# 19. Interagierende Prozesse: CCS

Prozesse werden in Ihrem Studium in diversen Variationen immer wiederkehrende Begleiter sein. Sie werden in etwas verallgemeinerter Form oft auch als *Automaten* bezeichnet. Wir nennen in diesem Zusammmenhang die Knoten des Graphen auch *Zustände*, die Kanten *Übergänge* oder *Transitionen*, und die Markierungen werden wir als *Aktionen* bezeichnen.

## 19.1 Ein Beispiel

Wir werden im folgenden ihre Verwendung zur Beschreibung des *Verhaltens* interagierender Prozesse beispielhaft detaillieren, um die darauf folgenden Gedanken (und insbesondere den Sprachgebrauch) darzulegen.

In der Form, in der wir Automaten bzw. Prozesse hier motivieren, können Sie sie sich am besten als Beschreibungen des *Verhaltens* einer Schnittstelle (Treiber, Device-driver etc) zu einem externen Dienst ansehen, zum Beispiel zu einem Course-Management System, oder dergleichen. Oder zu einem Cruise-Control (oder Tempomat) System im Auto. Wir verwenden das letztere als ein realistisches Szenario um einige Konzepte zu illustrieren. Das mögliche Verhalten eines Tempomat im Auto hängt von der Interaktion mehrerer Teile ab, die *nebenläufig* sind, und teilweise miteinander interagieren. Wir fokussieren uns zunächst auf den eigentlichen *CONTROLLER* und werden anschließend noch einige weitere Komponenten betrachten. Wir zielen darauf ab, das Verhalten dieser Teile jeweils abstrakt als Prozesse (also Elemente von $L$) zu beschreiben. Danach wenden wir uns der Frage zu, wie sich aus dem Verhalten der Teile das Verhalten des Gesamtsystemes ergibt.

### 19.1.1 Der Controller

Wir interessieren uns zunächst für die zentrale Komponente, den *Controller*. Für diesen beschreiben wir sein *Verhalten* zunächst informell – und sehr stark vereinfacht – wie folgt.

- Im Zustand *IDLE* wartet der *Controller* auf das Einschalten (*on?*).
- Der Controller quittiert das erfolgreiche Einschalten mit einem *ok!*.
- Durch Betätigen der Bremse (*brake?*) kann der Tempomat temporär suspendiert werden. Der Controller geht in den Zustand *SUSPEND* über.

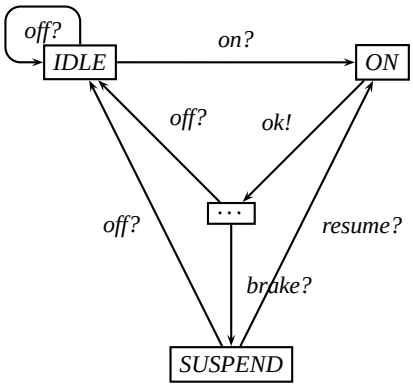



Abbildung 19.1: Ein sehr einfacher Tempomat.

– Im Zustand *SUSPEND* ist es möglich, mit *resume* die ursprüngliche Geschwindigkeit wieder aufzunehmen. Dies wird vom *Controller* durch einen *ok!* quittiert.
– Auschalten (*off?*) ist immer möglich, und führt zum Zustand *IDLE*.

Mit Hilfe einiger definierender Gleichungen, können wir uns das so intendierte Verhalten des Tempomats wie folgt darstellen:

$$\begin{array}{lcl} \mathit{IDLE} & = & \mathit{on?.ON} + \mathit{off?.\ IDLE} \\ \mathit{ON} & = & \mathit{ok!.(off?.\ IDLE} + \mathit{brake?.\ SUSPEND)} \\ \mathit{SUSPEND} & = & \mathit{resume?.ON} + \mathit{off?.\ IDLE} \end{array}$$

Die Semantik von $L$ liefert uns dazu den Prozess in Abb. 19.1, wobei wir einen Zustand durch '...' abgekürzt haben. Wir haben es mit einer Menge $M$ zu tun, die zumindest die Aktionen *off?*, *on?*, *brake?*, *resume?* und *ok!* umfasst.

**Aufgabe 19.1.** Bestimmen Sie den durch '...' abgekürzten Zustand, also den Term von $L$, der sein Verhalten beschreibt. Überprüfen Sie, ob der dargestellte Prozess alle die Eigenschaften hat, die wir informell postuliert haben. Wenn nicht, ändern Sie die definierenden Gleichungen entsprechend ab.

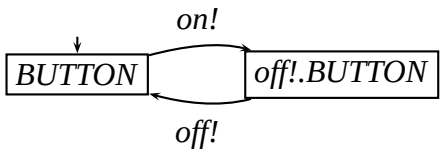
Wir haben in dem obigen Beispiel eine natürliche Unterscheidung in diejenigen Aktionen gefällt, die von dem Controller selbst initiiert werden (durch "*!*" gekennzeichnet), und denjenigen, die Reaktionen auf Interaktionen mit der Umgebung darstellen (durch "*?*" gekennzeichnet). Wir nennen erstere oft *Output*-Aktionen und letztere oft *Input*-Aktionen des Prozesses.

### 19.1.2 Die Umgebung

Der *CONTROLLER* des Tempomats interagiert mit seiner Umgebung. Diese Umgebung wird in unserem Beispiel durch einen Schalter am Armaturenbrett gebildet. Der Prozess *BUTTON*

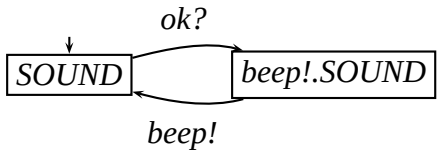
interagiert mit dem Controller in sehr einfacher Weise: Über den *BUTTON* wird von Zeit zu Zeit der Tempomat eingeschaltet (*on!*) und später wieder abgeschaltet (*off!*). Wir entwerfen für dieses Verhalten folgende Gleichung:

$$BUTTON = on!.off!.BUTTON$$

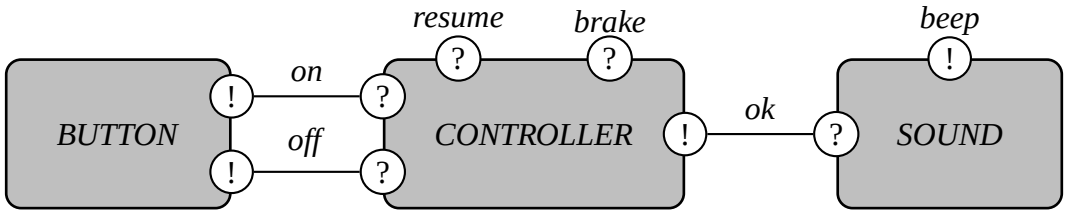



Ausserdem gibt es noch ein akustisches *beep!*-Signal, welches signalisiert, daß ein *ok?*. empfangen wurde.

$$SOUND = ok?.beep!.SOUND$$




Diese drei Prozesse sollen für den Moment als Beispiel genügen. Um zu verstehen, wie die Aktionen der Prozesse *SOUND*, *BUTTON* und *CONTROLLER* miteinander in Beziehung stehen, verwenden wir folgende Veranschaulichung:




Dieses schematische Bild der Interaktionen dient allein illustrativen Zwecken: Wir sehen, dass die Aktionen des Prozesses *BUTTON* komplementär zu denen von *CONTROLLER* sind; die Output-Aktionen des einen Prozesses passen zu den Input-Aktionen des anderen. Genauso verhält es sich mit dem Prozess *SOUND*. Für drei Aktionen gibt es keine Entsprechungen: die Output-Aktion *beep!*, sowie die Input-Aktionen *resume?* und *brake?*.

Wir werden uns bald auf die Suche nach Operatoren machen, die es uns erlauben, die bildlich dargestellten Zusammenhänge mit einer Sprache auszudrücken, und wir werden diesen Operatoren eine Semantik geben. Zunächst wollen wir jedoch noch unsere Intuition schärfen, welches Verhalten wir von derart interagierenden Prozessen erwarten.

*Nebenläufigkeit:* Betrachten wir zunächst die beiden Prozesse *SOUND* und *BUTTON*, und nehmen wir an, der *CONTROLLER* sei abwesend. Die beiden Prozesse sind dann in keiner Weise voneinander abhängig, und wir dürfen annehmen, daß jeder von beiden seine Aktionen unabhängig vom anderen Prozess ausführen kann, und dabei zwischen Zuständen wechseln kann. Wir sprechen hier von *Nebenläufigkeit*.

*Synchronisation:* Auch die Prozesse *CONTROLLER* und *SOUND* sind weitestgehend unabhängig voneinander. Einzige Ausnahme ist hier die In-/Ouput-Aktion *ok*. Diese Aktion,

kann von beiden gemeinsam ausgeführt werden, wobei beide einen simultanen Zustandswechsel vollziehen[1]. Man spricht hier von einer *Synchronisation* der beiden Prozesse.

Eine Synchronisation von Prozessen kann erfolgen, wenn Paare von passenden Input- und Ouput-Aktionen aufeinandertreffen.

Wir schliessen diese mehr exemplarischen Betrachtungen mit einer Beobachtung: Bisher haben wir einzelne Prozesse (in Form von markierten, gewurzelten Graphen) betrachtet, und eine Sprache dafür entwickelt. Nun ist es an der Zeit eine Sprache für interagierende nebenläufige Prozesse zu entwickeln, und mit Semantik zu versehen.

