{\tau}{\Rightarrow}$ die reflexiv-transitive Hülle von $\xrightarrow{\tau}$ (d.h. $P \stackrel{\tau}{\Rightarrow} Q$, wenn P mit 0 oder endlichen vielen τ-Transitionen in Q überführt werden kann) und sei $\stackrel{\alpha}{\Rightarrow}$ die Relation $\stackrel{\tau}{\Rightarrow}\xrightarrow{\alpha}\stackrel{\tau}{\Rightarrow}$, d.h. die Relation die zunächst beliebig viele τ-Transitionen ausführt, anschließend die Aktion α durchführt und danach nochmal beliebig viele τ-Transitionen durchführt.

Definition 6.2.21. *Eine binäre Relation $\mathcal{R}$ auf Prozessen ist eine* Bisimulation *wenn für alle $(P,Q) \in \mathcal{R}$ gilt*

- *Falls* $P \xrightarrow{\alpha} P'$*, dann gibt es einen Prozess* Q' $Q \stackrel{\alpha}{\Rightarrow} Q'$ *und* $(P', Q') \in \mathcal{R}$
- *Falls* $Q \xrightarrow{\alpha} Q'$*, dann gibt es einen Prozess* P'*, so dass gilt* $P \stackrel{\alpha}{\Rightarrow} P'$ *und* $(P', Q') \in \mathcal{R}$.

Zwei Prozesse P, Q *sind bisimilar (geschrieben* $P \sim_b Q$*), falls es eine Bisimulation* $\mathcal{R}$ *mit* $(P, Q) \in \mathcal{R}$ *gibt. Die Relation* $\sim_b$ *heißt* Bisimilarity.

Zwei Prozesse P, Q *sind* voll bisimilar *(*$P \sim_{b,full} Q$*) falls* $\sigma(P) \sim_b \sigma(Q)$ *für alle Substitutionen* σ *gilt.*

Während Bisimilarity keine Kongruenz ist (für die Prozesse P, Q aus Beispiel 6.2.19 gilt auch für die (nicht starke) Bisimilarity $P \sim_b Q$ aber $x(z).P \not\sim_b x(z).Q$), ist die volle Bisimilarity eine Kongruenz.

Offensichtlich gilt, dass starke (volle) Bisimilarity die (volle) Bisimilarity impliziert und dass die Umkehrung nicht gilt:

Satz 6.2.22. *Für alle Prozesse* P, Q *gilt:*

- $P \sim_{b,strong} Q \implies P \sim_b Q$
- $P \sim_{b,strong,full} Q \implies P \sim_{b,full} Q$.

Weiterhin gilt $\sim_{b,strong} \neq \sim_b$ *und* $\sim_{b,strong,full} \neq \sim_{b,full}$.

Der Beweis ist in der entsprechenden Literatur nachzulesen.

6.2.4.3 Barbed Kongruenz

Für den π-Kalkül gibt es als weiteren Gleichheitsbegriff die so genannte *barbed Kongruenz*.

Definition 6.2.23 (Barb). *Ein Prozess* P *hat einen* Input-Barb an Kanal x *falls* $P \xrightarrow{x(y)} P'$*. Wir schreiben in diesem Fall* $P \upharpoonright^x$.

Ein Prozess P *hat einen* Output-Barb an Kanal x *falls* $P \xrightarrow{\overline{x}y} P'$*. Wir schreiben in diesem Fall* $P \upharpoonright^{\overline{x}}$.

Für $\beta \in \{x, \overline{x}\}$ *schreiben wir* $P \downarrow_\beta$ *falls es einen Prozess* Q *gibt, so dass* $P \stackrel{\tau}{\Rightarrow} Q$ *und* $Q \upharpoonright^\beta$*. D.h.* P *kann nach einigen* τ*-Aktionen eine Input- bzw. Output-Aktion durchführen.*

Definition 6.2.24. *Eine binäre Relation* $\mathcal{R}$ *auf Prozessen ist eine* barbed Bisimulation*, falls für alle* $(P, Q) \in \mathcal{R}$ *gilt:*

- *Falls* $P \upharpoonright^x$ *(bzw.* $P \upharpoonright^{\overline{x}}$*), dann* $Q \downarrow_x$ *(bzw. (*$Q \downarrow_{\overline{x}}$*)*
- *Falls* $Q \upharpoonright^x$ *(bzw.* $Q \upharpoonright^{\overline{x}}$*), dann* $P \downarrow_x$ *(bzw. (*$P \downarrow_{\overline{x}}$*)*
- *Falls* $P \xrightarrow{\tau} P'$*, dann existiert* Q' *mit* $Q \stackrel{\tau}{\Rightarrow} Q'$ *und* $(P', Q') \in \mathcal{R}$
- *Falls* $Q \xrightarrow{\tau} Q'$*, dann existiert* P' *mit* $P \stackrel{\tau}{\Rightarrow} P'$ *und* $(P', Q') \in \mathcal{R}$

Zwei Prozesse P, Q *sind* barbed bisimilar *(geschrieben* $P \sim_{b,barbed} Q$*) falls es eine barbed Bisimulation gibt, die* (P, Q) *enthält.*

Die barbed Bisimulation wird zu einer Kongruenz durch die folgende Definition erweitert:

Definition 6.2.25. *Zwei Prozesse* P, Q *sind* barbed kongruent *(geschrieben* $P \sim_{c,b,barbed} Q$*) falls für alle Kontexte* C *gilt:* $C[P] \sim_{b,barbed} C[Q]$.

Man kann die barbed Kongruenz auch ohne Verwendung des markierten Übergangssystems, sondern nur unter Verwendung der Reduktion definieren, was aus dem folgenden Lemma und dem Harmony-Lemma folgt:

Lemma 6.2.26. *Für alle Prozesse des synchronen π-Kalküls (mit Summen) gilt:*

- $P \mathrel{\Rsh}^{x}$ *genau dann, wenn* $P \equiv \nu v_1 \ldots \nu v_n.(x(y).P' + M \mid Q)$ *wobei* $x \neq v_i$.
- $P \mathrel{\Rsh}^{\overline{x}}$ *genau dann, wenn* $P \equiv \nu v_1 \ldots \nu v_n.(\overline{x}y.P' + M \mid Q)$ *wobei* $x \neq v_i$.

Die barbed Kongruenz enthält die volle Bisimilarity:

Theorem 6.2.27. *Für alle Prozesse P, Q gilt:*
$P \sim_{b,full} Q \implies P \sim_{c,b,barbed} Q$

Es ist unbekannt, ob die Umkehrung dieser Implikation gilt.

6.2.4.4 May- und Must-Testing

Eine andere Gleichheitsdefinition, die keine Bisimulation verwendet, und analog zur kontextuellen Gleichheit, wie sie im Lambda-Kalkül verwendet wird, basiert ebenfalls auf der Beobachtung des Barb-Verhaltens. Dieses tritt an die Stelle des Terminierungstests.

Definition 6.2.28. *Die* may-testing *Präordnung $\leq_{may}$ auf Prozessen ist definiert als:*
$P \leq_{may} Q$ gdw. für alle Kontexte C, für alle Namen x: $C[P] \downarrow_x \implies C[Q] \downarrow_x$ und $C[P] \downarrow_{\overline{x}} \implies C[Q] \downarrow_{\overline{x}}$.
Die may-testing *Äquivalenz $\sim_{may}$ ist der symmetrische Abschluss der Präordnung, d.h. $P \sim_{may} Q$ gdw. $P \leq_{may} Q$ und $Q \leq_{may} P$*

Die may-testing Äquivalenz ist eine Kongruenz. Sie ist sehr grobkörnig, so gilt z.B.

Satz 6.2.29. *Für alle Prozesse P, Q gilt: $P \sim_{c,b,barbed} Q \implies P \sim_{may} Q$.*

Die Umkehrung gilt nicht, da z.B. $a(x).\mathbf{0} \oplus (b(x).\mathbf{0} \oplus c(x).\mathbf{0})$ und $(a(x).\mathbf{0} \oplus b(x).\mathbf{0}) \oplus c(x).\mathbf{0}$ may-testing äquivalent, aber nicht barbed kongruent sind.
Tatsächlich ist die may-testing Äquivalenz zu grob-körnig, da z.B. die Prozesse $x(y).\mathbf{0} \oplus \mathbf{0}$ und $x(y).\mathbf{0}$ may-testing äquivalent sind, obwohl ihr Verhalten sehr unterschiedlich ist: $x(y).\mathbf{0} \oplus \mathbf{0}$ kann zum Prozess werden, der keine Kommunikationsmöglichkeiten mehr hat, während $x(y).\mathbf{0}$ immer an Kanal x empfangen kann.
Daher wird zusätzlich die *should-testing* Äquivalenz betrachtet. Für $\mu \in \{x, \overline{x}\}$ schreiben wir $P \Downarrow_\mu$ (P muss einen Barb μ haben), falls

$$\forall Q : P \xrightarrow{*} Q \implies Q \downarrow_\mu .$$

D.h. jeder Reduktionsnachfolger von P kann einen Barb an μ haben.

Definition 6.2.30. *Die* should-testing *Präordnung $\leq_{should}$ auf Prozessen ist definiert als:*
$P \leq_{should} Q$ gdw. für alle Kontexte C, für alle Namen x: $C[P] \Downarrow_x \implies C[Q] \Downarrow_x$ und $C[P] \Downarrow_{\overline{x}} \implies C[Q] \Downarrow_{\overline{x}}$.
Die should-testing *Äquivalenz $\sim_{should}$ ist der symmetrische Abschluss der Präordnung, d.h. $P \sim_{should} Q$ gdw. $P \leq_{should} Q$ und $Q \leq_{should} P$*

Die should-testing Äquivalenz ist feiner als die may-testing Äquivalenz, d.h. $\sim_{should} \subset \sim_{may}$, aber gröber als barbed Kongruenz:

Satz 6.2.31. *Für alle Prozesse P, Q gilt:* $P \sim_{c,b,barbed} Q \implies P \sim_{should} Q$.

Insgesamt ergibt sich daher die folgende Hierarchie der Prozessgleichheiten:

Theorem 6.2.32. $\sim_{b,strong,full} \subseteq \sim_{b,full} \subseteq \sim_{b,c,barbed} \subseteq \sim_{should} \subseteq \sim_{may}$

6.3 Ein Shared-Memory-Modell: Der CHF-Kalkül

Der CHF-Kalkül ist ein Modell für Concurrent Haskell erweitert um Futures, die den impliziten Futures aus Abschnitt 5.2.11 entsprechen. CHF wurde in (Sabel & Schmidt-Schauß, 2011a) eingeführt und untersucht, wobei sich sie die operationale Semantik auch an den Arbeiten (Peyton Jones et al., 1996; Peyton Jones, 2001) orientiert.

6.3.1 Syntax

Wir definieren zunächst die Syntax von CHF. Diese ist zweistufig: auf der oberen Ebene befinden sich *Prozesse* und einige andere Komponenten, und innerhalb von Prozessen sind in der unteren Stufe *Ausdrücke* eines erweiterten Lambda-Kalküls zu finden.

Die Syntax von Prozessen $P \in \mathit{Proc}$ ist durch die folgende Grammatik definiert, wobei x eine Variable bezeichnet und e ein Ausdruck ist. Ausdrücke werden wir erst später erläutern.

$$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid x \Leftarrow e \mid x = e \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m} -$$

Wir beschreiben die einzelnen Konstrukte: $P_1 \mid P_2$ steht (wie im π-Kalkül) für die parallele Komposition von Prozessen (oder anderen Komponenten) und $\nu x.P$ schränkt den Gültigkeitsbereich der Variablen x auf den Prozess P ein. $x \Leftarrow e$ stellt einen nebenläufigen Thread dar, der den Ausdruck e auswertet. Der Name des Threads ist dabei x, wobei wir diesen Namen auch als *Future* bezeichnen. $x = e$ ist eine (globale) Bindung: die Variable x ist an den Ausdruck e gebunden. Solche Bindungen dürfen durchaus *rekursiv* sein, d.h. x darf wiederum in e (oder auch in anderen Bindungen) vorkommen. $x\,\mathbf{m}\,e$ und $x\,\mathbf{m} -$ stellen MVars dar: Eine MVar ist ein (synchronisierender) Speicherplatz, der leer oder gefüllt sein kann. $x\,\mathbf{m}\,e$ ist gerade die MVar namens x, die den Inhalt e enthält. $x\,\mathbf{m} -$ ist die leere MVar namens x.

Eine weitere syntaktische Bedingung (die nicht durch die Grammatik erfasst ist) ist, dass in einem Prozess ein Thread ausgezeichnet sein kann, als sogenannter *Main-Thread*. Wenn wir diesen kenntlich machen möchten, so benutzen wir die Notation $x \overset{\mathsf{main}}{\Leftarrow} e$. Der Main-Thread ist der Haupthread des Programms. Wie wir später sehen werden, endet die Auswertung eines Prozesses genau dann, wenn der Main-Thread erfolgreich beendet ist. D.h. wir benutzen die gleiche Vorgehensweise wie sie in Concurrent Haskell zu finden ist: Wenn der Main-Thread terminiert, dann werden dadurch alle weiteren nebenläufige Prozesse beendet.

Threads werten *Ausdrücke* aus und Ausdrücke kommen auch als Inhalt von MVars vor. Bevor wir Ausdrücke selbst definieren, treffen wir die Annahme, dass es eine Menge von *Datenkonstruktoren* gibt. Einen Datenkonstruktor schreiben wir abstrakt als $c_{T,i}$, oder manchmal auch einfach nur als c. Dabei gelte:

- Die Menge der Konstruktoren ist partitioniert in *Typkonstruktoren* T, d.h. insbesondere jeder Datenkonstruktor $c_{T,i}$ gehört genau zu einem Typkonstruktor T.
- Für einen Typkonstruktor T sind $c_{T,1}, \ldots, c_{T,|T|}$ genau die Datenkonstruktoren des Typkonstruktors T (wobei $|T|$ die Mächtigkeit der Menge bezeichnet)
- Jeder Datenkonstruktor $c_{T,i}$ hat eine *Stelligkeit* (arity) $\mathrm{ar}(c_{T,i}) \in \mathbb{N}_0$. Diese gibt gerade an, wieviele Argumente der Datenkonstruktor erhalten kann.
- Wir erlauben ausschließlich, dass Datenkonstruktoren *voll gesättigt* vorkommen, d.h. ausschließlich *Konstruktoranwendungen* der Form $(c_{T,i}\, e_1 \ \ldots \ e_{\mathrm{ar}(c_{T,i})})$ sind erlaubt, aber nicht solche mit weniger als $\mathrm{ar}(c_{T,i})$-Argumenten.

Dies ist eine abstrakte Darstellung für die Menge der Datentypen und erfasst daher sehr viele Möglichkeiten. Wir geben einige mögliche *Beispiele* an: Der Typ(-konstruktor) `Bool` hat die Datenkonstruktoren `True` und `False`, wobei beide Konstruktoren die Stelligkeit 0 haben (also Konstanten sind). Der Typkonstruktor `Liste` hat die Datenkonstruktoren `Nil` (für die leere Liste) und `Cons`, wobei $\mathrm{ar}(\texttt{Nil}) = 0$ und $\mathrm{ar}(\texttt{Cons}) = 2$. Der Konstruktor `Cons` erwartet daher zwei Argumente, üblicherweise ist das erste Argument das Kopfelement der Liste und das zweite Argument die Restliste. Z.B. kann man die Liste [1,2,3] damit darstellen durch `Cons` 1 (`Cons` 2 (`Cons` 3 `Nil`)). Der Typkonstruktor `Paar` hat einen Datenkonstruktor `Paar` mit Stelligkeit 2. Eine weitere Vorstellung wäre ein Typ(-konstruktor) `Int` mit 2^{32} Konstruktoren (für die Zahlen), die alle die Stelligkeit 0 haben. Wir nehmen außerdem an, dass es (wie in Haskell) einen Typkonstruktor `()` mit einem Datenkonstruktor `()` gibt, der die Stelligkeit 0 besitzt.

Ausdrücke $e \in$ *Exp* werden durch die folgende Grammatik gebildet, wobei eine Untermenge der Ausdrücke, die *monadischen Ausdrücke* $me \in$ *MExp* darstellen:

$$
\begin{aligned}
e, e_i \in \textit{Exp} ::= {} & x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid c\, e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 \\
& \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e \\
& \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots\ (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\
me \in \textit{MExp} ::= {} & \texttt{return}\ e \mid e_1 \gg\!= e_2 \mid \texttt{future}\ e \\
& \mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2
\end{aligned}
$$

Wie der Lambda-Kalkül beinhalten Ausdrücke Variablen x, Abstraktionen $\lambda x.e$, und Anwendungen $(e_1\ e_2)$. Darüber hinaus gibt es (voll gesättigte) Konstruktoranwendungen $(c\, e_1 \ldots e_{\mathrm{ar}(c)})$, `case`-Ausdrücke, die zum Zerlegen der Konstruktoren dienen, `seq`-Ausdrücke `seq` $e_1\ e_2$ die zur sequentiellen Auswertungen dienen (werte erst e_1 aus und im Anschluss e_2) und `letrec`-Ausdrücke, die es erlauben (lokale) Bindungen zu konstruieren, die auch rekursiv sein dürfen: In `letrec` $x_1 = e_1, \ \ldots, x_n = e_n$ `in` e ist jede Variable x_i in $e_1, \ldots, e_n$ und e gebunden.

Für `case`-Ausdrücke gibt es pro Typkonstruktor T ein eigenes `case`$_T$-Konstrukt und $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ bezeichnen wir als `case`-Alternative. Für einen fixen Typkonstruktor T muss ein `case`$_T$-Ausdruck für jeden Datenkonstruktor $c_{T,i}$ genau eine `case`-Alternative enthalten. In einer `case`-Alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ bezeichnen wir $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$ als das Pattern und e_i als die rechte Seite der Alternativen. In einem `case`-Pattern $c_{T,i}\, x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$ müssen die Variablen x_i alle paarweise verschieden sein. Ebenso müssen in einem `letrec`-Ausdruck `letrec` $x_1 = e_1, \ \ldots, x_n = e_n$ `in` e die Variablen x_i alle paarweise verschieden sein. Daher sind z.B. die Konstrukte `letrec` $x =$ `True`, $x =$ `False` `in` x und

$\texttt{case}_{\text{List}}\ x\ \texttt{of}\ (\texttt{Nil} \to \texttt{Nil})\ (\texttt{Cons}\ x\ x \to x)$ keine syntaktisch korrekten Ausdrücke.

Ausdrücke umfassen zusätzlich die Menge der monadischen Ausdrücke: Dies enthalten die beiden notwendigen Operatoren für die Monade: $(\texttt{return}\ e)$ verpackt einen beliebigen Ausdruck als monadische Aktion, der binäre >>= -Operator komponiert zwei monadische Aktionen zu einer Sequenz. Die Ausdrücke $\texttt{newMVar}\ e$, $\texttt{takeMVar}\ e$, $\texttt{putMVar}\ e_1\ e_2$ dienen dem Erzeugen und dem Zugriff auf MVars (wie in Concurrent Haskell). Schließlich gibt es noch den Operator $\texttt{future}\ e$, der ähnlich (aber nicht gleich) zu `forkIO` in Concurrent Haskell agiert: Es wird ein nebenläufiger Thread erzeugt, der die Aktion e ausführt. Der erzeugende Thread kann dabei auf der Ergebnis der nebenläufigen Auswertung zugreifen.

Für Ausdrücke und Prozesse führen die Konstrukte $\nu x.P$, $\lambda x.e$, $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$, und `case`-Alternativen Bindungsbereiche von Variablen ein. Wir verzichten auf die formale Definition, aber verwenden BV für die Menge der *gebundenen* Variablen und FV für die Menge der *freien* Variablen eines Prozesses bzw. eines Ausdrucks. Wir verwenden auch den damit verbundenen Begriff der α-Äquivalenz $=_\alpha$ von Prozessen und Ausdrücken und identifizieren α-äquivalente Prozesse. Wir verwenden auch für CHF die Konvention, dass in einem Prozess alle gebundenen Variablen unterschiedliche Namen haben und dass Namen gebundener Variablen stets verschieden sind von Namen freier Variablen. Wir nehmen auch an, dass diese Konvention von der Reduktion eingehalten wird, indem (implizit) α-Umbenennungen durchgeführt werden.

Eine andere Menge von Variablen sind die sogenannten *eingeführten Variablen* eines Prozesses P: Dies sind alle Namen der Threads (die Futures), die Namen der MVars, und die linken Seiten der globalen Bindungen.

Wir sagen ein Prozess P ist *wohlgeformt* genau dann, wenn alle eingeführten Variablen paarweise verschieden sind und P maximal einen Main-Thread besitzt.

Für Prozesse führen wir (ähnlich zum π-Kalkül) eine strukturelle Kongruenz ein.

Definition 6.3.1. *Strukturelle Kongruenz $\equiv$ ist die kleineste Kongruenz auf Prozessen, die die folgenden Regeln erfüllt:*

$$
\begin{array}{rcl}
P_1 \mid P_2 & \equiv & P_2 \mid P_1 \\
P_1 \mid (P_2 \mid P_3) & \equiv & (P_1 \mid P_2) \mid P_3 \\
(\nu x.P_1) \mid P_2 & \equiv & \nu x.(P_1 \mid P_2),\ \textit{falls}\ x \notin FV(P_2) \\
\nu x_1.\nu x_2.P & \equiv & \nu x_2.\nu x_1.P \\
P_1 & \equiv & P_2,\ \textit{falls}\ P_1\ \textit{und}\ P_2\ \alpha\textit{-äquivalente Prozesse sind}\ (P_1 =_\alpha P_2)
\end{array}
$$

6.3.2 Typisierung

Der CHF-Kalkül ist typisiert, d.h. es gibt ein Typsystem, dass ungetypte Ausdrücke und Prozesse herausfiltert. CHF verwendet ein *monomorphes* Typsystem, d.h. alle Ausdrücke und Unterausdrücke haben einen festen Typ, der keine Typvariablen enthält. Dies ist eine Einschränkung gegenüber Concurrent Haskell, da dort *polymorphe* Typen vorhanden sind, also Typen die auch Typvariablen haben dürfen (und semantisch für eine ganze Menge von monomorphen Typen stehen). CHFs Typsystem ist monomorph, da es formal leichter handhabbar ist, und die Unterschiede zu polymorpher Typisierung nicht allzu groß sind.

Die Syntax von Typen $\tau \in \mathit{Typ}$ ist durch die folgende Grammatik definiert:

$$\tau, \tau_i \in \mathit{Typ} ::= \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_{\mathrm{ar}(T)}) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$$

Hierbei ist T ein Typkonstruktor, wobei jeder Typkonstruktor eine Stelligkeit $\mathrm{ar}(T) \in \mathbb{N}_0$ besitzt, und innerhalb von Typen nur vollständig gesättigte Typkonstruktoranwendungen $(T\ \tau_1 \ldots \tau_{\mathrm{ar}(T)})$ auftreten dürfen. Z.B. haben die Typkonstruktoren `Bool` und `()` die Stelligkeit 0 und sind daher selbst Typen, hingegen hat der Typ `List` die Stelligkeit 1, und daher stellt z.B. `List Bool` den Typ von Listen mit Inhaltstyp `Bool` dar. `IO` τ is der Typ einer `IO`-Aktion mit Rückgabewert vom Typ τ, `MVar` τ ist der Typ einer MVar mit Inhaltstyp τ, und $\tau_1 \to \tau_2$ stellt einen Funktionstyp dar (d.h. eine Funktion mit Eingabe vom Typ τ_1 und Ausgabe vom Typ τ_2. Der Typpfeil $\to$ ist dabei rechtsassoziativ, d.h. $\tau_1 \to \tau_2 \to \tau_3$ meint den Typ $\tau_1 \to (\tau_2 \to \tau_3)$ und *nicht* den Typ $(\tau_1 \to \tau_2) \to \tau_3$.