## 19.2 Grundlegendes

Auf der Basis der Sprache $L$ führen wir nun eine Sprache für nebenläufige und kooperierende Systeme ein. Statt einer beliebigen Menge $M$ von Markierungen verwenden wir eine Menge, die wie bereits angedeutet, mehr Struktur aufweist:

$$M = \mathcal{A}^! \cup \mathcal{A}^? \cup \{\tau\},$$

wobei wir die Bedeutung von $\tau$ erst später kennen lernen werden. $\mathcal{A}^!$ ist eine Menge von Output-Aktionen, und $\mathcal{A}^?$ eine Menge von Input-Aktionen, mit der Eigenschaft, daß eine Output-Aktion $a!$ genau dann in $\mathcal{A}^!$ ist, wenn die entsprechende Input-Aktion $a?$ in $\mathcal{A}^?$ ist. Input- und Output-Aktionen treten also als Paare auf, wir sagen sie sind *komplementär* zueinander. Wir verwenden im folgenden $m$ für beliebige Elemente von $M$, und $\alpha$ für beliebige Aktionen außer $\tau$, also $\alpha \in \mathcal{A}^! \cup \mathcal{A}^?$. Für ein solches $\alpha$ verwenden wir $\overline{\alpha}$ um sein Komplement zu erhalten, also aus einer Input-Aktion eine Output-Aktion zu machen, und umgekehrt. Ausserdem müssen wir manchmal auch $a$ verwenden, um auf die zu "nackte" Markierung von $\alpha$ zuzugreifen, also wenn $\alpha = a!$ oder $\alpha = a?$.

### 19.2.1 Kooperation

Nach diesen Vorarbeiten wenden wir uns dem Kern dieses Kapitels zu. Bisher haben wir die Grundkonzepte informell und mit Hilfe eines Beispieles eingeführt, welches nicht zu abstrakt war, und daher als Vehikel für unsere Gedanken fungieren konnte. Wir haben in diesem Beispiel einige Grundkonzepte kennengelernt:

– Prozesse (im echten Leben) haben Zustände. In diesen Zuständen können sie Aktionen ausführen, und dabei Zustandswechsel durchführen.

– Aktionen selbst sind *atomar*, und dienen zur Interaktion zwischen Prozessen.

– Verschiedene Prozesse können nebenläufig zueinander existieren, und Aktionen ausführen.

[1] Wir postulieren hier einen simultanen Zustandswechsel, um einer langwierigen – wenn auch fundamentalen – Diskussion über die Essenz von Kommunikation und Interaktion zu entgehen.

– Interaktion zwischen Prozessen kann erfolgen, wenn Paare von passenden Input- und Ouput-Aktionen aufeinandertreffen. Dies führt zu einer Synchronisation der beteiligten Prozesse, das heißt, zu einem simultanen Zustandswechsel in den beteiligten Prozessen.

Wir werden nun den zentralen Operator unserer Sprache einführen, mit welchem Parallelität und Interaktion abstrakt repräsentiert werden. Dieser Operator heißt *Kooperationsoperator* (oft auch Parallel-Operator genannt), und wir werden diesen mit dem Symbol '|' notieren. Wir haben es also, wenn $P, Q \in L$ bei dem Term $P \mid Q$ mit zwei kooperierenden Prozessen zu tun, beispielsweise *CONTROLLER* und *BUTTON*.

Wir gehen nun der Frage nach, welches Verhalten wir von diesen kooperierenden Prozessen erwarten. Wir wollen also die *Semantik* dieses Operators festlegen. Wie gehabt werden wir diese durch geeignete Inferenzregeln festlegen. Doch welche sind geeignet? Wir dürfen natürlich annehmen, daß wir das Verhalten der Prozesse $P$ und $Q$ jeweils kennen.

Zunächst wollen wir die Essenz der Nebenläufigkeit festhalten, nämlich, daß beide Prozesse sich unabhängig voneinander entscheiden können, eine Aktion auszuführen, und dabei einen Zustandswechsel vorzunehmen. Wir formalisieren dies mit den folgenden Inferenzregeln:

$$\textsf{par\_l}\quad \frac{P \xrightarrow{m} P'}{P \mid Q \xrightarrow{m} P' \mid Q} \qquad\qquad \textsf{par\_r}\quad \frac{P \xrightarrow{m} P'}{Q \mid P \xrightarrow{m} Q \mid P'}$$

Diese beiden Regeln sind zueinander symmetrisch. Beachten Sie, daß in jeder der beiden Regeln, einer der beiden Prozesse eine Transition ausführen kann, während der andere an seinen aktuellen Zustand eingefroren bleibt. Die Transition kann natürlich nur dann ausgeführt werden, wenn sie aufgrund der Prämisse auch für den jeweiligen Prozeß möglich ist.

**Aufgabe 19.2.** Versuchen Sie, den Prozess für $(links!.0 \mid rechts!.0)$ abzuleiten, der sich aus den Regeln par_l, par_r, und der Regel prefix ergibt. Tun Sie dasselbe für $(links!.links!.links!.0 \mid rechts!.0)$, und für (*SOUND* | *BUTTON*). Die Beweisbäume sind jeweils nicht groß.

Nun bleibt noch eine Frage zu klären, nämlich, wie wir die Synchronisation zweier Prozesse semantisch fassen können. Aus dem bisher Gesagten kann man entnehmen, daß beide Prozesse dann simultan einen Zuständswechsel vornehmen können, wenn sie komplementäre Aktionen ausführen können. Dies deutet daraufhin, dass wir es mit einer Inferenzregel der folgenden Art zu tun haben. Diese hat zwei Prämissen, um die komplementäre Bedingungen für beide Teile festzuhalten:

$$\textsf{sync}\quad \frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\cdots} P' \mid Q'}$$

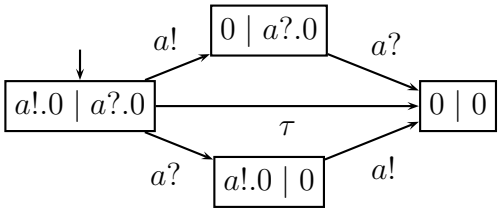
Allerdings wartet nun eine entscheidende Frage auf eine Antwort: Welches ist die richtige Markierung, die anstelle der Markierung '. . .' in der Regel sync verwendet werden sollte? Sollte dies $\alpha$ sein? Oder das Komplement $\overline{\alpha}$? Oder generell $a?$ oder $a!$? Nun, in jedem dieser Varianten ist

es möglich, daß ein dritter Prozess, z.B. $R$ in $(P \mid Q) \mid R$ an der Synchronisation teilnimmt. Man hat es dann mit einer "öffentlichen" Kooperation zu tun, die für Dritte in gewissem Sinne sichtbar ist. Dies ist manchmal erwünscht, oft jedoch nicht.

Daher gehen wir einen anderen Weg, und machen die Annahme, die Synchronisation sei prinzipiell nicht beobachtbar. Dazu spendieren für alle erfolgreichen Synchronisationen eine einzige eigene Markierung: $\tau$. Diese steht für ein unbeobachtbares Verhalten, und wir nennen $\tau$ auch die *unsichtbare* Aktion. Insbesondere kann dieses $\tau$ nicht in der Prämisse der Inferenzregel sync eingesetzt werden, es ist also unmöglich, diese zur Synchronisation zu nutzen. Wir halten die folgenden drei Regeln als Semantik des Operators '|' fest, wobei die Unterscheidung in $m$ (alles mögliche) und $\alpha$ (alles mögliche außer $\tau$) hier wesentlich ist.

$$\text{par\_l}\quad \frac{P \xrightarrow{m} P'}{P \mid Q \xrightarrow{m} P' \mid Q} \qquad \text{par\_r}\quad \frac{P \xrightarrow{m} P'}{Q \mid P \xrightarrow{m} Q \mid P'} \qquad \text{sync}\quad \frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Wir erhalten damit zum Beispiel für den $(a!.0 \mid a?.0)$ folgenden Prozeß:

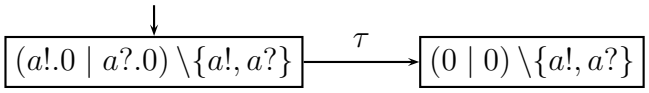



**Aufgabe 19.3.** Zeichnen sie den von $((a!.0 \mid a!.0) \mid a?.0)$ erzeugten Prozeß. Wie steht es mit dem von $((a!.0 \mid a!.0) \mid X)$, wobei $X = a?.X$ sei? Macht es etwas aus, wenn Sie den Prozess zu $((a!.0 \mid X) \mid a!.0)$ permutieren?

### 19.2.2 Restriktion

Wie aus den obigen Beispielen deutlich wird, erzwingt die Semantik von '|' nicht, daß eine mögliche Synchronisation tatsächlich stattfindet. Dies ist sinnvoll, um zu erlauben, daß mehrere Prozesse um die Synchronisation mit einem Partner konkurrieren. Um sich den Nutzen dieser Tatsache zu veranschaulichen, denken Sie an die gemeinsame Benutzung eines Druckers. Um die Synchronisation zu erzwingen gibt es einen weiteren Operator. Dieser *Restriktions*-Operator '$\backslash$' kann bestimmte (Paare von) Aktionen eines Prozesses unterbinden. '$\backslash$' ist daher ein zweistelliger Operator, der einerseits als Argument einen Term $P$ und andererseits eine Menge von Aktionen $H \subseteq M$ als Argument besitzt. Seine Syntax ist also $P \setminus H$.

Wir erhalten damit zum Beispiel für $(a!.0 \mid a?.0) \setminus \{a!, a?\}$ den folgenden Prozeß:



$$
\begin{array}{l}
m \in M \\
X \in Var \\
H \subseteq M \\
P \in CCS \;=\; 0 \;\mid\; m.P \;\mid\; P + P \;\mid\; X \;\mid\; P \mid P \;\mid\; P \setminus H
\end{array}
$$

Abbildung 19.2: Abstrakte Grammatik von $CCS$.