Wir behandeln Datenkonstruktoren etwas polymorpher als es bei reinen monomorphen Typsystemen zulässig wäre. Wir formalisieren dies etwas genauer: Ein polymorpher Typ wird genauso gebildet, wie ein monomopher Typ $\tau \in \mathit{Typ}$, mit dem Unterschied, dass *Typvariablen* auch zugelassen sind, d.h. *polymorphe Typen* $\overline{\tau} \in \mathit{Typ}_{poly}$ werden durch die folgende Grammatik gebildet, wobei α ein Typvariable (aus einer unendlichen Menge von Variablen) ist:

$$\overline{\tau}, \overline{\tau_i} \in \mathit{Typ}_{poly} ::= \alpha \mid \texttt{IO}\ \overline{\tau} \mid (T\ \overline{\tau_1} \ldots \overline{\tau_{\mathrm{ar}(T)}}) \mid \texttt{MVar}\ \overline{\tau} \mid \overline{\tau_1} \to \overline{\tau_2}$$

Ein Datenkonstruktor $c_{T,i}$ vom Typkonstruktor T hat einen (u.U. polymorphen) Typ der Form

$$\overline{\tau_1} \to \overline{\tau_2} \to \ldots \to \overline{\tau_{\mathrm{ar}(c_{T,i})}} \to T\ \alpha_1 \ldots \alpha_{\mathrm{ar}(T)}$$

wobei alle vorkommenden Typvariablen in der Menge $\{\alpha_1, \ldots, \alpha_{\mathrm{ar}(T)}\}$ enthalten sind.

Z.B. hat der Listenkonstruktor `Cons` den polymorphen Typ $\alpha \to$ `List` α, und der `Paar`-Konstruktor hat den polymorphen Typ $\alpha_1 \to \alpha_2 \to$ `Paar` $\alpha_1\ \alpha_2$.

Eine *Typsubstitution* σ ist eine Abbildung, die Typvariablen auf Typen abbildet. Wir verwenden direkt die Erweiterung von Typsubstitutionen auf Typen, sodass wir $\sigma(\overline{\tau})$ für einen Typen schreiben dürfen, wobei $\sigma(\tau)$ gerade der Typ τ ist, wobei alle auftretenden Typvariablen α in τ durch $\sigma(\tau)$ ersetzt sind. Für einen polymorphen Typ $\overline{\tau}$ ist die Typsubstitution σ eine *Grundsubstitution*, wenn $\sigma(\overline{\tau})$ ein monomorpher Typ ist (d.h. $\sigma(\overline{\tau}) \in \mathit{Typ}$). Jetzt können wir definieren, wie ein Datenkonstruktor $c_{T,i}$ im monomorphen Typsystem von *CHF* verwendet werden darf: Für einen Datenkonstruktor $c_{T,i}$ vom polymorphen Typ $\overline{\tau}$ ist die Menge seiner monomorphen Typen definert als

$$\mathrm{types}(c_{T,i}) = \{\sigma(\overline{\tau}) \mid \sigma \text{ ist Grundsubstitution für } \tau\}$$

.

Für die Typisierung nehmen wir der Einfachheit halber an, dass jede Variable x bereits einen festen Typ $\Gamma(x)$ besitzt. Wir listen im Folgenden die Typisierungsregeln für *CHF* auf. Die Schreibweise einer Herleitungsregel ist dabei

$$\frac{\text{Voraussetzung}}{\text{Konsequenz}}$$

und $\Gamma \vdash e :: \tau$ bedeutet, dass aus der Typannahme für Variablen Γ für Ausdruck e der Typ τ hergeleitet werden kann. Die Schreibweise $\Gamma \vdash P :: \texttt{wt}$ bedeutet, dass Prozess P *wohl-getypt* ist.

Typisierungsregel für Variablen Für Variablen x muss der vorgegebene Typ $\Gamma(x)$ mit dem hergeleiteten Typ übereinstimmen. Die folgende Regel drückt dies aus:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$

Typisierungsregeln für Prozesse Für Prozesse gibt es sechs Typisierungsregeln. Wohlgetyptheit der parallelen Komposition erfordert die Wohlgetyptheit beider Subprozesse, der ν-Binder stört die Typisierung nicht, Threads sind wohl getypt, wenn der auszuwertende Ausdruck eine IO-Aktion ist, deren Rückgabetyp mit dem Typ der Future übereinstimmt. Für Bindungen müssen die linke und die rechte Seite den gleichen Typ besitzen. MVars sind wohlgetypt, wenn der Inhaltstyp mit dem Typ des Inhalts übereinstimmt. Die entsprechenden Regeln sind:

$$\frac{\Gamma \vdash P_1 :: \texttt{wt}, \Gamma \vdash P_2 :: \texttt{wt}}{\Gamma \vdash P_1 \mid P_2 :: \texttt{wt}} \quad \frac{\Gamma \vdash P :: \texttt{wt}}{\Gamma \vdash \nu x.P :: \texttt{wt}} \quad \frac{\Gamma \vdash x :: \tau, \Gamma \vdash e :: \texttt{IO}\ \tau}{\Gamma \vdash x \Leftarrow e :: \texttt{wt}}$$

$$\frac{\Gamma \vdash x :: \tau, \Gamma \vdash e :: \tau}{\Gamma \vdash x = e :: \texttt{wt}} \quad \frac{\Gamma \vdash x :: \texttt{MVar}\ \tau, \Gamma \vdash e :: \tau}{\Gamma \vdash x\ \mathbf{m}\ e :: \texttt{wt}} \quad \frac{\Gamma \vdash x :: \texttt{MVar}\ \tau}{\Gamma \vdash x\ \mathbf{m}\ - :: \texttt{wt}}$$

Typisierungsregeln für monadische Ausdrücke Die Typisierungsregeln für die monadischen Operatoren drücken gerade die Typen der Operatoren aus: Der Operator `return` verpackt einen Ausdruck als IO-Aktion und hat daher den Typ $\tau \to \texttt{IO}\ \tau$ für alle τ. Der `>>=` -Operator erwartet eine IO-Aktion und eine Funktion, die das Ergebnis der IO-Aktion verwendet und daraus eine neue IO-Aktion erstellt. Der Typ ist daher $\texttt{IO}\ \tau_1 \to (\tau_1 \to \texttt{IO}\ \tau_2) \to \texttt{IO}\ \tau_2$. Die `takeMVar`-Operation erwartet einen Namen einer MVar und liefert den Inhalt der MVar, allerdings als IO-Aktion, d.h. der Typ ist $\texttt{MVar}\ \tau \to \texttt{IO}\ \tau$. Die `putMVar`-Operation erwartet den Namen einer MVar und den neuen Inhalt der MVar. Das Ergebnis ist eine IO-Aktion, deren Rückgabewert egal ist, daher wählen wir das Nulltupel `()`, insgesamt ergibt dies den Typ $\texttt{MVar}\ \tau \to \tau \to \texttt{IO}\ ()$. Die Operation `newMVar` erwartet den Inhalt der neuen MVar und erzeugt anschließend die gefüllte MVar innerhalb einer IO-Aktion, die den Namen der MVar zurückliefert. Der Typ ist daher $\tau \to \texttt{IO}\ (\texttt{MVar}\ \tau)$. Die Typisierungsregeln sind somit:

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{return}\ e :: \texttt{IO}\ \tau} \quad \frac{\Gamma \vdash e_1 :: \texttt{IO}\ \tau_1, \Gamma \vdash e_2 :: \tau_1 \to \texttt{IO}\ \tau_2}{\Gamma \vdash e_1 \texttt{>>=} e_2 :: \texttt{IO}\ \tau_2}$$

$$\frac{\Gamma \vdash e :: \texttt{IO}\ \tau}{\Gamma \vdash \texttt{future}\ e :: \texttt{IO}\ \tau} \quad \frac{\Gamma \vdash e :: \texttt{MVar}\ \tau}{\Gamma \vdash \texttt{takeMVar}\ e :: \texttt{IO}\ \tau}$$

$$\frac{\Gamma \vdash e_1 :: \texttt{MVar}\ \tau, \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \texttt{putMVar}\ e_1\ e_2 :: \texttt{IO}\ ()} \quad \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{newMVar}\ e :: \texttt{IO}\ (\texttt{MVar}\ \tau)}$$

Typisierungsregeln für funktionale Ausdrücke Für Abstraktionen muss ein Funktionstyp hergeleitet werden, wobei der Argumenttyp dem Typ der Variablen und der Ergebnistyp dem Typ des Rumpfs entsprechen muss. Für Anwendungen muss das linke Argument in Funktionsposition einen Funktionstyp erhalten, der zum rechten Argument passt. Die beiden Typisie-

rungsregeln sind:

$$\frac{\Gamma \vdash x :: \tau_1,\ \Gamma \vdash e :: \tau_2}{\Gamma \vdash (\lambda x.e) :: \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2,\ \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

Für eine Konstruktoranwendung muss ein passender Typ in $\mathrm{types}(c_{T,i})$ enthalten sein, so dass alle Argumente mit den passenden Typen typisiert werden können. Für `seq`-Ausdrücke müssen beide Argumente wohl-getypt sein und der Typ des `seq`-Ausdrucks entspricht dem Typ des zweiten Arguments. Wir fordern hierbei eine weitere Einschränkung: Der Typ des ersten Arguments muss ein Konstruktortyp oder ein Funktionstyp sein. Oder umgekehrt: Wir verbieten, dass der Typ ein `IO`- oder ein `MVar`-Typ ist. Diese Einschränkung ist in Haskell nicht vorhanden, allerdings ist sie notwendig, damit die monadischen Gesetze für die `IO`-Monade wirklich gelten (sowohl in *CHF* als auch in Haskell). Die Typregeln sind daher:

$$\frac{\begin{array}{c}\forall i : \Gamma \vdash e_i :: \tau_i, \\ \tau_1 \to \ldots \to \tau_{n+1} \in \texttt{types(c)}\end{array}}{\Gamma \vdash (c\ e_1\ \ldots\ e_n) :: \tau_{n+1}} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 :: \tau_1,\ \Gamma \vdash e_2 :: \tau_2, \\ \text{wobei } \tau_1 = \tau_3 \to \tau_4 \text{ oder } \tau_1 = (T\ldots)\end{array}}{\Gamma \vdash (\texttt{seq}\ e_1\ e_2) :: \tau_2}$$

Für `letrec`-Ausdrücke müssen für alle Bindungen die Typen der linken und rechten Seiten übereinstimmen (und wohlgetypt sein). Der `in`-Ausdruck muss ebenfalls wohlgetypt sein, er bestimmt den Typ des gesamten `letrec`-Ausdrucks:

$$\frac{\forall i : \Gamma \vdash x_i :: \tau_i,\ \forall i : \Gamma \vdash e_i :: \tau_i,\ \Gamma \vdash e :: \tau}{\Gamma \vdash (\texttt{letrec}\ x_1 = e_1, \ldots\ x_n = e_n\ \texttt{in}\ e) :: \tau}$$

Für `case`-Ausdrücke müssen alle Unterausdrücke wohlgetypt sein und der zu zerlegende Ausdruck muss vom Konstruktortyp T des $\texttt{case}_T$-Konstrukts sein. Die Typen der `case`-Pattern müssen mit diesem Typ übereinstimmen. Die Typen aller rechten Seiten der Alternativen müssen identisch sein und dieser Typ entspricht dem Typ des gesamten `case`-Ausdrucks:

$$\frac{\Gamma \vdash e :: \tau_1 \text{ und } \tau_1 = (T\ \ldots),\ \forall i : \Gamma \vdash (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,n_i}) :: \tau_1,\ \forall i : \Gamma \vdash e_i :: \tau_2}{\Gamma \vdash (\texttt{case}_T\ e\ \texttt{of}(c_{T,1}\ x_{1,1}\ \ldots\ x_{1,n_1} \to e_1) \ldots (c_{T,|T|}\ x_{|T|,1}\ \ldots\ x_{|T|,n_{|T|}} \to e_{|T|})) :: \tau_2}$$

6.3.2.1 Beispiele und Bemerkungen

Als erstes Beispiel betrachten wir die Identität $\lambda x.x$. Sei $\Gamma(x) = \tau$ für irgendeinen Typen τ. Dann lässt sich $\lambda x.x$ mit dem Typ $\tau \to \tau$ typisieren:

$$\frac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{\Gamma \vdash (\lambda x.x) :: \tau \to \tau}$$

Als weiteres Beispiel betrachten wir den Ausdruck $\lambda x.(x\ x)$. Egal wie wir $\Gamma(x)$ wählen, der Ausdruck ist nicht typisierbar. Sei $\Gamma(x) = \tau$:

$$\dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}, \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}}{(x\ x) ::?}}{\lambda x.(x\ x) :: \tau \to ?}$$

An der Stelle für das Fragezeichen, müsste gelten $\tau = \tau_1 \to \tau_2$ für das linke x und gleichzeitig $\tau = \tau_1$ für das rechte x. Dies ist jedoch unmöglich. Der Ausdruck ist auch in Haskell nicht typisierbar.

Als weiteres Beispiel betrachten wir den Prozess

$$id = \lambda x.x \mid y = id\ \texttt{True} \mid z = id\ \texttt{Nil}.$$

Der Prozess ist nicht typisierbar, da $\Gamma(id)$ ein monomorpher Typ sein muss, er jedoch einmal als $\texttt{Bool} \to \texttt{Bool}$ (in der Bindung für y) und einmal als $\texttt{List}\ \tau \to \texttt{List}\ \tau$ (in der Bindung für z) benötigt würde. Mit einem polymorphen Typsystem wäre der Prozess typisierbar, da wir dann den polymorphen Typ $\alpha \to \alpha$ für $\Gamma(id)$ wählen dürften. Der Prozess

$$id_1 = \lambda x_1.x_1 \mid id_2 = \lambda x_2.x_2 \mid y = id_1\ \texttt{True} \mid z = id_2\ \texttt{Nil}$$

ist jedoch in *CHF* typisierbar, für $\Gamma(id_1) = \texttt{Bool} \to \texttt{Bool}$ und $\Gamma(id_2) = \texttt{List}\ \tau \to \texttt{List}\ \tau$ (wobei τ beliebig wählbar ist).

6.3.3 Operationale Semantik

Wir definieren nun die Auswertung von Prozessen. Wir benutzen hierfür eine small-step operationale Semantik. In einem Reduktionsschritt wird dabei nur ein Thread einen Auswertungsschritt machen (u.U. unter Zuhilfenahme von weiteren Komponenten, wie MVars und Bindungen). Wir definieren die call-by-need Auswertung von CHF, in (Sabel & Schmidt-Schauß, 2011a) ist auch eine alternative call-by-name Auswertung zu finden. Zunächst benötigen wir verschiedene Klassen von Kontexten, die dabei helfen, die richtige Position zu finden, an der reduziert wird.

Auf Prozessebene definieren wir *Prozesskontexte* $\mathbb{D} \in \mathit{PC}$ durch die folgende Grammatik:

$$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$$

Die Grammatik drückt aus, dass Prozesskontexte gebildet werden, indem wir an irgendeiner Stelle in einem Prozess anstelle eines Unterprozesses das Kontextloch schreiben. Beachte, dass Prozesskontexte daher sozusagen Prozesse mit einem „Prozessloch“ sind, d.h. wir können Prozesse in dieses Loch einsetzen. Ein weiterer interessanter Fakt ist, dass für einen Prozesskontext $\mathbb{D}$ und einen Prozess P, der Prozess $\mathbb{D}[P]$ immer wohlgetypt ist, wenn $\mathbb{D}[P]$ wohlgeformt ist und $\mathbb{D}$ und P selbst wohlgetypt sind. Der Grund dafür liegt darin, dass wir die Typen der Variablen als fest vorgegeben (durch die Funktion Γ) gewählt haben.

Für die Ausdrucksebene definieren wir zunächst die monadischen Kontexte $\mathbb{M} \in \mathit{MC}$ durch die Grammatik

$$\mathbb{M} \in \mathit{MC} ::= [\cdot] \mid \mathbb{M} \gg\!= e$$

Diese dienen dazu in einer Sequenz von $>\!\!>\!\!=$ -Operationen, das linkeste Argumente zu finden, d.h. in $e_1 >\!\!>\!\!= e_2 >\!\!>\!\!= e_3 >\!\!>\!\!= \ldots >\!\!>\!\!= e_n$ möchten wir später den Redex ganz links in e_1 finden.

Für funktionale Ausdrücke verwenden wir zunächst die schon bekannten call-by-name Reduktionskontexte (ähnlich zum Lambda-Kalkül), die bei der Anwendung und bei case-Ausdrücken links ins erste Argument springen. Diese Evaluationskontexte $\mathbb{E} \in \mathit{EC}$ sind definiert durch die Grammatik

$$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ alts) \mid (\texttt{seq}\ \mathbb{E}\ e)$$

Manchmal reicht es für Ausdrücke jedoch nicht aus, allein diese Kontexte zu verwenden, denn bei den Operationen `takeMVar` und `putMVar` müssen wir sicherstellen, dass das erste Argument zu einem Namen einer MVar ausgewertet wird. Daher führen wir als weitere Klasse die *Forcing Kontexte* $\mathbb{F} \in \mathit{FC}$ ein, die entweder normale *EC*-Kontexte sind, oder in das erste Argument von `takeMVar` bzw. `putMVar` hineingehen.

$$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$$

Beachte, dass *MC*-, *EC*- und *FC*-Kontexte Ausdrücke mit einem „Ausdrucksloch“ sind, d.h. ein Ausdruck kann in sie eingesetzt werden und dann entsteht ein Ausdruck.

Schließlich fehlt noch der Übergang von Prozessen zu Ausdrücken. Der einfache Fall ist, dass direkt in einem Thread reduziert werden kann, d.h. die Reduktion findet in an einer Position statt, die ein Kontext der Form $x \Leftarrow \mathbb{M}[\mathbb{F}]$ beschreibt. Da wir jedoch eine call-by-need Strategie verfolgen, ist es manchmal nötig, entlang einer Kette von globalen Bindungen $x_1 = e_1 \mid \ldots, x_n = e_n$ nach der richtigen Reduktionsstelle zu suchen. Diese Suche geht von einem Thread aus, d.h. wir betrachten Kontexte der Form $x \Leftarrow \mathbb{M}[\mathbb{F}[x_1]] \mid x_1 = e_1 \mid \ldots, x_n = \mathbb{E}[\cdot]$. Da wir zwei Varianten dieser Kontexte benötigen, definieren wir sowohl $\mathbb{L} \in \mathit{LC}$ als auch die Variante $\widehat{\mathbb{L}} \in \widehat{LC}$:

$$\begin{aligned}
&\mathbb{L} \in \mathit{LC} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \\
&\quad \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\
&\qquad\qquad \text{wobei } \mathbb{E}_2, \ldots \mathbb{E}_n \text{ nicht der leere Kontext sind.} \\
&\widehat{\mathbb{L}} \in \widehat{LC} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \\
&\quad \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\
&\qquad\qquad \text{wobei } \mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n \text{ nicht der leere Kontext sind.}
\end{aligned}$$

Der Unterschied zwischen *LC*- und $\widehat{LC}$ ist klein: Für *LC*-Kontexte darf der letzte Kontext in der Kette $\mathbb{E}_1$ leer sein, während dieser bei $\widehat{LC}$ nicht leer sein darf. Beachte, dass *LC*- und $\widehat{LC}$-Kontexte Prozesse mit einem „Ausdrucksloch“ sind, d.h. man kann in das Loch einen Ausdruck einsetzen und es entsteht ein Prozess.

In *CHF* ist ein *funktionaler Wert* eine Abstraktion oder eine Konstruktoranwendung, ein *Wert* ist ein funktionaler Wert oder ein monadischer Ausdruck. D.h. auf der funktionalen Ebene werden monadische Ausdrücke wie Werte (genauer: wie Konstruktoranwendungen) behandelt, auf der obersten Ebene eines Threads, werden die monadischen Ausdrücke jedoch ausgeführt.

Die Auswertungsregeln der Standardreduktion $\xrightarrow{\mathit{CHF}}$ sind nun wie folgt definiert, wobei wir diese in zwei Klassen teilen: Regeln für monadische Berechnungen und Regeln für die funktionale Auswertung.

Monadische Berechnungen:

$(CHF,\text{lunit})\quad y \Leftarrow \mathbb{M}[\texttt{return}\ e_1 \gg\!= e_2] \xrightarrow{CHF} y \Leftarrow \mathbb{M}[e_2\ e_1]$

$(CHF,\text{tmvar})\quad y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{CHF} y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\,\mathbf{m}\,-$

$(CHF,\text{pmvar})\quad y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\,\mathbf{m}\,- \xrightarrow{CHF} y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\,\mathbf{m}\,e$

$(CHF,\text{nmvar})\quad y \Leftarrow \mathbb{M}[\texttt{newMVar}\ e] \xrightarrow{CHF} \nu x.(y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x\,\mathbf{m}\,e)$

$(CHF,\text{fork})\quad y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{CHF} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
wobei z ein neuer Name ist und der erzeugte Thread kein Main-Thread ist

$(CHF,\text{unIO})\quad y \Leftarrow \texttt{return}\ e \xrightarrow{CHF} y = e$, wenn Thread y kein Main-Thread ist

Funktionale Auswertung:

$(CHF,\text{cp})\quad \widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{CHF} \widehat{\mathbb{L}}[v] \mid x = v,$
falls v eine Abstraktion oder eine Variable ist

$(CHF,\text{cpcx})\quad \widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots e_n,$
$\xrightarrow{CHF} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
falls c ein Konstruktor, oder ein monadischer Operator ist

$(\text{mkbinds})\quad \mathbb{L}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e]$
$\xrightarrow{CHF} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

$(CHF,\text{lbeta})\quad \mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{CHF} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

$(CHF,\text{case})\quad \mathbb{L}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots (c\ y_1\ \ldots\ y_n \to e)\ldots]$
$\xrightarrow{CHF} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

$(CHF,\text{seq})\quad \mathbb{L}[(\texttt{seq}\ v\ e)] \xrightarrow{CHF} \mathbb{L}[e]$ wenn v ein funktionaler Wert ist

Zusätzlich nehmen wir an, dass $\xrightarrow{CHF}$ abgeschlossen bzgl. Prozesskontexten und struktureller Kongruenz ist, d.h.

Wenn $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$ und $P_1' \xrightarrow{CHF} P_2'$, dann gilt auch $P_1 \xrightarrow{CHF} P_2$

Wir beschreiben und erläutern die Reduktionsregeln. Wir betrachten zunächst die Regeln zu monadischen Berechnungen: Die Regel (lunit) entspricht dem ersten Monadengesetz und dient dazu in einer Sequenz aus Aktionen $a_1 \gg\!= a_2$ fortzufahren, wenn die Aktion a_1 beendet ist (und daher von der Form `return` e ist). Dann wird die nächste Aktion durch Anwendungen von a_2 auf das Resultat e erzeugt. Die Regeln (tmvar) und (pmvar) dienen dem Ausführen einer `takeMVar`- bzw `putMVar`-Operation. Beachte, dass die Regeln nur anwendbar sind, wenn die MVar gefüllt bzw. leer ist. In anderen Fällen (z.B. `takeMVar`, aber die MVar ist leer) ist keine Reduktion möglich, was das Blockieren des Threads modelliert. Die Regel (nmvar) dient dem Erzeugen einer neuen MVar mittels der `newMVar`-Operation. Die Regel (fork) führt eine `future`-Operation aus: Eine neuer Thread wird erzeugt und der ausführende Prozess erhält den Namen der neu erzeugten Future. Falls der ausführende Prozess den Wert der Future benötigt, wird dieser solange Warten müssen, bis die Future ihre Berechnung beendet hat und das Ergebnis mittels der Regel (unIO) als globale Bindung zur Verfügung steht. Beachte, dass wir die (unIO)-Regel nicht für den Hauptthread verwenden.

Die Regeln zur funktionalen Auswertung verfolgen die call-by-need Strategie. Daher werden nur *benötigte Werte* kopiert. Für Abstraktionen und Variablen wird dieses Kopieren durch die Regel (cp) durchgeführt. Müssen Konstruktoranwendungen oder monadische Ausdrücke kopiert

werden, so werden dies nicht voll kopiert, sondern die Argumente werden durch neue globale Bindungen geshared. Dies vermeidet Doppelauswertungen. Die passende Regel dazu ist (cpcx). Die Regel (mkBinds) schiebt lokale `letrec`-Bindungen auf die Prozessebene und macht aus diesen globale Bindungen. Die Regel (lbeta) ist ähnlich zur Beta-Reduktion im Lambda-Kalkül, allerdings wird das Argument nicht in den Rumpf der Abstraktion eingesetzt, sondern durch eine neue globale Bindung geshared. Auch dies dient der Vermeidung von Doppelauswertungen. Die (case)-Regel wertet einen `case`-Ausdruck aus und vermeidet Doppelauswertungen, indem die Argumente der Konstruktoranwendung durch neue Bindungen geshared werden. Schließlich kann mit der (seq)-Regel ein `seq`-Ausdruck ausgewertet werden, wenn das erste Argument ein funktionaler Wert ist. Beachte, dass aufgrund der Typisierung ein monadischer Ausdruck an dieser Stelle nicht auftreten kann.