Die Semantik dieses Operators ist leicht durch eine einzige Inferenzregel zu definieren. [2]

$$\mathsf{res}\frac{P \xrightarrow{m} P' \qquad m \notin H}{P \setminus H \xrightarrow{m} P' \setminus H}$$

**Aufgabe 19.4.** Zeichnen Sie den von $((a!.0 \mid a!.0) \mid a?.0) \setminus \{a!, a?\}$ erzeugten Prozeß.

## 19.3 Syntax und Semantik von CCS

Nachdem wir alle wesentlichen Konstrukte vorgeführt haben, ist es nun an der Zeit, die Sprache und ihre Semantik endgültig festzuzurren. Die Sprache hört auf den Namen $CCS$ für *Calculus of Communicating Systems*. Die abstrakte Grammatik von $CCS$ ist in Abb. 19.2 angegeben, und enthält keine Überraschungen. Die Semantik der Terme von $CCS$ ist wie zuvor bezüglich einer partiellen Funktion $\Gamma \in Var \rightharpoonup CCS$ definiert, und verwendet eine kaskadierte Funktion, welche als erstes Argument die rekursiven Gleichungen gemäß $\Gamma$ erhält, und als zweites Argument den Term $P \in CCS$, welcher als Anfangsknoten fungieren soll.

$$
\begin{array}{l}
[\![\ \_\ ]\!] \in\ (Var \rightharpoonup CCS) \to CCS \to \mathcal{G}_{CCS} \times CCS \\
[\![\ \_\ ]\!]\ \Gamma\ P = ((CCS, M, \to_\Gamma), P)
\end{array}
$$

Dabei ist wiederum $\mathcal{G}_{CCS} = \{(CCS, M, E) \mid E \subseteq CCS \times M \times CCS\}$, und $\to_\Gamma$ ist die kleinste Relation, die den Regeln aus Abb. 19.3 genügt. Wir wollen nun noch einige Klammersparregeln einführen:

| | | | |
|---|---|---|---|
| $P \mid Q \mid R$ | $\rightsquigarrow$ | $(P \mid Q) \mid R$ | $\mid$ klammert links |
| $a.P \mid Q$ | $\rightsquigarrow$ | $(a.P) \mid Q$ | Punkt vor Strich |
| $P + Q \mid R$ | $\rightsquigarrow$ | $(P + Q) \mid R$ | $+$ vor $\mid$ |

Beachten Sie, dass wir vereinbaren, dass Terme, auf die der Restriktionsoperator angewandt wird, immer geklammert geschrieben werden, z.B. $a.0 + (0)\ \{a!, a?\}, (a.O + 0) \setminus \{a!, a?\}$.

[2] Es macht gemeinhin Sinn, die folgenden Annahmen über zulässige Mengen $H$ zu machen:

– Die unsichtbare Aktion kann nicht unterbunden werden: $\tau \notin H$;
– Aktionen treten in $H$ paarweise auf: $a! \in H \iff a? \in H$.

$$\textsf{prefix}\ \frac{}{m.P \xrightarrow{m} P} \qquad \textsf{choice\_l}\ \frac{P \xrightarrow{m} P'}{P + Q \xrightarrow{m} P'} \qquad \textsf{choice\_r}\ \frac{Q \xrightarrow{m} Q'}{P + Q \xrightarrow{m} Q'}$$

$$\textsf{sync}\ \frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \textsf{par\_l}\ \frac{P \xrightarrow{m} P'}{P \mid Q \xrightarrow{m} P' \mid Q} \qquad \textsf{par\_r}\ \frac{P \xrightarrow{m} P'}{Q \mid P \xrightarrow{m} Q \mid P'}$$

$$\textsf{res}\ \frac{P \xrightarrow{m} P' \qquad m \notin H}{P \setminus H \xrightarrow{m} P' \setminus H} \qquad \textsf{rec}\ \frac{\Gamma(X) = P \quad P \xrightarrow{m} P'}{X \xrightarrow{m} P'}$$

Abbildung 19.3: Semantik von $CCS$ für $\Gamma \in Var \rightharpoonup CCS$

**Aufgabe 19.5.** Geben Sie einen Term $P \in CCS$ an, der die in Abschnitt 19.1.2 eingeführten Gleichungen für *CONTROLLER*, *BUTTON*, und *SOUND* als partielle Funktion $\Gamma$ verwendet, und die gewünschten Interaktionen erzwingt. Stellen Sie experimentell fest, wie der Prozess $Reach(\llbracket\ P\ \rrbracket_\Gamma)$ aussieht.

**Aufgabe 19.6.** Betrachten Sie die folgende Menge rekursiver Gleichungen $\Gamma$.

$$\begin{aligned}
CDWriter &= getW?.putW?.CDWriter\\
CDReader &= getR?.putR?.CDReader\\
User1 &= getR!.getW!.rip!.burn!.putW!.putR!.User1\\
User2 &= getW!.getR!.rip!.burn!.putR!.putW!.User2
\end{aligned}$$

Untersuchen Sie experimentell $Reach(\llbracket\ (CDReader \mid CDWriter \mid User1 \mid User2) \setminus H\ \rrbracket_\Gamma)$, wobei $H$ alle vorkommenden Aktionen außer $rip!$, $rip?$, $burn!$ und $burn?$ enthalten soll. Was beobachten Sie, wenn Sie versuchen, diesen Prozeß bis zur Tiefe 3 zu explorieren?

**Aufgabe 19.7.** Betrachten Sie die folgende Menge rekursiver Gleichungen $\Gamma$.

$$\begin{aligned}
Fork1 &= getF1?.putF1?.Fork1\\
Fork2 &= getF2?.putF2?.Fork2\\
Fork3 &= getF3?.putF3?.Fork3
\end{aligned}$$

$$\begin{aligned}
PhilA &= getF1!.getF2!.eat!.putF1!.putF2!.think!.PhilA\\
PhilB &= getF2!.getF3!.eat!.putF2!.putF3!.think!.PhilB\\
PhilC &= getF3!.getF1!.eat!.putF3!.putF1!.think!.PhilC
\end{aligned}$$

Ergooglen Sie '*Dining Philosophers*'. Untersuchen Sie experimentell

$$Reach(\llbracket\ (Fork1 \mid PhilA \mid Fork2 \mid PhilB \mid Fork3 \mid PhilC\ \rrbracket_\Gamma) \setminus H$$

wobei $H$ alle vorkommenden Aktionen außer $eat!$, $eat?$, $think!$ und $think?$ enthalten soll. Nach welcher Spur sind Sie Zeuge des Hungertodes der Philosophen?

**Aufgabe 19.8.** Erweitern Sie die Sprache $CCS$ um einen zweistelligen Operator $P[f]$, wobei $P \in CCS$ und $f \in M \rightharpoonup M$ eine partielle Funktion auf Aktionen ist, für die gilt, daß $f(\alpha) = \beta \iff f(\overline{\alpha}) = \overline{\beta}$ und $f(\tau) = \tau$. Dieser Operator soll es erlauben, die Aktionen von $P$ gemäß der Funktion $f$ umzubenennen. Wie sehen die nötigen Inferenzregeln aus?

Wir werden in Zukunft die Unterscheidung zwischen einem Term $P \in CCS$ und seinem Prozess $Reach([\![ P ]\!]_\Gamma)$ verwischen, und insbesondere letzteres mit dem Begriff 'der Prozess $P$' belegen.

## 19.4 Interagierende Prozesse konkret

Im Laufe der Vorlesung haben wir die Sprachen Standard ML und $CCS$ kennengelernt. Wir haben uns bisher jedoch hauptsächlich damit beschäftigt, wie man mit einer solchen Sprache arbeitet. Aber auch, vor allem am Beispiel von $CCS$, wie die formalen Grundlagen einer solchen Sprache aussehen. Im folgenden werden wir an Hand der Sprache $CCS$ lernen, wie man aus der formalen Definition einer Sprache eine benutzbare Implementierung erstellt.

Erinnern wir uns, dass wir bei Programmiersprachen zwischen semantischen und syntaktischen Aspekten unterscheiden. In diesem Abschnitt wollen wir uns zuerst ansehen, wie die Objekte, die durch die abstrakte Syntax gegeben sind, in SML repräsentiert werden, und wie die dazugehörigen semantischen Regeln implementiert werden. Dazu wollen wir aus der abstrakten Grammatik von $CCS$ zuerst geeignete Datenstrukturen ableiten, die die einzelnen Objekte unserer Sprache repräsentieren.

### 19.4.1 Datenstrukturen

Die mathematischen Objekte, die wir benutzt haben, um $CCS$ zu definieren, müssen wir in geeigneter Weise in SML abbilden. Fassen wir zunächt zusammen, welche Objekte wir verwendet haben:

– Eine Menge $M$, bestehend aus Output-Aktionen ($\mathcal{A}^!$), Input-Aktionen ($\mathcal{A}^?$) und $\tau$;

– eine Menge $Var$ von Rekursionvariablen;

– Terme der Sprache $CCS$, gegeben durch die abstrakte Grammatik in Abb. 19.2;

– eine Menge von rekursiven Gleichungen $\Gamma \in Var \rightharpoonup CCS$;

– die semantische Abbildung $[\![\ ]\!] \in (Var \rightharpoonup CCS) \to CCS \to \mathcal{G}_{CCS} \times CCS$.

Die meisten dieser Objekte können wir direkt als Datenstruktur in SML abbilden. In Abb. 19.4 sehen Sie die grundlegenden Datentypen unserer Implementierung. Insbesondere die abstrakte Grammatik der Sprache können wir in Standard ML direkt durch eine Typdeklaration (`datatype`) beschreiben, wie wir es bereits bei der Darstellung arithmetischer Ausdrücke in

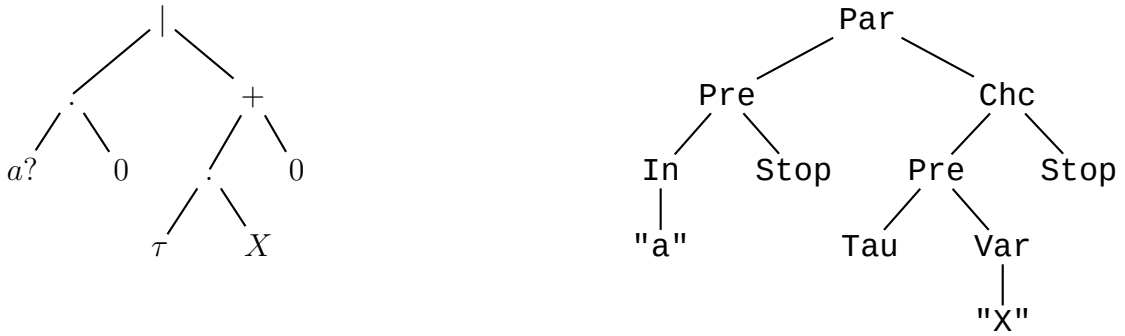
```
type name = string
type var = string

datatype mark = In of name | Out of name | Tau
type markset = mark list

datatype ccs = Stop
             | Var of var
             | Pre of mark * ccs
             | Chc of ccs * ccs
             | Par of ccs * ccs
             | Res of ccs * markset
```

Abbildung 19.4: Typdefinitionen für $CCS$

Kapitel 6.4 gesehen haben. Beachten Sie die deutlichen Übereinstimmungen zwischen der abstrakten Grammatik von $CCS$ aus Abb. 19.2, und Ihrer Implementierung in SML. Rekursionsvariablen und Aktionsnamen sind als String implementiert, und es gibt, wie erwartet, drei Arten von Aktionen. Jeder der Operatoren wird durch einen entsprechenden Konstruktor des Datentyps `ccs` implementiert. Mit diesen Typdefinitionen wird der $CCS$-Term $(a?.0) \mid (\tau.X + 0)$ beispielsweise durch

```
Par(Pre(In("a"),Stop),Chc(Pre(Tau,Var("X")),Stop))
```

deklariert. Die folgenden Syntaxbäume veranschaulichen den Zusammenhang:




**Aufgabe 19.9.** Repräsentieren Sie die Terme $a!.0$, $a!.(0 + a!.0)$, $a!.0 + a!.0$, $a!.a!.a?.a!.a!.0$, $a!.0 + (b?.0)$ und $a!.0 + a!.b?.X$ in SML gemäß der Typdefinitionen aus Abb. 19.4.

Wir haben nun zwei Fragen zu beantworten.

1. Wie kommt man von einem konkreten Folge von Zeichen zu einem Element der abstrakten Syntax von `ccs` in SML? Hier geht es also um den *Lexer* und den *Parser* der Sprache.
2. Wie kommt von einer Element der abstrakten Syntax von `ccs`, zu dem von diesem gemäß der Semantik erzeugten Graphen? Hier geht es also um den *Evaluator*, welcher die dynamische Semantik von $CCS$ implementiert.

Wir wenden uns hier nur der zweiten Frage zu. Die erste Frage ist eine Variante dessen, was in Kapitel 13 – insbesondere in Kapitel 13.5 – ausführlich diskutiert wird.

Die Kernfrage ist also, wie wir jetzt die (dynamische) Semantik von $CCS$ implementieren. Der direkte Weg führt uns dabei zu einer Prozedur $sem$, die eine Menge rekursiver Gleichungen $\Gamma$ und einen Term $P$ aus $CCS$ bekommt und dazu einen Prozess aus $\mathcal{G}_{CCS} \times CCS$ liefert, wobei wir uns auf den erreichbaren Teil $Reach[\![ P ]\!]_\Gamma$ beschränken wollen. [3]

Die partielle Funktion $\Gamma \in Var \rightharpoonup L$, welche die Menge der rekursiven Gleichungen umfasst, hat für $CCS$ eine ähnliche Bedeutung wie die *Umgebung* eines Standard ML Programmes (siehe Kapitel 2.4) oder arithmetischen Ausdruckes (siehe Kapitel 6.4): Sie enthält Bindungen der Bezeichner (hier Rekursionsvariablen) an Werte (hier $CCS$-Terme). Wir nennen $\Gamma$ daher im folgenden eine *Umgebung*, und werden mit dieser Umgebung ganz exakt so arbeiten, wie wir es für arithmetische Ausdrücke in Kapitel 6.4.2 gelernt haben.

### 19.4.2 Adjazenzmengen

Ein Prozess ist ein Tupel, welches in unserem Fall aus einem Graph und einem Term von $CCS$ besteht. Damit wir verstehen können, wie wir diesen Graph erzeugen, müssen wir zunächst lernen, wie wir überhaupt allgemein Graphen als Datenstruktur darstellen können.

Es gibt mehrere Möglichkeiten, einen Graph effektiv als Datenstruktur darzustellen. Eine mehr oder minder effektive Möglichkeit dieses für Bäume zu tun, haben Sie bereits im Kapitel 7 kennengelernt. Wir werden nun eine klassische Art, beliebige endliche Graphen darzustellen, erläutern. Diese beruht auf sogenannten *Adjazenzmengen*, Sie finden diese in ähnlicher Form auch in Kapitel 16.2 – für unmarkierte Graphen.

Bei der Darstellung eines Graphens $G = (V, M, E)$ mit Adjazenzmengen verwendet man eine Funktion $adj \in V \to \mathcal{P}(M \times V)$ die zu jedem Knoten in $V$ die Menge aller Nachfolger zusammen mit der jeweiligen Kantenmarkierung, die dorthin führt, liefert. Wir nennen $adj$ eine Adjazenzfunktion, und nennen das Tripel $(V, M, adj)$ eine Adjazenzmengendarstellung von $G$, falls $G = (V, M, E)$ ist und $adj$ folgende Eigenschaft erfüllt:

$$\forall v, v' \in V\ \forall a \in M : (v, a, v') \in E \iff (a, v') \in adj(v)$$

Es gilt also in der Adjazenzmengendarstellung, daß für jeden Knoten $v$, die durch $adj(v)$ bezeichnete Menge die – und nur die – Paare $(a, v')$ enthält, für die $(v, a, v')$ eine Kante ist. Anders gesagt enthält $adj(v)$ alle Nachfolger von $v$, jeweils als Paar zusammen mit der Kantenmarkierung. Beispielsweise liefert für $V = \{a!.0 + b!.0, 0\}$ die auf $V$ folgendermassen definierte Funktion $adj$ eine Adjazenzmengendarstellung des Graphen des Prozesses $Reach([\![ a!.0 + b!.0 ]\!])$:

$$\begin{aligned} adj(a!.0 + b!.0) &= \{(a!, 0), (b!, 0)\} \\ adj(0) &= \emptyset \end{aligned}$$

[3] Ohne die Einschränkung auf $Reach$ ist die Semantik eines Termes jeweils eine unendliche Menge, da die Knoten- und Kantenmengen unendlich sind. Diese Tatsache allein muß uns nicht erschrecken, da wir bereits wissen, wie wir unendliche Objekte elegant repräsentieren können. (Denken Sie an die Definition der unendlichen Folge aus Kapitel 8.9.4.) Nichtsdestotrotz ist es handlicher, endlichen Objekte zu erhalten.

**Aufgabe 19.10.** Geben Sie die Adjazenzfunktion zum Graph des Prozesses $Reach([\![\ a!.c?.0|(a?.0 + c!.0) + a!.0\ ]\!])$ an.

**Aufgabe 19.11.** Sei $(\mathbb{N}, \mathbb{N}, adj)$ ein Baum in Adjazenzmengendarstellung mit der Wurzel 0. Schreiben Sie folgende Prozeduren:

- `size: (int-> int*int list) -> int -> int`, die zu einer Adjazenzfunktion $adj$ und einer Wurzel $w$ die Größe des durch die Adjazenzfunktion und die Wurzel $w$ eindeutig bestimmten Baumes liefert.
- `depth: (int-> int*int list) -> int -> int`, die zu einer Adjazenzfunktion $adj$ und einer Wurzel $w$ die Tiefe des durch die Adjazenzfunktion und die Wurzel $w$ eindeutig bestimmten Baumes liefert.

### 19.4.3 Nachfolger als Adjazenzen

Um die Semantik von $CCS$ durch eine Funktion $sem$ zu realisieren, spielen die Inferenzregeln aus Abb. 19.3 eine zentrale Rolle, da durch diese – und nur durch diese – Transitionen abgeleitet werden können. Wir werden nun die Funktion $steps$, welches die Inferenzregeln gemäß Abb. 19.3 anwendet, so implementieren, daß alle für einen Term (bzw. Knoten bzw. Zustand) $P$ durch diese Regeln beweisbaren Transitionen in Form einer Adjazenzmenge gesammelt werden. Die Funktion $steps$ hat die Aufgabe, für eine Umgebung $\Gamma$ die Adjazenzmenge eines Termes $P$ zu liefern. Die Applikation der Funktion $steps\ \Gamma$ auf diesen Term liefert also stets die Menge *aller* Paare $(m, P') \in M \times CCS$, für die $P \xrightarrow{m} P'$ mit den Inferenzregeln bewiesen werden kann. Die definierenden Gleichungen der Funktion $steps$ sehen Sie in Abb. 19.5.

Die auf einen $CCS$-Term jeweils anwendbaren Regeln ergeben sich aus seiner Struktur, wir haben es hier also mit einer strukturellen Rekursion zu tun (vgl. Kapitel 6.4), wobei die Teilterme bei jeder Anwendung kürzer werden (Sind Sie da sicher?). Sofern mehrere Regeln anwendbar sind, wie bei $P + Q$ oder $P \mid Q$, ist es offensichtlich geboten, für jeden Teil die Rekursion auszuführen, und die erzeugten Adjazenzmengen zu vereinigen.