Die Auswertung $\xrightarrow{CHF}$ ist nicht-deterministisch, d.h. es können mehrere Regeln auf einen Prozess anwendbar sein, allerdings kann pro Thread $x \Leftarrow e$ stets nur eine Regel anwendbar sein. Wir nehmen an, dass die Auswertung irgendeine der anwendbaren Regeln ausführt. Wir nehmen außerdem an, dass nur wohlgeformte Prozesse reduziert werden dürfen und dass die Auswertung stoppt wenn ein *erfolgreicher Prozess* erreicht wird:

Definition 6.3.2. *Ein wohlgeformter Prozess P ist genau dann* erfolgreich*, wenn er von der Form $\nu x_1, \ldots, x_n.x \xleftarrow{\text{main}} \texttt{return}\ e \mid P'$ ist.*

Mit $\xrightarrow{CHF,+}$ und $\xrightarrow{CHF,*}$ bezeichnen wir die transitive bzw. reflexiv-transitive Hülle von $\xrightarrow{CHF}$, oder anders ausgedrückt $\xrightarrow{CHF,+}$ steht für beliebig viele aber mindestens eine Reduktion und $\xrightarrow{CHF,*}$ steht für 0 oder mehr Reduktionen.

Aufgrund der nichtdeterministischen Reduktion reicht es zur Beurteilung des Verhaltens eines Prozesses nicht aus, nur danach zu schauen, ob es eine Reduktionsfolge gibt, die in einem erfolgreichen Prozess endet, sondern es muss auch betrachtet werden, ob man (durch Reduktion) zu einem Prozess gelangen kann, der sozusagen fehlerhaft ist und nicht mehr in einen erfolgreichen Prozess überführt werden kann. Wir führen daher zwei Konvergenzbegriffe für Prozesse ein. Der erste Begriff legt fest, ob es möglich ist, einen Prozess zu einem erfolgreichen Prozess zu reduzieren. Der zweite Begriff legt fest, ob nach beliebiger Reduktion der Prozess immer noch konvergieren kann.

Definition 6.3.3. *Ein Prozess P* may-konvergiert *genau dann, wenn er mit beliebig vielen Reduktionen in einen erfolgreichen Prozess überführt werden kann. Wir schreiben dann $P{\downarrow_{CHF}}$ und definieren dies formal als*

$$P{\downarrow_{CHF}} \text{ genau dann, wenn } \exists P' : P \xrightarrow{CHF,*} P' \text{ und } P' \text{ ist erfolgreich}$$

Ein Prozess P should-konvergiert *genau dann, wenn er nach beliebig vielen Reduktionen may-konvergent verbleibt. Wir schreiben dann $P{\Downarrow_{CHF}}$ und definieren dies formal als*

$$P{\Downarrow_{CHF}} \text{ genau dann, wenn } \forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow_{CHF}}$$

In der Literatur findet sich noch ein weiterer Konvergenzbegriff: Die sogenannte Must-Konvergenz: Ein Prozess P ist must-konvergent, wenn er should-konvergent ist und es keine unendliche lange Reduktion $P \xrightarrow{CHF} P_1 \xrightarrow{CHF} P_2 \xrightarrow{CHF} P_3 \ldots$ gibt. Der Begriff unterscheidet sich

von der Should-Konvergenz bzgl. der unendlichen langen Reduktionen, da es durchaus should-konvergente Prozesse gibt, die unendlich lange reduzieren können. Betrachte z.B. den Prozess P definiert als

$$
\begin{array}{l}
P := x \overset{\mathsf{main}}{\Longleftarrow} \texttt{future}\ (loopPut\ \texttt{True}) \gg\!= \lambda\_.\texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda\_.loop \\
\quad \mid loop = \texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad \lambda\_.\texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad\ \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop) \\
\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!=\ \lambda\_.loopPut\ z \\
\quad \mid x\ \mathbf{m} -
\end{array}
$$

Der Main-Thread erzeugt zunächst zwei nebenläufige Threads, wobei der erste wiederholt versucht `True` in die MVar x zu schreiben und der zweite wiederholt versucht `False` in die MVar x zu schreiben. Der Main-Thread führt danach wiederholt das folgende aus: Er liest zweimal die MVar und schaut sich das zweite Resultat an: Ist dieses `True`, so terminiert der Main-Thread sofort. Ist das Resultat `False`, so geht die Schleife weiter.

Daher lässt sich leicht nachvollziehen, dass es ein Interleaving gibt, sodass der Main-Thread stets `False` als zweites Resultat liest und daher endlos reduziert werden kann. Trotzdem gibt es stets die Möglichkeit noch in einem erfolgreichen Prozess zu enden. Daher ist P zwar should-konvergent, aber nicht P must-konvergent. Wir betrachten die Must-Konvergenz nicht weiter, da sie beweistechnisch eher schwierig handhabbar ist und sich trotz dieser feinen Unterschiede ähnlich zur Should-Konvergenz verhält.

Zur Verdeutlichung der Reduktion CHF zeigen wir einige mögliche Reduktionsschritte für den Prozess P:

$$
\begin{array}{l}
P := x \overset{\mathsf{main}}{\Longleftarrow} \texttt{future}\ (loopPut\ \texttt{True}) \gg\!= \lambda y_1.\texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda y_2.loop \\
\quad \mid loop = \texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad\ \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop) \\
\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!=\ \lambda\_.loopPut\ z \\
\quad \mid x\ \mathbf{m} - \\
\xrightarrow{CHF, fork} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ x_t \gg\!= \lambda y_1.\texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda y_2.loop \\
\qquad \mid loop = \texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad\ \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop) \\
\qquad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!=\ \lambda\_.loopPut\ z \\
\qquad \mid x\ \mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \\
\xrightarrow{CHF, lunit} x \overset{\mathsf{main}}{\Longleftarrow} (\lambda y_1.\texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda y_2.loop)\ x_t \\
\qquad \mid loop = \texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!= \\
\qquad\qquad\ \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop) \\
\qquad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!=\ \lambda\_.loopPut\ z \\
\qquad \mid x\ \mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True})
\end{array}
$$

$$\xrightarrow{\text{CHF},lbeta} x \overset{\text{main}}{\Longleftarrow} \texttt{future}\ (loopPut\ \texttt{False}) \gg\!= \lambda y_2.loop$$
$$\quad \mid loop = \texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad\quad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid y_1 = x_t$$

$$\xrightarrow{\text{CHF},fork} x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ x_f \gg\!= \lambda y_2.loop$$
$$\quad \mid loop = \texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad\quad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t$$

$$\xrightarrow{\text{CHF},lunit} x \overset{\text{main}}{\Longleftarrow} (\lambda y_2.loop)\ x_f$$
$$\quad \mid loop = \texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad\quad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t$$

$$\xrightarrow{\text{CHF},lbeta} x \overset{\text{main}}{\Longleftarrow} loop$$
$$\quad \mid loop = \texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad\quad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t \mid y_2 = x_f$$

$$\xrightarrow{\text{CHF},cpcx} x \overset{\text{main}}{\Longleftarrow} l_1 \gg\!= l_2$$
$$\quad \mid loop = l_1 \gg\!= l_2$$
$$\quad \mid l_1 = \texttt{takeMVar}\ x$$
$$\quad \mid l_2 = \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t \mid y_2 = x_f$$

$$\xrightarrow{\text{CHF},cpcx} x \overset{\text{main}}{\Longleftarrow} (\texttt{takeMVar}\ l_{1,1}) \gg\!= l_2$$
$$\quad \mid loop = l_1 \gg\!= l_2$$
$$\quad \mid l_1 = \texttt{takeMVar}\ l_{1,1} \mid l_{1,1} = x$$
$$\quad \mid l_2 = \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\qquad \lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\quad \mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\quad \mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t \mid y_2 = x_f$$

$$\xrightarrow{CHF,cp} x \xleftarrow{\mathsf{main}} (\texttt{takeMVar}\ x) \gg\!= l_2$$
$$\mid loop = l_1 \gg\!= l_2$$
$$\mid l_1 = \texttt{takeMVar}\ l_{1,1} \mid l_{1,1} = x$$
$$\mid l_2 = \lambda y_3.\texttt{takeMVar}\ x \gg\!=$$
$$\lambda w.\texttt{case}_{Bool}\ w\ \texttt{of}\ (\texttt{True} \to \texttt{return}\ \texttt{True})\ (\texttt{False} \to loop)$$
$$\mid loopPut = \lambda z.\texttt{putMVar}\ x\ z \gg\!= \lambda\_.loopPut\ z$$
$$\mid x\,\mathbf{m} - \mid x_t \Leftarrow (loopPut\ \texttt{True}) \mid x_f \Leftarrow (loopPut\ \texttt{False}) \mid y_1 = x_t \mid y_2 = x_f$$

Dieses Beispiel zeigt, dass die Reduktion syntaktisch ziemlich komplex ist und große Prozesse dabei entstehen. Dies geschieht insbesondere durch die Reduktionen, die neue Bindungen (aufgrund des Sharing) erzeugen.

Wir führen noch Notationen für die Negation der May- und der Should-Konvergenz ein:

Definition 6.3.4. *Sei P ein Prozess. Wenn P nicht may-konvergent ist, so sagen wir P ist* must-divergent *und schreiben dies als $P{\Uparrow}_{CHF}$. Wenn P nicht should-konvergent ist, so sagen wir P ist* may-divergent *und schreiben dies als $P{\uparrow}_{CHF}$.*

Satz 6.3.5. *Für alle Prozesse P gilt: $P{\uparrow}_{CHF} \iff \exists P' : P \xrightarrow{CHF,*} P' \wedge P{\Uparrow}_{CHF}$*

Beweis. $P{\uparrow}_{CHF}$ ist äquivalent zu $\neg P{\Downarrow}_{CHF}$, was wiederum äquivalent zu $\neg(\forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow}_{CHF})$ ist. Durch prädikatenlogische Umformung erhalten wir zunächst $(\exists P' : \neg(P \xrightarrow{CHF,*} P' \implies P'{\downarrow}_{CHF}))$, anschließend $(\exists P' : P \xrightarrow{CHF,*} P' \wedge P'\neg{\Downarrow}_{CHF}))$ und schließlich da Must-Divergenz die Negation der May-Konvergenz ist $(\exists P' : P \xrightarrow{CHF,*} P' \wedge P'{\Uparrow}_{CHF})$. □

Satz 6.3.5 zeigt, dass man für may-Divergenz (genau wie für May-Konvergenz) Induktion verwenden kann, da nur eine endliche Reduktionsfolge betrachtet werden muss.

6.3.4 Gleichheit von Prozessen

Nachdem wir nun die Auswertung von Prozessen definiert haben, stellt sich die Frage, welche Prozesse man in CHF als „gleich“ bzw. als ungleich betrachten soll. Wir verwenden für den Begriff der Gleichheit die *kontextuelle Gleichheit*, wobei wir aufgrund des Nichtdeterminismus nicht nur testen, ob ein Prozess terminieren kann, sondern zusätzlich die Should-Konvergenz betrachten.

Definition 6.3.6. *Die* kontextuelle Approximation $\leq_{CHF}$ *auf Prozessen ist definiert als: $P_1 \leq_{CHF} P_2$ genau, dann wenn$P_1 \lesssim_{CHF} P_2$ und $P_1 \lesssim_{\Downarrow CHF} P_2$, wobei*

$$P_1 \lesssim_{CHF} P_2 \quad \textit{genau dann, wenn } \forall \mathbb{D} \in \mathbf{PC} : \mathbb{D}[P_1]{\downarrow}_{CHF} \implies \mathbb{D}[P_2]{\downarrow}_{CHF}$$
$$P_1 \lesssim_{\Downarrow CHF} P_2 \quad \textit{genau dann, wenn } \forall \mathbb{D} \in \mathbf{PC} : \mathbb{D}[P_1]{\Downarrow}_{CHF} \implies \mathbb{D}[P_2]{\Downarrow}_{CHF}$$

Oder informell ausgedrückt: P_2 approximiert P_1 kontextuell, wenn das Konvergenzverhalten von P_2 bezüglich May- und Should-Konvergenz mindestens genauso gut ist wie das Konvergenzverhalten von P_1.