Wir wenden uns nun der eigentlichen Implementierung der Prozedur `steps` zu. Wir verwenden hier, der Einfachheit halber Listen anstelle von Mengen. Doppelauftreten sind daher ohne weiteres erlaubt. Man beachte hierbei, daß man ohne weiteres Listen als Mengen betrachten kann. So beschreiben $\{x, y\}$, $\{x, x, y\}$, $\{y, x, y\}$ dieselbe Menge, auch wenn sie in Listendarstellung verschieden sind.

Wir verwenden in unserer Implementierung eine Listendarstellung – und verweisen auf Kapitel 14.3 für eine interessante Diskussion der Vor- und Nachteile der Darstellung von Mengen als Listen. Daher reden wir im folgenden auch von *Adjazenzlisten*, und nicht von -mengen, und werden die Vereinigung von Mengen durch die Konkatenation der entsprechenden Listen umsetzen. In Abbildung 19.6 sehen Sie den Torso einer Implementierung der Prozedur `steps`, zusammen mit zwei Hilfsfunktionen `successors` und `complement`. Dabei erfüllt letztere die Aufgabe, zu einer gegebenen Aktion $\alpha \in M$ das Komplement $\overline{\alpha}$ zu liefern.

$$
\begin{array}{ll}
steps\ \Gamma\ 0 & = \emptyset \\
steps\ \Gamma\ m.P & = \{(m, P)\} \\
steps\ \Gamma\ X & = steps\ \Gamma\ (\Gamma X) \\
steps\ \Gamma\ (P + Q) & = (steps\ \Gamma\ P) \cup (steps\ \Gamma\ Q) \\
steps\ \Gamma\ (P|Q) & = \{(m, P'|Q) \quad | \quad (m, P') \in steps\ \Gamma\ P\} \\
 & \cup \{(m, P|Q') \quad | \quad (m, Q') \in steps\ \Gamma\ Q\} \\
 & \cup \{(\tau, P'|Q') \quad | \quad \exists \alpha : \ (\alpha, P') \in steps\ \Gamma\ P \ \wedge \ (\overline{\alpha}, Q') \in steps\ \Gamma\ Q\} \\
steps\ \Gamma\ (P \setminus H) & = \{(m, P' \setminus H) \quad | \quad (m, P') \in steps\ \Gamma\ P \ \wedge \ m \notin H\}
\end{array}
$$

Abbildung 19.5: Definierenden Gleichungen von $steps$

```
(* mark -> (mark * ccs) list -> ccs list *)
fun successors act sl = ...

(* mark -> mark *)
fun complement (In a)  = ...
  | complement (Out a) = ...
  | complement Tau     = ...


(* env -> ccs -> (mark * ccs) list  *)
fun steps env Stop          = []
  | steps env (Var X)       = steps env (env X)
  | steps env (Pre (u,P))   = ...
  | steps env (Res (G, set)) = ...
  | steps env (Chc (P,Q))   = (steps env P) @ (steps env Q)
  | steps env (Par (P,Q))   = (map (fn (a,G) => (a,Par(G,Q))) (steps env P) )@
                              (map (fn (a,G) => (a,Par(P,G))) (steps env Q))@
                              (foldl (fn ((a,P'), l1) =>
                                         ((map (fn (Q')=> (Tau,Par(P',Q')))
                                               (successors (complement a) (steps env Q)))
                                          handle TauComplement => [])
                                          @l1
                                     )
                                     nil (steps env P)
                              )
```

Abbildung 19.6: Implementierung von steps

Die Implementierung verwendet – ganz wie wir es in Kapitel 6.4 gelernt haben, strukturelle Rekursion um die Adjazenzlisten zu bilden, welche die Transitionen enthalten, die sich jeweils mit den Inferenzregeln aus Abb. 19.3 beweisen lassen.

Bis auf die Regel für Parallelität sind alle Fälle gut anhand der definierenden Gleichungen zu verstehen, wir werden deshalb im Folgenden diese noch etwas ausführlicher diskutieren: Die definierende Gleichung besteht auf der rechten Seite aus der Vereinigung dreier Mengen. Die ersten beiden Mengen erkennen Sie in der Implementierung in den ersten beiden Zeilen wieder; sie geben alle Anwendungen der Inferenzregeln par_l und par_r wieder. Die Umsetzung der letzten Menge, und damit der Regel sync ist ein wenig komplizierter.

Die verwendete Version ist nicht sehr effektiv, setzt die Inferenzregel aber relativ direkt um. Machen wir uns zunächts klar, was zu tun ist. Um die Menge für die Regelanwendugen von sync algorithmisch zu bestimmen, müssen wir für einen Term $P|Q$ zunächst alle Paare von Elementen aus $steps\ \Gamma\ P$ und $steps\ \Gamma\ Q$ finden, die komplementäre Aktionen an erster Stelle stehen haben. Dazu benötigen wir die Funktion $complement : M \rightharpoonup M$, die zu einer Aktion $a$ ihr Komplement liefert, falls dieses existiert (zu welchen $m \in M$ gibt es kein Komplement?). Um sich das folgende leichter vorstellen zu können, werden wir im folgenden ein Element aus $steps\ \Gamma\ P$ gedanklich fixieren, und den Teilalgorithmus für dieses fixierte Element beschreiben. Der gesamte Algorithmus ergibt sich dann, indem wir das hier beschriebene für alle Elemente aus $steps\ \Gamma\ P$ durchführen. Dieses "für alle" findet durch die Verwendung von `foldl` seine Entsprechung in der Implementierung.

Zunächts bestimmen wir die Liste aller Elemente in $steps\ \Gamma\ Q$, deren erste Komponente zu der des fixierten Elements aus $steps\ \Gamma\ P$ komplementär ist. In der Implementierung ist dies realisiert durch die Prozedur `successors:mark ->(mark * ccs) list -> ccs list` (vgl. Aufgabe 19.14). Dann bauen wir aus der zweiten Komponente, d.h. dem Term $P'$, des fixierten Elements und jedem Term $Q'$ aus der zuvor bestimmten Menge jeweils ein neues Objekt $(\tau, P'|Q')$. Beachtem Sie dazu die Verwendung von `map`. Die so neu entstandene Liste ist die Liste, welche die durch die Inferenzregel sync beweisbaren Transitionen in Form einer Adjazenzliste sammelt.

**Aufgabe 19.12.** Vervollständigen Sie die Prozedur `complement`. Hinweis: Betrachten Sie auch die Verwendung der Prozedur, um sie korrekt zu implementieren.

**Aufgabe 19.13.** Implementieren Sie die Prozedur einer Funktion $mapPartial : (A \rightharpoonup B) \to \mathcal{L}(A) \to \mathcal{L}(B))$, die zu einer *partiellen* Funktion $f : A \rightharpoonup B$ und einer Liste $xs \in \mathcal{L}(A)$ eine Liste $ys \in \mathcal{L}(B)$ liefert, so dass ein Element $x' \in B$ genau dann in $ys$ ist, wenn es in $xs$ ein Element $x \in A$ gibt, so dass $f(x) = x'$ gilt. Hinweis: Eine partielle Funktion können Sie durch eine Prozedur realisieren, die eine Ausnahme genau dann wirft, wenn die Funktion für den entsprechenden Parameter undefiniert ist.

**Aufgabe 19.14.** Vervollständigen Sie die Prozedur `successors`, die zu einer Markierung $m$ und einer Liste $xs$ von Paaren aus $mark * ccs$ eine maximale Liste von Termen $P \in ccs$ findet,

so dass $(m, P)$ in $xs$ enthalten ist. Hinweis: Verwenden Sie die Prozedur `mapPartial` aus Aufgabe 19.13.

**Aufgabe 19.15.** Vervollständigen Sie die fehlenden Teile der Implementierung von `steps`, also die Behandlung von `Pre` und `Res`.

Wir haben damit die Implementierung von $steps$ abgeschlossen. Damit ist allerdings die Frage nach der Semantik eines Termes $P \in CCS$ noch nicht beantwortet. Alles was uns $steps$ für ein bestimmtes $P$ und $\Gamma$ liefert, sind die initialen Transitionen, die von $P$ ausgehen, als Adjazenzliste. Um hieraus den von $P$ erreichbaren Graphen $Reach(\llbracket\ P\ \rrbracket_\Gamma)$ zu erhalten, müssen wir noch einiges über Graphalgorithmen lernen. Glücklicherweise haben wir mit Kapitel 16 alle nötigen Informationen zur Hand, um dies im *Selbststudium* erfolgreich zu erledigen.

Allerdings bietet sich noch eine direktere Antwort auf die Frage an, was denn die Semantik eines Termes $P$ in einer bestimmten Umgebung $\Gamma$ ist:

```
fun sem Gamma P = (steps Gamma, P)
```

Hier ist der zu $P$ gehörende Graph durch die mit $\Gamma$ instantiierte kaskadierte Funktion $steps$ gegeben. Diese liefert für alle $CCS$-Terme die Nachfolgerfunktion, und ist deshalb eine adäquate Darstellung der Semantik von $P$, also des Prozesses $(CCS, M, \rightarrow_\Gamma, P)$, da die Menge der Knoten $CCS$ und die Menge der Markierungen $M$ implizit gegeben sind.

**Aufgabe 19.16.** Die Prozedur `steps` terminiert nicht für alle $CCS$-Terme, und es gibt dafür drei verschiedene Gründe. Um diese Gründe kennenzulernen, untersuchen Sie die Adjazenzmenge von $X$ für die folgenden drei Umgebungen: $\Gamma = \{(X, X)\}$, $\Gamma = \{(X, a!.0 + X)\}, \Gamma = \{(X, a!.0 \mid X)\}$. Die Gründe für sind verschieden, aber die gemeinsame Ursache ist, dass in jedem Fall die strukturelle Rekursion direkt von $X$ auf $X$ zurückführt. Dies können Sie vermeiden, indem Sie festlegen, daß jede Rekursionsvariable erst durch mindestens eine Transition erreicht wird. Syntaktisch bedeutet dies, daß Sie in der abstrakten Grammatik statt mit $X$ mit $m.X$ arbeiten. Damit terminiert die Prozedur `steps` für alle Terme der Sprache. Sind Sie furchtlos genug, um dies zu beweisen? Was ist eine natürliche Terminierungsfunktion?

## 19.5 Semantische Äquivalenz

Wir greifen nun die bereits in Kapitel 18.2 andiskutierte Frage wieder auf, wann zwei Prozesse als *gleich* anzusehen sind. Da wir mit $CCS$ nun eine Sprache für (nebenläufige) Prozesse haben, geht es ganz analog zu der in Kapitel 2.9 für SML und in Kapitel 9.8 für abstrakte Prozeduren geführten Diskussion um die *semantische Äquivalenz*: Wann haben zwei Terme von $CCS$ dieselbe Bedeutung? Oder: Wann sind die Prozesse $P$ und $Q$ semantisch äquivalent? Auf diese Frage gibt es prinzipiell verschiedene Antworten, zum Beispiel:

– Alle Prozesse in $CCS$ sind gleich.

– Zwei Prozesse $P, Q \in CCS$ sind genau dann äquivalent, wenn sie spuräquivalent sind.

– Zwei Prozesse $P, Q \in CCS$ sind genau dann äquivalent, wenn sie isomorph sind.

– Zwei Prozesse $P, Q \in CCS$ sind genau dann äquivalent, wenn sie identisch ist.

Auch wenn nicht all dieser Optionen sinnvoll sind, wollen wir zunächst unser Gedächtnis etwas schärfen: Wir können all diese Relationen auf $CCS$ formal fassen. Die erste entspricht der Relation $CCS \times CCS$, die zweite und dritte haben wir als $\sim_{sp}$ und $\cong$ in Definition 18.1 und Definition 18.2 kennnengelernt. Die letzte ist die Identitätsrelation auf $CCS$, $Id(CCS)$. All diese sind Äquivalenzrelationen auf $CCS$, und es gilt:

$$Id(CCS) \ \subset \cong \subset \sim_{sp} \subset \ CCS \times CCS$$

Die beiden Extreme dieser Inklusions-'Kette' sind offensichtlich nicht sinnvoll: $CCS \times CCS$ ignoriert die Bedeutung der verschiedenen Terme vollständig. $Id(CCS)$ dagegen unterscheidet beispielsweise $0 + 0$ und $0$, was nicht sinnvoll erscheint.

**Aufgabe 19.17.** Beweisen Sie auf $CCS$ die Inklusionen $Id(CCS) \ \subset \cong \subset \sim_{sp} \subset \ CCS \times CCS$.

### 19.5.1 Verklemmung

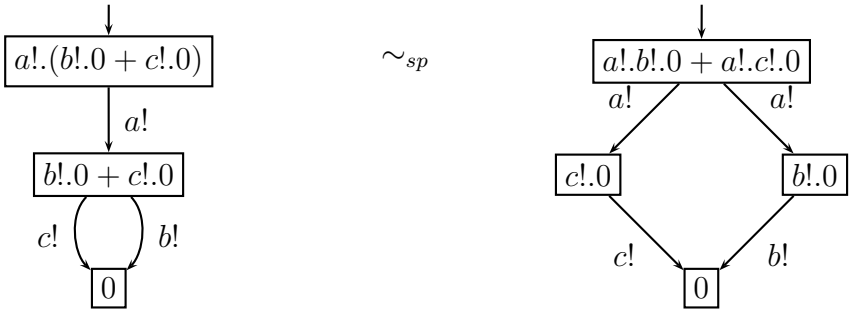
In den Aufgaben 19.6 und 19.7 haben Sie eines der Phänomene kennengelernt, die das Arbeiten mit nebenläufigen und parallelen Systemen schwierig machen. Das Phänomen heißt *Verklemmung* (engl. *Deadlock*), und bedeutet, daß die darin involvierten Prozesse sich hoffnungslos ineinander verhakt haben. Das Problem hat viele Facetten, und tritt typischerweise auf, wenn verschiedene Prozesse mehrere *Ressourcen* benötigt. Unter Ressourcen versteht man typischerweise recht konkrete Dinge, wie Drucker, Lesegeräte, Speicherbereiche, oder Systembusse, die nur begrenzt verfügbar sind.

Für uns ist hier das prinzipielle Phänomen interessant, welches wir wie folgt charakterisieren: Eine Term $P \in CCS$ ist verklemmt, wenn er keine Nachfolger besitzt, obwohl er Teilterme der Form $m.Q$ enthält. So ist beispielsweise $(a!.0 \mid 0) \setminus \{a!, a?\}$ verklemmt. $(0 + 0 \mid 0)$ diese jedoch nicht. Bis auf weiteres brauchen wir keine spezifischen rekursiven Gleichungen, wir nehmen daher, soweit nicht explizit anders dargestellt ein beliebiges $\Gamma$ an.

**Aufgabe 19.18.** Geben Sie eine oder mehrere Gleichungen Ihrer Wahl für *SOUND'* an, so daß sich der Prozess $(BUTTON \mid IDLE \mid SOUND') \setminus \{$ *on!,on?,off!,off?,ok!,ok?* $\}$ verklemmen *muß*.

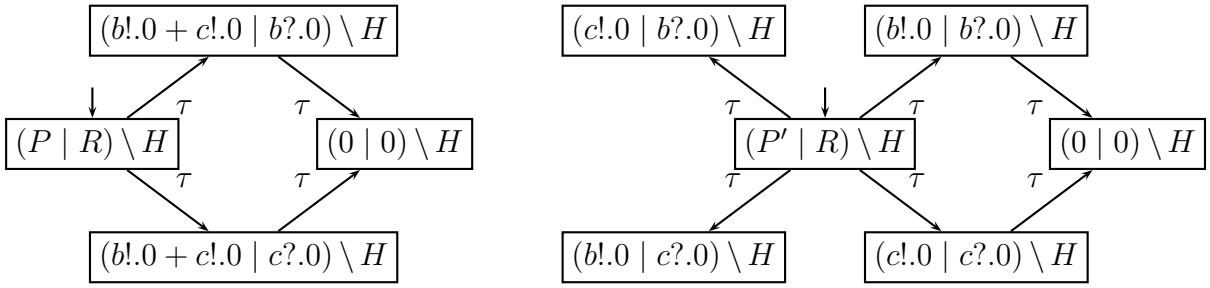
### 19.5.2 Pfui, Spuräquivalenz!

Die in Definition 18.1 eingeführte Spuräquivalenz basiert auf der Idee, dass zwei Prozesse dann gleich sind, wenn diese dieselben Spuren haben. So sind beispielsweise die folgenden Prozessen spuräquivalent: $P = a!.(b!.0 + c!.0)$ und $P' = a!.b!.0 + a!.c!.0$:




Da die Aktionen für das Verhalten und die Synchronisationsmöglichkeiten nebenläufiger Systeme zentral sind, sind auch Abfolgen von Aktionen, also Spuren, für uns wesentlich. Aus den obigen Beispielen wird beispielsweise deutlich, dass die Verklemmungsgefahr von den möglichen Spuren eines Prozesses abhängt: Hätten in Aufgabe 19.6 die Prozesse $User1$ und $User2$ identische Spuren gehabt, so wäre es nicht zu einer Verklemmung gekommen.

Allerdings gilt dies leider nicht generell: Betrachten Sie dazu $(P \mid R) \setminus H$ und $(P' \mid R) \setminus H$, wobei $R = a?.b?.0 + a?.c?.0$ und $H = \{a!, b!, c!, a?, b?, c?\}$. Die durch die Semantk bestimmten Prozesse sehen wie folgt aus:




Zunächst stellen wir fest, dass aus der Tatsache, daß $P$ und $P'$ spuräquivalent sind, folgt, dass auch $(P \mid R) \setminus H$ und $(P' \mid R) \setminus H$ spuräquivalent sind. Dies gilt im übrigen auch allgemein, also für beliebige $P$, $P'$, $R$ und $H$, sofern $P \sim_{sp} P'$.

Allerdings gibt es einen wesentlich Unterschied in den beiden Prozessen: Der rechte kann sich verklemmen, also einen verklemmten Zustand erreichen, der linke nicht. Da Verklemmungen eine sehr unschöne Angelegenheit sind, bereitet uns dies einige Kopfschmerzen. Wir halten zunächst fest:

– Auch wenn zwei Prozesse spuräquivalent sind, ist es nichtsdestotrotz möglich, dass einer von beiden sich verklemmen kann, der andere jedoch nicht.

– Auch wenn zwei Prozesse spuräquivalent sind, und beide sich nicht verklemmen können, ist es nichtsdestotrotz möglich, dass einer von beiden in Kooperation mit einem dritten Prozess zu einer Verklemmung führen kann, der andere jedoch nicht.

Die Spuren eines Prozesses sagen uns also nicht genug über das Verhalten, welches ein Prozess im Zusammenspiel mit anderen Prozessen entfalten kann.