Die Kontextuelle Gleichheit *(oder alternativ: Kontextuelle Äquivalenz) $\sim_{CHF}$ auf Prozessen ist definiert als:*

$$P_1 \sim_{CHF} P_2 \textit{ genau dann, wenn } P_1 \leq_{CHF} P_2 \textit{ und } P_2 \leq_{CHF} P_1$$

Würde man die Gleichheit ausschließlich auf May-Konvergenz aufbauen, so könnte man offensichtlich verschiedene Prozesse nicht unterscheiden. Wir betrachten als Beispiel die Prozesse P_1, P_2:

$$P_1 := \nu z.(z \xLeftarrow{\text{main}} \texttt{return True})$$
$$P_2 := \nu x, z, y_1, y_2, loop.$$
$$(z \xLeftarrow{\text{main}} \texttt{takeMVar}\ x \ggg= \lambda w.\texttt{case}_{Bool}\ w\ (\texttt{True} \rightarrow \texttt{return True})\ (\texttt{False} \rightarrow loop)$$
$$\mid loop = loop \mid y_1 \Leftarrow \texttt{putMVar}\ x\ \texttt{False} \mid y_2 \Leftarrow \texttt{putMVar}\ x\ \texttt{True} \mid x\,\mathbf{m}\,-)$$

Man kann leicht nachvollziehen, dass $\mathbb{D}[P_1]$ für jeden Prozesskontext $\mathbb{D}$ direkt erfolgreich ist, also auch may-konvergent ist. Für $\mathbb{D}[P_2]$ kann man stets eine Reduktionsfolge angeben, in welcher der Thread y_1 den Wert True in die MVar x schreibt und der Main-Thread nach einigen Schritten mit return True endet. Darum gilt auch für alle $\mathbb{D}$-Kontexte $\mathbb{D}[P_2]\downarrow_{CHF}$ und daher sind P_1 und P_2 bezüglich der kontextuellen Gleichheit aufgebaut nur aus der May-Konvergenz nicht zu unterscheiden. Unser Begriff der kontextuellen Gleichheit unterscheidet P_1 und P_2 jedoch, da z.B. im leeren Kontext $P_1\Downarrow_{CHF}$, aber $\neg P_2\Downarrow_{CHF}$ denn wir können so reduzieren, dass y_2 den Wert False in die MVar x schreibt, danach kann der Main-Thread nicht mehr erfolgreich werden.

Auch die alleinige Betrachtung der Should-Konvergenz reicht nicht aus: Wähle $P_1 := \nu z, loop.(z \xLeftarrow{\text{main}} loop) \mid loop = loop)$ und P_2 wie vorher. Dann kann man zeigen, dass weder $\mathbb{D}[P_1]$ noch $\mathbb{D}[P_2]$ should-konvergent für jeden beliebigen Prozesskontext $\mathbb{D}$ sind. Daher wären beide Prozesse gleich bzgl. der kontextuellen Gleichheit, die nur auf Should-Konvergenz testet. Da P_1 auch nicht may-konvergent ist und P_2 jedoch may-konvergent ist, unterscheidet unsere Definition der kontextuellen Äquivalenz beide Prozesse.

6.3.5 Fairness

Die bisher definierte Standardreduktion von *CHF* beachtet keinerlei Fairness, insbondere auch nicht die Fairnessannahme aus Abschnitt 1.5.1. Z.B. erlaubt daher die Standardreduktion den Prozess

$$x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\,\mathbf{m}\,\texttt{True} \mid y \Leftarrow loop \mid loop = loop$$

unendlich lange zu reduzieren, indem immer Thread y einen Schritt machen darf, aber nie Thread x:

$$\begin{array}{ll} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\,\mathbf{m}\,\texttt{True} \mid y \Leftarrow loop \mid loop = loop \\ \xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\,\mathbf{m}\,\texttt{True} \mid y \Leftarrow loop \mid loop = loop \\ \xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\,\mathbf{m}\,\texttt{True} \mid y \Leftarrow loop \mid loop = loop \\ \xrightarrow{CHF,cp} & x \xLeftarrow{\text{main}} \texttt{takeMVar}\ z \mid z\,\mathbf{m}\,\texttt{True} \mid y \Leftarrow loop \mid loop = loop \\ \xrightarrow{CHF,cp} & \ldots \end{array}$$

Die (cp)-Reduktion kopiert dabei stets die Variable $loop$. Für eine echte Implementierung wäre diese *unfaire* Reduktionsfolge nicht wünschenswert, und man würde Fairness fordern. Wir formalisieren nun den Begriff der fairen Auswertung, wobei wir auch alternative Definitionen für die Begriffe May- und Should-Konvergenz einführen.

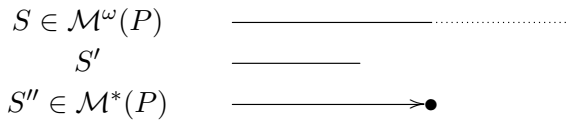
Definition 6.3.7. *Für einen Prozess P sei $\mathcal{M}(P)$ die Menge aller* maximalen *Reduktionsfolgen bzgl. der Reduktion $\xrightarrow{CHF}$, d.h. alle endlichen Reduktionsfolgen, die mit P starten und mit einem*

irreduziblen Prozess enden, sowie alle unendlich langen Reduktionsfolgen, die mit P starten. Mit $\mathcal{M}^\omega(P)$ notieren wir die Menge aller unendlich *langen Reduktionssequenzen aus $\mathcal{M}(P)$, und mit $\mathcal{M}^*(P)$ die Menge aller endlichen Reduktionsfolgen aus $\mathcal{M}_P$ (d.h. $\mathcal{M}^\omega(P) = \mathcal{M}(P) \setminus \mathcal{M}^*(P)$ und $\mathcal{M}^*(P) = \mathcal{M}(P) \setminus \mathcal{M}^\omega(P)$).*

Mithilfe dieser Notation lassen sich May- und Should-Konvergenz auch wie folgt alternativ – aber gleichwertig zur vorherigen Definition – definieren:

Definition 6.3.8.

- *Ein Prozess P ist genau dann may-konvergent, wenn $\mathcal{M}^*(P)$ eine Folge enthält, die mit einem erfolgreichen Prozess endet.*
- *Ein Prozess P ist genau dann should-konvergent, wenn alle Folgen aus $\mathcal{M}^*(P)$ mit einem erfolgreichen Prozess enden und für jede unendliche Folge $S \in \mathcal{M}^\omega(P)$ gilt: Für jeden endlichen Präfix S' von S gibt es eine endliche Reduktionsfolge $S'' \in \mathcal{M}^*(P)$, sodass S' ein Präfix von S'' ist. Als Bild dargestellt:*



Man kann leicht nachvollziehen, dass diese Definition der ursprünglichen Definition von May- und Should-Konvergenz entspricht.

Um nun Fairness zu formalisieren, legen wir zunächst fest, wann ein Thread *ausführbar* ist: Für einen Prozess $P \equiv \mathbb{D}[x \Leftarrow e]$ ist der Thread x *ausführbar*, wenn es eine Reduktion $P \xrightarrow{CHF} P'$ gibt, die entweder innerhalb des Ausdrucks e reduziert, oder eine Reduktion ausführt, an der Thread x beteiligt ist (z.B. eine MVar liest oder schreibt, oder in e wird der Wert einer Bindung kopiert).

Nun definieren wir, wann an eine Reduktionsfolge *unfair* ist:

Definition 6.3.9. *Für einen Prozess P ist die Reduktionsfolge $S \in \mathcal{M}(P)$* unfair*, wenn S einen unendlich langen Suffix S' hat, in welchem ein Thread x unendlich oft ausführbar ist, aber niemals reduziert wird. Eine Reduktionsfolge, die nicht unfair ist, bezeichnen wir als faire Reduktionsfolge.*

Beachte, dass daher alle endlichen Reduktionsfolgen stets fair sind. Wir definieren nun faire May- und Should-Konvergenz:

Definition 6.3.10. *Für einen Prozess P sei $\mathcal{M}_f(P)$ die Menge aller fairen Reduktionsfolgen, die mit P starten. Ferner sei $\mathcal{M}_f^*(P)$ die Menge aller endlichen fairen Reduktionsfolgen für P und $\mathcal{M}_f^\omega(P)$ die Menge aller unendlich langen fairen Reduktionsfolgen. Faire May- und Should-Konvergenz sind nun genauso definiert wie May- und Should-Konvergenz in Definition 6.3.8, wobei die Mengen $\mathcal{M}(P), \mathcal{M}^*(P), \mathcal{M}^\omega(P)$ durch die fairen Mengen $\mathcal{M}_f(P), \mathcal{M}_f^*(P), \mathcal{M}_f^\omega(P)$ ersetzt werden:*

- *Ein Prozess P ist genau dann fair may-konvergent (geschrieben als $P{\downarrow_{CHF,f}}$), wenn $\mathcal{M}_f^*(P)$ eine Folge enthält, die mit einem erfolgreichen Prozess endet.*
- *Ein Prozess P ist genau dann fair should-konvergent (geschrieben als $P{\Downarrow_{CHF,f}}$), wenn alle Folgen aus $\mathcal{M}_f^*(P)$ mit einem erfolgreichen Prozess enden und für jede unendliche Folge $S \in \mathcal{M}_f^\omega(P)$ gilt: Für jeden endlichen Präfix S' von S gibt es eine faire endliche Reduktionsfolge $S'' \in \mathcal{M}_f^*(P)$ sodass S' ein Präfix von S'' ist.*

Da jede endliche Reduktionsfolge auch fair ist, folgt sofort $\mathcal{M}^*(P) = \mathcal{M}_f^*(P)$ und daher auch:

Satz 6.3.11. *Für alle Prozesse P gilt:* $P{\downarrow_{CHF}} \iff P{\downarrow_{CHF,f}}$.

Wir betrachten nun die Should-Konvergenz und zeigen zunächst die einfache Richtung:

Lemma 6.3.12. *Für alle Prozesse P gilt: Wenn $P{\Downarrow_{CHF}}$, dann gilt auch $P{\Downarrow_{CHF,f}}$*

Beweis. Sei P ein Prozess mit $P{\Downarrow_{CHF}}$. Wir zeigen $P{\Downarrow_{CHF,f}}$. Da jede endliche Reduktionsfolge beginnend mit P auch eine faire Reduktionsfolge ist, gilt $\mathcal{M}_f^*(P) = \mathcal{M}^*(P)$ und da alle Folgen in $\mathcal{M}^*(P)$ mit einem erfolgreichen Prozess enden, enden auch alle Folgen aus $\mathcal{M}_f^*(P)$ in einem erfolgreichen Prozess.
Sei $S \in \mathcal{M}_f^\omega(P)$ eine unendliche Folge. Da $\mathcal{M}_f^\omega(P) \subseteq \mathcal{M}^\omega(P)$, gilt auch $S \in \mathcal{M}^\omega(P)$. Sei nun S' ein endlicher Präfix von S. Dann gibt es eine Folge $S'' \in \mathcal{M}^*(P)$, sodass S' ein Präfix von S'' ist. Da $\mathcal{M}^*(P) = \mathcal{M}_f^*(P)$ folgt sofort $S'' \in \mathcal{M}_f^*(P)$. Da S, S' beliebig gewählt wurden, gilt $P{\Downarrow_{CHF,f}}$. □

Die Rückrichtung zu zeigen, d.h. dass aus fairer Should-Konvergenz auch die normale Should-Konvergenz folgt, ist etwas aufwändiger. Deshalb zeigen wir zunächst einen Hilfssatz, der besagt, dass man für jeden Prozess mindestens eine faire Reduktionsfolge angeben kann.

Lemma 6.3.13. *Für jeden Prozess P gibt es eine Reduktionsfolge $S \in \mathcal{M}_f(P)$.*

Beweis. Wenn es eine endliche maximale Reduktionsfolge beginnend mit P gibt, dann sind wir fertig und die Aussage stimmt. Anderenfalls gibt es nur unendlich lange Reduktionsfolgen für Prozess P. Ziel ist es jetzt zu zeigen, dass diese nicht alle unfair sind. Es reicht zu zeigen, dass es eine faire Reduktionsfolge gibt, die wir wiefolgt erzeugen: Wir merken uns für jede MVar x zwei FIFO-Queues:

- Eine take-Queue für blockierte `takeMVar`-Operationen (d.h. die Liste der dazugehörigen Futures)
- Eine put-Queue für blockierte `putMVar`-Operationen.

Außerdem ordnen wir die Threads in einer weiteren FIFO-Queue $\mathcal{Q}$.

Jede Reduktion wird nun nach dem folgenden Schema durchgeführt:

Sei x die erste Future in $\mathcal{Q}$:

- Wenn der zu x zugehörige Thread ausführbar ist, und die Reduktion ist keine (tmvar)- oder (pmvar)-Reduktion, dann reduziere den zugehörigen Thread und schiebe anschließend x an das Ende der Queue $\mathcal{Q}$. Wenn im Anschluss der Thread x eine `takeMVar`- oder `putMVar`-Operation durchführen will, dann hänge x zusätzlich an das Ende der entsprechenden take- oder put-Queue an. Wenn ein neuer Thread durch eine (fork)-Reduktion erzeugt wird, dann wird dieser Thread ebenfalls an das Ende der Queue $\mathcal{Q}$ angehängt (und entsprechend an eine take- oder put-Queue, wenn der neue Thread eine `takeMVar`- oder `putMVar`-Operation durchführen will).
- Wenn der zu x zugehörige Thread eine (tmvar)- oder (pmvar)-Reduktion durchführen kann, und der Thread der erste in der take- bzw. put-Queue ist, dann wird die Reduktion durchgeführt. Anschließend wir der Thread aus der take- bzw. put-Queue entfernt und an das Ende der Queue $\mathcal{Q}$ gehängt.

- Wenn der zu x zugehörige Thread nicht ausführbar ist, oder aber eine (tmvar)- or (pmvar)-Reduktion durchführen kann, aber nicht am Anfang der zugehörigen take- bzw. put-Queue steht, dann wird der Thread x an das Ende der Queue $\mathcal{Q}$ gehängt (und nicht reduziert).

Wir zeigen nun, dass auf diese Weise keine unfaire Reduktionsfolge startend mit P entstehen kann. Es ist unmöglich, dass es einen unendlich langen Suffix gibt, in der ein Thread ausfühbar ist, aber nie reduziert wird: Dies sichert die FIFO-queue $\mathcal{Q}$ zu, wobei bei (tmvar) und (pmvar)-Reduktionen evtl. mehrere Durchläufe durch die FIFO-Queue erforderlich sind. Die Anzahl der Durchläufe ist jedoch durch die entsprechende take- bzw. put-Queue beschränkt. □

Korollar 6.3.14. *Sei S eine endliche Reduktionssequenz startend mit Prozess P. Dann gibt es eine faire endliche Reduktionssequenz $S' \in \mathcal{M}_f(P)$, sodass S ein Präfix von S' ist.*

Beweis. Wir führen zunächst $P \xrightarrow{S} P'$ aus, und hängen anschließend die faire Reduktionssequenz entsprechend Lemma 6.3.13 startend mit P' daran. Die entstehende Reduktionssequenz ist offensichtlich fair, da $P \xrightarrow{S} P'$ nur eine endliche Sequenz ist. □

Nach dieser Vorabeit können wir zeigen, dass faire Should-Konvergenz auch die normale Should-Konvergenz impliziert:

Lemma 6.3.15. *Für alle Prozesse P: $P{\Downarrow_{CHF,f}} \implies P{\Downarrow_{CHF}}$.*

Beweis. Nehme an, dass $P{\Downarrow_{CHF,f}}$ gilt. Sei $S \in \mathcal{M}(P)$. Wenn S eine endliche Reduktionsfolge ist (d.h. $S \in \mathcal{M}^*(P)$), dann gilt auch $S \in \mathcal{M}_f^*(P)$ und da $P{\Downarrow_{CHF,f}}$ muss S mit einem erfolgreichen Prozess enden. Wenn S eine unendliche Reduktionsfolge ist (d.h. $S \in \mathcal{M}^\omega(P)$), dann wähle einen beliebigen endlichen Präfix S' von S. Korollar 6.3.14 zeigt, dass es eine faire Reduktionssequenz $S'' \in \mathcal{M}_f(P)$ gibt, sodass S' ein Präfix von S'' ist. Wenn S'' endlich ist, dann muss S'' mit einem erfolgreichen Prozess enden, und $S'' \in \mathcal{M}(P)$ muss ebenso gelten, da $\mathcal{M}_f(P) \subseteq \mathcal{M}(P)$. Wenn S'' eine unendliche Folge ist, dann folgt aus $P{\Downarrow_{CHF,f}}$, dass es eine endliche Folge $S''' \in \mathcal{M}_f^*(P)$ gibt, die S' als Präfix hat und die in einem erfolgreichen Prozess endet. Wiederum muss gelten $S''' \in \mathcal{M}^*(P)$, da $\mathcal{M}_f(P) \subseteq \mathcal{M}(P)$. Da wir S, S', S'' beliebig gewählt haben, folgt $P{\Downarrow_{CHF}}$. □

Insgesamt haben wir daher gezeigt:

Theorem 6.3.16. *Für alle Prozesse P gilt: $P{\downarrow_{CHF}} \iff P{\downarrow_{CHF,f}}$ und $P{\Downarrow_{CHF}} \iff P{\Downarrow_{CHF,f}}$.*

Da die Konvergenzprädikate durch Fairness nicht verändert werden, gilt auch:

Theorem 6.3.17. *Kontextuelle Äquivalenz in CHF bleibt unverändert, wenn unfaire Reduktionssequenzen verboten sind.*

Der Vorteil dieser Aussage ist, dass wir Programmgleichheiten beweisen können, ohne explizit auf Fairness zu achten. Alle Gleichheiten gelten jedoch auch für den Auswerter, der fair auswertet. Tatsächlich ist die Betrachtung aller (auch der unfairen) Reduktionsfolgen wesentlich einfacher, als sich ausschließlich auf die fairen Folgen zu konzentrieren, da die Fairness stets erfordern würde, die gesamte Folge aufeinmal zu betrachten.

6.3.6 Korrektheit von Programmtransformationen

Eine Programmtransformation ist eine binäre Relation auf Prozessen, d.h. sie formt Prozesse in andere Prozesse um. Z.B. kann dies zur Programmoptimierung (z.B. durch partielle Auswertung oder Entfernen von nicht mehr benötigtem Code) dienen. Eine Transformation T ist genau dann *korrekt*, wenn für alle $(P_1, P_2) \in T$ gilt $P_1 \sim_{CHF} P_2$. Korrektheit zu zeigen erfordert daher, kontextuelle Gleichheit nachzuweisen. Im Allgemeinen ist es einfach, kontextuelle Gleichheit von zwei Prozessen P_1, P_2 zu widerlegen, denn man muss nur *einen* Kontext $\mathbb{D}$ finden, für den $\mathbb{D}[P_1]$ und $\mathbb{D}[P_2]$ unterschiedliches Konvergenzverhalten bzgl. May- oder Should-Konvergenz aufweisen. Z.B. sind die Prozesse $P_1 := x \Leftarrow \texttt{return True}$ und $P_2 := x \Leftarrow \texttt{return False}$ kontextuell verschieden, da für $\mathbb{D} := y \overset{\text{main}}{\Longleftarrow} \texttt{case}_{Bool}\, x\, (\texttt{True} \to \texttt{returnTrue})\, (\texttt{False} \to \texttt{letrec}\, w = w\, \texttt{in}\, w) \mid [\cdot]$ gilt: $\mathbb{D}[P_1]{\downarrow_{CHF}}$ aber $\mathbb{D}[P_2]{\Uparrow_{CHF}}$.

Der Beweis von kontextueller Gleichheit ist im Allgemeinen wesentlich schwieriger, da *alle* Kontexte untersucht werden müssen, und dies unendlich viele Kontexte sind. Da das Halteproblem unentscheidbar ist, folgt auch sofort das im Allgemeinen kontextuelle Gleichheit (und Ungleichheit) unentscheidbar ist, denn die Frage, ob die Gleichung $P \sim_{CHF} x \overset{\text{main}}{\Longleftarrow} \texttt{letrec}\, y = y\, \texttt{in}\, y)$ für einen Prozess P gilt, genau das Halteproblem löst. Trotzdem kann man für manche Prozesse natürlich kontextuelle Gleichheit bzw. Ungleichheit nachweisen.

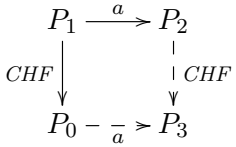
Wir beweisen im Folgenden einige Gleichheiten.

Satz 6.3.18. *Die Reduktionen* $(CHF, lunit)$, $(CHF, nmvar)$, $(CHF, fork)$, $(CHF, unIO)$, $(CHF, mkbinds)$ *sind korrekte Programmtransformationen.*

Beweis. Seien P_1, P_2 Prozesse sind mit $P_1' \xrightarrow{a} P_2'$ wobei $a \in \{(CHF, lunit), (CHF, nmvar), (CHF, fork)\}$ und $P_1 \equiv \mathbb{D}[P_1']$ und $P_2 \equiv \mathbb{D}[P_2']$ für einen *PC*-Kontext $\mathbb{D}$. Wir müssen vier Richtungen zeigen:

1. $P_1{\downarrow_{CHF}} \implies P_2{\downarrow_{CHF}}$
2. $P_2{\downarrow_{CHF}} \implies P_1{\downarrow_{CHF}}$
3. $P_1{\Downarrow_{CHF}} \implies P_2{\Downarrow_{CHF}}$
4. $P_2{\Downarrow_{CHF}} \implies P_1{\Downarrow_{CHF}}$

Untersucht man alle Überlappungen der Form $P_0 \xleftarrow{CHF} P_1 \xrightarrow{a} P_2$, indem man alle Fälle für die Standardreduktion durchgeht, so stellt man fest, dass die Reduktion und die Transformation „vertauschbar“ sind, d.h. es gibt einen Prozess P_3, so dass $P_0 \xrightarrow{a} P_3 \xleftarrow{CHF} P_2$, oder als sogenanntes Gabeldiagramm dargestellt:



Wir werden diese Eigenschaft im folgenden benutzen.

1. $P_1{\downarrow_{CHF}} \implies P_2{\downarrow_{CHF}}$: Wir nehmen an, dass $P_1{\downarrow_{CHF}}$ gilt, d.h. $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_{1,n}$ und zeigen $P_2{\downarrow_{CHF}}$ mit Induktion über n. Für die Induktionsbasis sei $n = 0$. Dann ist P_1 ein erfolgreicher Prozess und daher irreduzibel, d.h. die Aussage gilt. Für den Induktionsschritt wenden wir das Gabeldiagram auf $P_{1,1} \xleftarrow{CHF} P_1 \xrightarrow{a} P_2$ an. D.h. $P_{1,1} \xrightarrow{a} P_3 \xleftarrow{CHF} P_2$.

Da $P_{1,1}$ in weniger als n Schritten konvergiert, dürfen wir aus der Induktionshypothese schließen $P_3{\downarrow_{CHF}}$. Da $P_2 \xrightarrow{CHF} P_3$ folgt sofort auch $P_2{\downarrow_{CHF}}$.