**Aufgabe 19.19.** Ersetzen Sie in der definierenden Gleichung des *Controller*s, die Gleichung für *ON* durch *ON = ok!.off?. IDLE +ok!.brake?. SUSPEND)*. Sind die beiden Prozesse *IDLE* (vor und nach dem Ersetzen) spuräquivalent? Und wie steht es mit

$$(\mathit{BUTTON} \mid \mathit{IDLE} \mid \mathit{SOUND}) \setminus \{\mathit{on!},\mathit{on?},\mathit{off!},\mathit{off?},\mathit{ok!},\mathit{ok?}\}$$

Können Sie in einem der beiden Varianten dieses Systemes eine Verklemmung beobachten?

### 19.5.3 Pfui, Isomorphie!

Betrachten wir noch einmal die Aufgabe 18.11. Darin haben wir gesehen, dass

$$a.(P+Q) \cong a.(P+Q) + a.(P+Q)$$

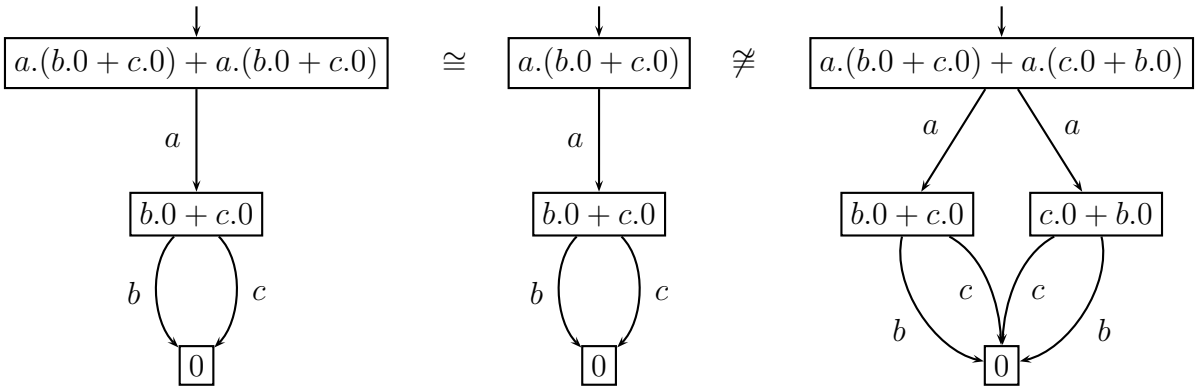
gilt, aber andererseits

$$a.(P+Q) \cong a.(P+Q) + a.(Q+P)$$

*nicht* gilt, und das obwohl

$$P+Q \cong Q+P$$

für alle $P, Q$ gilt. Für $P = a.0$ und $Q = b.0$ ist dieses Phänomen unten skizziert.




Dieses Phänomen ist nicht nur verwirrend, sondern sowohl mathematisch als auch praktisch bedenklich. Von einem vernünftigen Begriff von Gleichheit erwarten wir etwas anderes.

**Definition 19.1.** *Sei $\sim$ eine Äquivalenzrelation auf $CCS$. Wir sagen, dass $\sim$* mit den Operatoren von $CCS$ verträglich *ist, wenn gilt:*

– $\forall P, P' \in CCS \quad \forall m \in M : P \sim P' \implies m.P \sim m.P'$.
– $\forall P, P', R \in CCS \quad \forall m \in M : P \sim P' \implies R + P \sim R + P'$.
– $\forall P, P', R \in CCS \quad \forall m \in M : P \sim P' \implies P + R \sim P' + R$.
– $\forall P, P', R \in CCS \quad \forall m \in M : P \sim P' \implies R \mid P \sim R \mid P'$.

– $\forall P, P', R \in CCS \quad \forall m \in M : P \sim P' \implies P \mid R \sim P' \mid R$.

– $\forall P, P' \in CCS \quad \forall H \subseteq M : P \sim P' \implies P \setminus H \sim P' \setminus H$.

Eine Äquivalenzrelation, die sich im obigen Sinne mit den Operatoren[4] einer Sprache verträgt, wird gemeinhin als Kongruenzrelation, oder auch *Kongruenz* bezeichnet. Kurz gesagt, können Sie in in einem beliebigen Term der Sprache an beliebiger Stelle ein $P$ durch ein äquivalentes $P'$ ersetzen, falls $\sim$ mit den Operatoren der Sprache verträglich ist. Machen Sie sich klar, daß dies eine äußerst natürlich Forderung an eine Äquivalenzrelation ist.

> *Beispiel:* Es ist recht nutzlos, wenn wir für die ganzen Zahlen nicht sicherstellen können, daß aus $(3+7) \sim (10)$ folgt, daß dann auch $13 + (3+7) \sim 13 + (10)$. Dies ist zum Beispiel für die folgendermassen auf $\mathbb{Z}$ definierte Äquivalenzrelation:
>
> $$\{(x, y) \in \mathbb{Z}^2 \mid |x| = |y|\}$$
>
> *nicht* gesichert. Es gilt zwar im konkreten Fall, aber nicht allgemein, denn aus $(3+7) \sim (-10)$ folgt $13+(3+7) \sim 13+(-10)$ *nicht*. Genauso ist es für SML recht nutzlos, wenn zwei Prozeduren zwar äquivalent sind, dieses jedoch, wenn man Sie in einer anderen Prozedur verwendet, zu verschiedenen Endergebnissen führen können.

**Aufgabe 19.20.** Zeigen Sie, daß obige Relation $\sim$ eine Äquivalenzrelation auf $\mathbb{Z}$ ist.

Augenscheinlich ist Isomorphie nicht mit den Operator $+$ von $CCS$ verträglich, ist also *keine* Kongruenz. Spuräquivalenz *ist* hingegen eine Kongruenz auf $CCS$. Die semantische Äquivalenz aus Kapitel 9 ist eine Kongruenz auf SML bezüglich der *idealisierten* Semantik, aber streng genommen nicht bezüglich der implementierten Semantik, siehe Kapitel 2.9.

**Aufgabe 19.21.** Ist $Id(CCS)$ eine Kongruenz auf $CCS$? Und wie steht es mit $CCS \times CCS$?

**Aufgabe 19.22.** Warum erleichtert Ihnen die Information, daß $\sim_{sp}$ eine Kongruenz ist, die Antwort auf Teile der Aufgabe 19.19?

### 19.5.4 Bisimulation

Nachdem wir die Nachteile von Spuräquivalenz und von Isomorphie kennengelernt und verstanden haben, ist es nun an der Zeit, *die* kanonische Äquivalenz auf $CCS$ herauszuarbeiten.

**Definition 19.2.** *Eine binäre Relation $R$ auf $CCS$ ist eine Bisimulation, wenn $(P, Q) \in R$ impliziert, dass $\forall m \in M$ :*

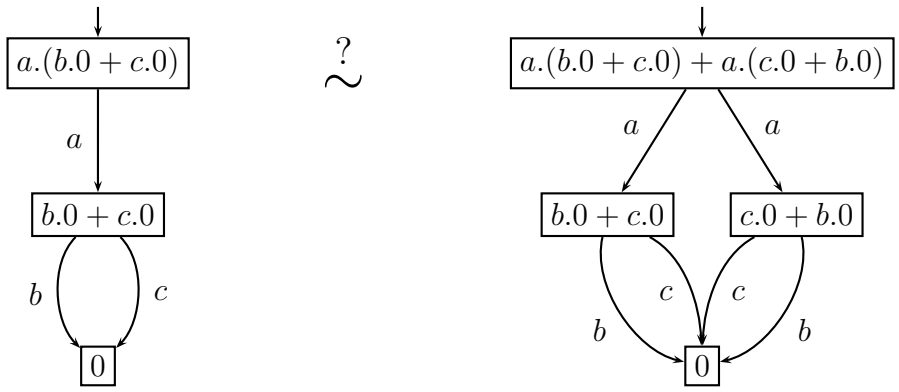
– $P \xrightarrow{m} P' \implies \exists\, Q' \in CCS : Q \xrightarrow{m} Q' \wedge (P', Q') \in R$, *und*

[4] Der Vollständigkeit halber: Wir reden hier von den Operatoren einer beliebigen algebraischen Struktur, $CCS$ ist nur ein Beispiel.

– $Q \stackrel{m}{\to} Q' \implies \exists\, P' \in CCS :\ P \stackrel{m}{\to} P' \ \wedge\ (P', Q') \in R$.

*Zwei Prozesse, $P$ und $Q$, sind bisimilar, geschrieben $P \sim Q$, wenn es eine Bisimulation $R$ gibt, mit $(P, Q) \in R$.*

Die obige Definition besagt, dass zwei Knoten bisimilar sind, wenn sie mit jeweils identischen Kantenmarkierungen jeweils Knoten erreichen können, die jeweils wieder paarweise bisimilar sind. Wir betrachten folgendes Beispiel, und wollen überprüfen, ob $a.(b.0 + c.0) \sim a.(b.0 + c.0) + a.(c.0 + b.0)$ gilt.




Aufgrund der Definition 19.2 gilt dies, sofern wir eine Relation $R$ angeben können, welche die Bedingungen der Definition erfüllt. Wir versuchen es mit der folgenden Relation:

$$R = \{(0,0), (b.0+c.0, b.0+c.0), (b.0+c.0, c.0+b.0), (a.(b.0+c.0), a.(b.0+c.0)+a.(c.0+b.0))\}$$

und stellen fest, daß in der Tat, alle geforderten Bedingungen erfüllt sind.

**Aufgabe 19.23.** Zeigen Sie

– $0 \ \sim\ 0 + 0$,

– $a.0 \ \sim\ a.0 + a.0$,

– $a.0 \ \sim\ a.(0 + 0) + a.0$.

**Aufgabe 19.24.** Zeigen Sie, dass $\sim$ eine Äquivalenzrelation ist. Zeigen Sie, dass $\sim$ selbst eine Bisimulation ist!