2. $P_2{\downarrow_{CHF}} \implies P_1{\downarrow_{CHF}}$: Dies gilt sofort, da die Transformation $P_1 \xrightarrow{a} P_2$ auch eine Standardreduktion ist, und daher eine konvergente Reduktionsfolge für P_2 verlängert werden kann zu einer konvergenten Reduktionsfolge für P_1.
3. $P_1{\Downarrow_{CHF}} \implies P_2{\Downarrow_{CHF}}$: Wir zeigen die äquivalente Aussage $P_2{\uparrow_{CHF}} \implies P_1{\uparrow_{CHF}}$: Dies ist jedoch offensichtlich, da die Transformation $P_1 \xrightarrow{a} P_2$ auch eine Standardreduktion in CHF ist.
4. $P_2{\Downarrow_{CHF}} \implies P_1{\Downarrow_{CHF}}$: Wir zeigen die äquivalente Aussage $P_1{\uparrow_{CHF}} \implies P_2{\uparrow_{CHF}}$: Nehme an $P_1 \xrightarrow{CHF} P_{1,1} \xrightarrow{CHF} \ldots \xrightarrow{CHF} P_{1,n}$, wobei $P_{1,n}$ ein must-divergenter Prozess ist. Wir zeigen $P_2{\uparrow_{CHF}}$ mit Induktion über n. Für $n = 0$ ist P_1 schon must-divergent. Dann zeigt Teil (2) dieses Beweises, dass P_2 auch must-divergent ist (denn $P_2{\downarrow_{CHF}} \implies P_1{\downarrow_{CHF}}$ ist äquivalent zu $P_1{\Uparrow_{CHF}} \implies P_2{\Uparrow_{CHF}}$). Für den Induktionsschritt wenden wir das Gabeldiagramm auf $P_{1,1} \xleftarrow{CHF} P_1 \xrightarrow{a} P_2$ an. D.h. $P_{1,1} \xrightarrow{a} P_3 \xleftarrow{CHF} P_2$. Da $P_{1,1}$ in weniger als n Schritten in einen must-divergenten Prozess überführt werden kann, dürfen wir aus der Induktionshypothese schließen $P_3{\uparrow_{CHF}}$. Da $P_2 \xrightarrow{CHF} P_3$ folgt sofort auch $P_2{\uparrow_{CHF}}$.

□

6.3.7 Gleichheiten und Programmtransformationen für Ausdrücke

Wir erweitern zunächst die kontextuelle Gleichheit auf Ausdrücke. Seien *CC* alle Prozesse, die an einer Ausdrucksposition ein Loch haben.

Definition 6.3.19. *Kontextuelle Approximation $\leq_{CHF}$ und kontextuelle Gleichheit $\sim_{CHF}$ für gleich getypte Ausdrücke ist in CHF definiert also: $\leq_{CHF} := \lesssim_{CHF} \cap \lesssim_{\Downarrow CHF}$ und $\sim_{CHF} := \leq_{CHF} \cap \geq_{CHF}$, wobei für Ausdrücke e_1, e_2 vom Typ τ:*

$$e_1 \lesssim_{CHF} e_2 \quad \textit{gdw.} \quad \forall \mathbb{C}[\cdot^\tau] \in \textit{CC} : \mathbb{C}[e_1]{\downarrow_{CHF}} \implies \mathbb{C}[e_2]{\downarrow_{CHF}}$$
$$e_1 \lesssim_{\Downarrow,CHF} e_2 \quad \textit{gdw.} \quad \forall \mathbb{C}[\cdot^\tau] \in \textit{CC} : \mathbb{C}[e_1]{\Downarrow_{CHF}} \implies \mathbb{C}[e_2]{\Downarrow_{CHF}}$$

Ein Programmtransformation T auf Ausdrücken ist eine binäre Relation auf Ausdrücken, sodass für $(e_1, e_2) \in T$ stets gilt e_1 uns e_2 sind vom gleichen Typ. Eine Programmtransformation T auf Ausdrücken ist *korrekt*, gdw. für alle $(e_1, e_2) \in T$ gilt $e_1 \sim_{CHF} e_2$. In (Sabel & Schmidt-Schauß, 2011a) wurde die Korrektheit von einigen Programmtransformationen gezeigt. Wir verzichten an dieser Stelle auf eine vollständige Auflistung. Erwähnenswert ist jedoch, dass die drei monadischen Gesetze in CHF gelten:

Satz 6.3.20. *In CHF gelten für alle (korrekt getypten) Ausdrücke e_1, e_2, e_3 die folgenden Gleichheiten:*

$$\begin{array}{lll} \texttt{return}\, e_1 \gg\!= e_2 & \sim_{CHF} & e_2\, e_1 \\ e_1 \gg\!= \lambda x.\texttt{return}\, x & \sim_{CHF} & e_1 \\ e_1 \gg\!= (\lambda x.(e_2\, x \gg\!= e_3)) & \sim_{CHF} & (e_1 \gg\!= e_2) \gg\!= e_3 \end{array}$$

6.4 Quellennachweis

Zum Lambda-Kalkül gibt es viele Quellen. Ein nicht ganz leicht verständliches Standardwerk ist (Barendregt, 1984). Ein gute Einführung ist in (Hankin, 2004) zu finden. Zum π-Kalkül sind

(Milner, 1999) und (Sangiorgi & Walker, 2001) die Standardwerke. Die Kodierung des Lambda-Kalküls in den π-Kalkül ist aus (Milner, 1992) entnommen. Die Kodierung des synchronen π-Kalküls in den asynchronen π-Kalkül stammt aus (Boudol, 1992). Die Gleichheitsdefinitionen sind im Wesentlichen in (Sangiorgi & Walker, 2001) nachzulesen.

Der CHF-Kalkül mit Futures wurde in (Sabel & Schmidt-Schauß, 2011a; Sabel & Schmidt-Schauß, 2011b) eingeführt und analysiert. Weitere Resultate und Untersuchungen zum CHF-Kalkül sind in (Sabel & Schmidt-Schauß, 2011c; Sabel & Schmidt-Schauß, 2012; Sabel, 2012; Schmidt-Schauß et al., 2018) zu finden. Arbeiten zur kontextuellen Äquivalenz mit may- und should-Konvergenz (insbesondere der hier verwendeten Definition der should-Konvergenz) sind in (Carayol et al., 2005; Rensink & Vogler, 2007; Niehren et al., 2007; Sabel & Schmidt-Schauß, 2008; Sabel, 2008; Schmidt-Schauß & Sabel, 2010) zu finden.

Literatur

Barendregt, H. P. (**1984**). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York.

Ben-Ari, M. (**2006**). *Principles of concurrent and distributed programming*. Addison-Wesley, Harlow, UK.

Bird, R. S. (**1998**). *Introduction to Functional Programming Using Haskell*. Prentice-Hall.

Boudol, G. (**1992**). Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA.

Buhr, P. A., Fortier, M., & Coffin, M. H. (**1995**). Monitor classification. *ACM Comput. Surv.*, 27(1):63–107.

Carayol, A., Hirschkoff, D., & Sangiorgi, D. (**2005**). On the representation of McCarthy's amb in the pi-calculus. *Theoretical Computer Science*, 330(3):439–473.

Carriero, N. & Gelernter, D. (**1992**). *How to Write Parallel Programs*. MIT Press, Cambridge, MA. http://www.lindaspaces.com/book/.

Church, A. (**1941**). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.

Dice, D., Shalev, O., & Shavit, N. (**2006**). Transactional locking ii. In S. Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer.

Dziuma, D., Fatourou, P., & Kanellou, E. (**2015**). Consistency for transactional memory computing. In R. Guerraoui & P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, volume 8913 of *Lecture Notes in Computer Science*, pages 3–31. Springer.

Engelfriet, J. & Gelsema, T. (**2007**). An exercise in structural congruence. *Inf. Process. Lett.*, 101(1):1–5.

Fischer, M. J., Lynch, N. A., & Paterson, M. S. (**1985**). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

Gelernter, D. (**1985**). Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112.

Guerraoui, R. & Kapalka, M. (**2008**). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184. ACM, New York, NY, USA.

Guerraoui, R. & Kapalka, M. (**2010**). *Principles of Transactional Memory*. Morgan and Claypool Publishers, 1st edition.

Hankin, C. (**2004**). *An introduction to lambda calculi for computer scientists*. Number 2 in Texts in Computing. King's College Publications, London, UK.

Harris, T., Marlow, S., Peyton-Jones, S., & Herlihy, M. (**2005**). Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, New York, NY, USA.

Herlihy, M. (**1991**). Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149.

Herlihy, M. & Shavit, N. (**2008**). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hoare, C. A. R. (**1969**). An axiomatic basis for computer programming. *Communications*

of the ACM, 12(10):576–580.

Larus, J. R. & Rajwar, R. (**2006**). *Transactional Memory*. Morgan & Claypool.

Larus, J. R. & Rajwar, R. (**2010**). *Transactional Memory, 2nd edition*. Morgan & Claypool.

Marlow, S. (**2011**). Parallel and concurrent programming in Haskell. In V. Zsók, Z. Horváth, & R. Plasmeijer, editors, *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, volume 7241 of *Lecture Notes in Computer Science*, pages 339–401. Springer.

Marlow, S. (**2013**). *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. O'Reilly.

Milner, R. (**1992**). Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141.

Milner, R. (**1999**). *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA.

Niehren, J., Sabel, D., Schmidt-Schauß, M., & Schwinghammer, J. (**2007**). Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electronic Notes in Theoretical Computer Science*, 173:313–337.

O'Sullivan, B., Goerzen, J., & Stewart, D. (**2008**). *Real World Haskell*. O'Reilly Media, Inc.

Palamidessi, C. (**1997**). Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–265. ACM, New York, NY, USA.

Palamidessi, C. (**2003**). Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical. Structures in Comp. Sci.*, 13(5):685–719.

Peyton Jones, S., editor (**2003**). *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, New York, NY, USA. `www.haskell.org`.

Peyton-Jones, S. (**2007**). Beautiful concurrency. In A. Oram & G. Wilson, editors, *Beautiful code*. O'Reilly.

Peyton Jones, S. & Singh, S. (**2009**). A tutorial on parallel and concurrent programming in haskell. In P. Koopman, R. Plasmeijer, & D. Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 267–305. Springer. Nonoisbn-13 978-3-642-04651-3.

Peyton Jones, S. L. (**2001**). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, & R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, Amsterdam, The Netherlands.

Peyton Jones, S. L., Gordon, A., & Finne, S. (**1996**). Concurrent Haskell. In *POPL '96: 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308. ACM, St Petersburg Beach, Florida.

Raynal, M. (**2013**). *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer.

Rensink, A. & Vogler, W. (**2007**). Fair testing. *Information and Computation*, 205(2):125–198.

Reppy, J. H. (**2007**). *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England.

Sabel, D. (**2008**). *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, J. W. Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik.

Sabel, D. (**2012**). An abstract machine for concurrent haskell with futures. In S. Jähnichen, B. Rumpe, & H. Schlingloff, editors, *Software Engineering 2012 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin*, volume 199 of *LNI*, pages 29–44. GI.

Sabel, D. & Schmidt-Schauß, M. (**2008**). A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Mathematical Structures in Computer Science*, 18(3):501–553.

Sabel, D. & Schmidt-Schauß, M. (**2011**a). A contextual semantics for Concurrent Haskell with futures. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP '11, pages 101–112. ACM, New York, NY, USA.

Sabel, D. & Schmidt-Schauß, M. (**2011**b). A contextual semantics for Concurrent Haskell with futures. Frank report 44, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.

Sabel, D. & Schmidt-Schauß, M. (**2011**c). On conservativity of Concurrent Haskell. Frank report 47, Institut für Informatik, Goethe-Universität Frankfurt am Main.

Sabel, D. & Schmidt-Schauß, M. (**2012**). Conservative concurrency in haskell. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 561–570. IEEE.

Sangiorgi, D. & Walker, D. (**2001**). *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.

Schmidt-Schauß, M., Rau, C., & Sabel, D. (**2013**). Algorithms for Extended Alpha-Equivalence and Complexity. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 255–270. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Schmidt-Schauß, M. & Sabel, D. (**2010**). Closures of may-, should- and must-convergences for contextual equivalence. *Information Processing Letters*, 110(6):232 – 235.

Schmidt-Schauß, M. & Sabel, D. (**2013**). Correctness of an stm haskell implementation. In G. Morrisett & T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 161–172. ACM.

Schmidt-Schauß, M., Sabel, D., & Dallmeyer, N. (**2018**). Sequential and parallel improvements in a concurrent functional programming language. In D. Sabel & P. Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, pages 20:1–20:13. ACM, New York, NY, USA.

Taubenfeld, G. (**2006**). *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Varacca, D. & Völzer, H. (**2006**). New perspectives on fairness. *Bulletin of the EATCS*, 90:90–108.