**Aufgabe 19.25.** Zeigen Sie, dass $a!.(b!.0 + c!.0) + a!.(b!.0 + c!.0) \ \sim\ a!.(b!.0 + c!.0) + a!.(c!.0 + b!.0)$. Gilt $a!.\tau.\tau.b!.0 \ \sim\ a!.\tau.b!.0$?

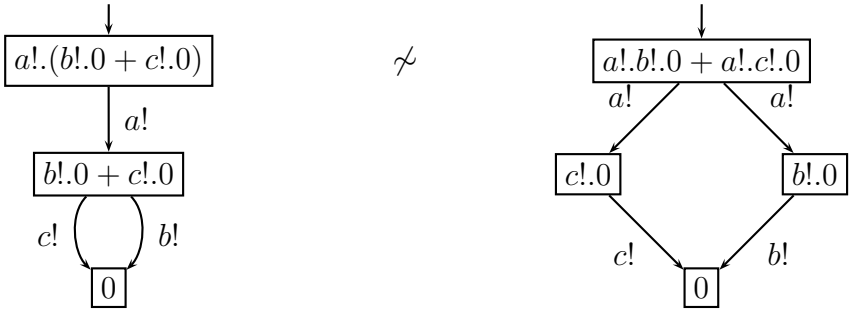
**Proposition 19.1.** *Für $P, Q \in CCS$ gilt*

1. *Aus $P \ \sim\ Q$ folgt $P \ \sim_{sp}\ Q$.*

*2. Aus $P \cong Q$ folgt $P \sim Q$.*

**Aufgabe 19.26.** Beweisen Sie Proposition 19.1: Zeigen Sie, dass zwei bisimilare Prozesse stets spuräquivalent sind. Zeigen Sie, dass zwei isomorphe Prozesse stets bisimilar sind.

Wir haben also mit der Bisimilarität eine Äquivalenzrelation, die – bezüglich der Inklusionsordnung – zwischen Spuräquivalenz und Isomorphie angesiedelt ist. Die Inklusionen sind echt. Für den Unterschied zwischen $\sim$ und $\cong$ haben wir bereits ein Beispiel gesehen. Der Unterschied zwischen $\sim$ und $\sim_{sp}$ manifestiert sich in folgendem Beispiel.




Wenn Sie versuchen, für das Paar $(a!.(b!.0 + c!.0), a!.(b!.0 + c!.0))$ eine Bisimulation $R$ zu konstruieren, werden Sie schnell feststellen, daß es keine Relation gibt, die die Bedingungen aus Definition 19.2 erfüllt. Das Problem ist der Zustand $b!.0 + c!.0$, für den Sie kein passendes Paar bilden können.

Anhand dieses Beispieles haben wir in Abschnitt 19.5.2 das Verklemmungsverhalten eines Prozesses diskutiert, und dabei festgestellt, daß es es trotz $P \sim_{sp} P'$ vorkommen kann, dass z.B. $P'$ in Kooperation mit einem dritten Prozess zu einer Verklemmung führt, $P$ jedoch nicht. Dies gilt für Bisimilarität nicht: Die Äquivalenzrelation $\sim$ hat – im Unterschied zu $\sim_{sp}$ – die wünschenswerte Eigenschaft, dass das Verklemmungsverhalten bisimilarer Prozesse identisch ist. Ausserdem ist $\sim$ – im Unterschied zu $\cong$ – mit den Operatoren von $CCS$ verträglich. Diese beiden Eigenschaften, sowie die Tatsache, daß bisimilare Prozesse gleiche Spuren haben (also $\sim \subset \sim_{sp}$) begründen, warum wir $\sim$ als semantische Äquivalenz für $CCS$ verwenden: Wir sagen also, daß $P, Q \in CCS$ dann gleich sind, wenn $P \sim Q$ gilt[5].

**Aufgabe 19.27.** Betrachten Sie die Prozesse $X = a!.a!.a!.X$ , $X = a!.a!.X$ und $X = a!.(a!.(X + 0) + a!.X)$ , $X = a!.(a!.0 + a!.X)$ . Welche dieser Prozesse sind gleich?

**Aufgabe 19.28.** Beweisen Sie, dass $P + Q$ und $Q + P$ für alle $P, Q \in CCS$ gleich sind. Wie steht es mit $P + P$ und $P$?

**Aufgabe 19.29.** Beweisen Sie, dass $\sim$ verträglich mit dem Auswahloperator ist, also, daß $P \sim Q$ auch $P + R \sim Q + R$ impliziert.

[5] Dies erspart uns insbesondere die weitere Benutzung des etwas sperrigen Wortes *Bisimilarität* für *Gleichheit*.

## Bemerkungen

In diesem Kapitel haben wir die Sprache $CCS$ eingeführt. Wir fassen noch einmal zusammen.

– $0$ ist der Prozess, der keinerlei Verhalten zeigt.

– Der Prozess $m.P$ führt zunächst die Aktion $m$ aus, und verhält sich danach wie der Prozeß $P$.

– $P_1 + P_2$ stellt einen Prozess dar, der sich entweder wie der Prozeß $P_1$ oder der Prozeß $P_2$ verhält. Die Auswahl zwischen den beiden Prozessen ist nicht-determininistisch.

– Der Prozess $X$ verhält sich genau wie $P$, sofern $X = P$ eine der definierenden Gleichungen von $\Gamma$ ist.

– $P_1 \mid P_2$ stellt die parallele Ausführung der Prozesse $P_1$ und $P_2$ dar. Alle Aktionen von $P_1$ und $P_2$ sind darin unabhängig voneinander möglich. Außerdem können Paare von Namen und Co-Namen synchronisiert stattfinden.

– $P \setminus H$ restringiert das Verhalten von $P$. Nur Aktionen $m \notin H$ sind für diesen Prozess möglich.

Wir haben ausserdem gesehen, wie diese Sprache verwendet werden kann, um das Verhalten nebenläufiger Systeme zu untersuchen und zu verstehen. Die Sprache $CCS$ ist allerdings nicht komfortabel genug, um wirklich damit zu arbeiten. Es fehlen insbesondere jede Art von Datentypen. Man kann $CCS$ jedoch zum Beispiel dadurch erweitern, daß man einfache (`bool`, `int`) oder komplizierte (`html`, Funktionen höherer Ordnung) Datentypen mit entsprechender Semantik erlaubt. Die dann entscheidende Änderung ist, daß die Inferenzregel sync erweitert wird, um die Übertragung von Daten zu erlauben. Sie sieht dann im Prinzip wie folgt aus:

$$\frac{v : t \qquad P \xrightarrow{a!v} P' \qquad Q \xrightarrow{a?x:t} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{v/x\}}$$

Diese Regel ist wie folgt zu lesen: Wenn

– der Wert $v$ den Typ $t$ hat und

– $P$ diesen Wert über die Aktion $a$ senden kann und

– $Q$ bereit ist, mit der Aktion $a$ einen Wert vom Typ $t$ zu empfangen,

dann kann dies passieren, und nach der Synchronisation ist der Bezeichner $x$ an $v$ gebunden. Damit läßt sich im Prinzip jede Art von Datenkommunikation auf $CCS$ abbilden.

Wir haben darüberhinaus gelernt, wie man die Syntax und Semantik der Sprache mit Standard ML realisiert, und was präzise die Semantik einer Sprache ist. Die Frage der semantischen Gleichheit hat uns etwas beschäftigt. Wir haben drei Kriterien aufgestellt, die eine sinnvoller Gleichheitsbegriff auf $CCS$ haben sollte.

1. Er sollte das Verklemmungsverhalten eines Prozesses respektieren.
2. Er sollte die Spuren eines Prozesses respektieren.

3. Er sollte mit den Operatoren der Sprache verträglich sein.

Spuräquivalenz und Isomorphie scheitern bei jeweils einem dieser Kriterien. Wir haben dann den Begriff der *Bisimulation* kennnengelernt, welchen wir nach eingehender Untersuchung als den *den* Gleichheitsbegriff auf $CCS$ postiliert haben.

Für die Operatoren der Sprache gelten aufgrund dieses Gleichheitsbegriffes diverse Gesetze, wie zum Beispiel Kommutativität und Assoziativität von $+$ und $|$. Diese liefern uns eine Rechtfertigung für das Weglassen diverser Klammern, und erlaubt uns Teilterme in größeren Termen zu manipulieren. Dies ist ganz so wie in der klassischen *Algebra*. Die Sprache $CCS$ ist ein Vertreter der sogenannten *Prozessalgebren*, sie stammt von Robin Milner (1980). Andere Prozessalgebren sind *CSP* von Tony Hoare (1985) und *ACP* von Jan Bergstra und Jan-Willem Klop (1985). Der Begriff der Bisimulation stammt von David Park (1981).

Die einseitige Variante der Bisimulation, wird *Simulation* genannt. Dabei fordert man nur eine der beiden Punkte aus Definition 19.2. Simulationsrelationen spielen eine wichtige Rolle, wenn es darum geht, die konkrete Implementierung eines nebenläufigen Systems in Beziehung mit seiner abstrakten Verhaltensbeschreibung (typischerweise ein gewurzelter Graph) zu setzen. Diese Frage wird Ihnen in Kürze bereits begegnen.

**Weiterführende Literatur**


– Robin Milner, *Communication and Concurrency*. Prentice-Hall (1989).

– Luca Aceto, Anna Ingolfsdottir, Kim Guldstrand Larsen, Jiri Srba, *An Introduction to Milner's CCS*. `http://www.cs.auc.dk/~luca/SV/intro2ccs.pdf` (2005).

– Jeff Kramer, Jeff Magee, *Concurrency: State Models & Java Programms*. Wiley (1999).

– Tony Hoare, *Communicating Sequential Processes*. Prentice-Hall, `http://www.usingcsp.com/` (1985).