

Der π -Kalkül als Message-Passing-Modell

Prof. Dr. David Sabel

LFE Theoretische Informatik



Letzte Änderung der Folien: 30. Januar 2020

Der π -Kalkül

Rückblick:

- Der Lambda-Kalkül ist ein Modell **sequentieller** Programmiersprachen
- wobei **Operationen** und **Daten** durch **Funktionen** (Abstraktionen) ausgedrückt wird.
- der **Kontrollfluss** wird durch **Anwendungen** von Funktionen auf Argumente ausgedrückt.

Übersicht

- 1 Der π -Kalkül: Einleitung
- 2 Synchroner π -Kalkül
- 3 Der asynchrone π -Kalkül
- 4 Erweiterungen und Varianten des π -Kalküls
 - Nichtdeterministische Auswahl
 - Polyadischer π -Kalkül
 - Rekursive Definitionen
- 5 Prozess-Gleichheit im π -Kalkül
 - Starke Bisimulation
 - Bisimulation
 - Barbed Kongruenz

Der π -Kalkül (2)

Der π -Kalkül:

- Der π -Kalkül ist ein Modell **nebenläufiger** und **verteilter** Programmiersprachen
- Daten und Operationen werden durch **Prozesse** ausgedrückt
- der Kontrollfluss wird durch **Prozesskommunikation** ausgedrückt.
- D.h. ein **Message-Passing-Modell**: Kommunikation nur über Nachrichten
- Besonderes Merkmal: **Mobile** Prozesse

Der π -Kalkül (3)

- Entwickelt von Robin Milner, Joachim Parrow and David Walker in den 1990er Jahren

Anwendungen auch außerhalb der nebenläufigen Programmierung sogar außerhalb der Informatik, z.B.

- Spi-Kalkül: π -Kalkül zur Beschreibung und Verifizieren von kryptographischen Protokollen
- Microsofts XLANG: Beschreibungssprache für Geschäftsprozesse
- Biochemie: Stochastischer π -Kalkül zur formalen Darstellung biochemischer Prozesse

Parallele Komposition



$P \mid Q$

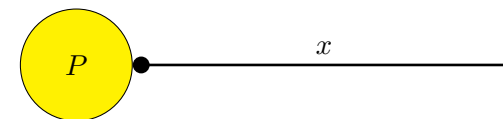
“Prozesse P und Q laufen nebenläufig”

Der π -Kalkül (4)

Verschiedene Varianten

- Synchroner π -Kalkül
- Asynchroner π -Kalkül
- Mit oder ohne Summen: Nichtdeterministische Auswahl
- Mit Rekursion / mit Replikation
- monadisch / polyadisch
- etc.

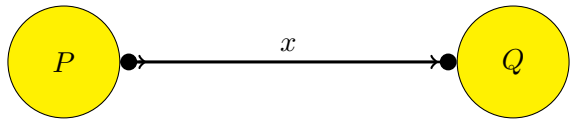
Links



$x.P$ oder $\bar{x}.P$

“ P ist mit Kanal namens x verbunden”

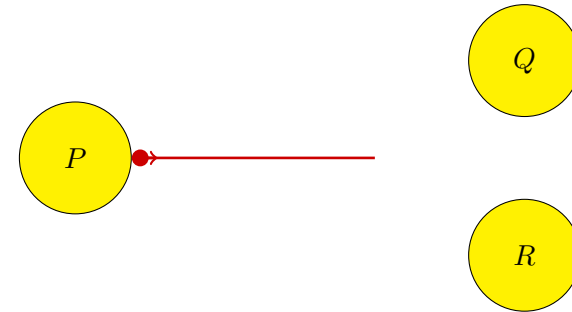
Kommunikation



$$\bar{x}.P \mid x.Q \rightarrow P \mid Q$$

“ P (Sender) und Q (Empfänger) können kommunizieren”
 “ P sendete eine Nachricht an Q ”

Nicht-Determinismus



$$\bar{x}.P \mid x.Q \mid x.R \rightarrow \begin{cases} P \mid Q \mid x.R \\ P \mid x.Q \mid R \end{cases}$$

$$\rightarrow P \mid Q$$

Nachrichten

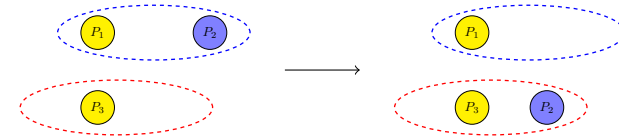


$$\bar{x}m.P \mid x(y).Q \rightarrow P \mid Q[m/y]$$

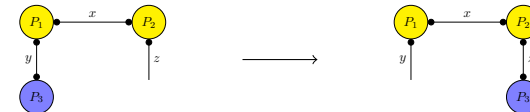
“ P versendet Nachricht m entlang x ”

Mobilität: Ansätze

1. Prozesse verändern ihren Ort im **physikalischen Raum** der Prozesse

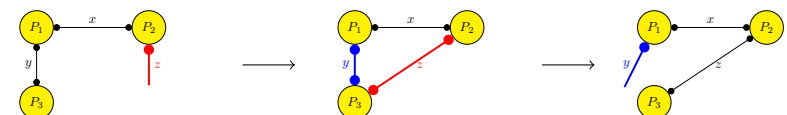


2. Prozesse verändern ihren Ort im **virtuellen Raum** der **verbundenen** Prozesse



3. **Verbindungen** verändern ihren Ort im **virtuellen Raum** der **verbundenen** Prozesse

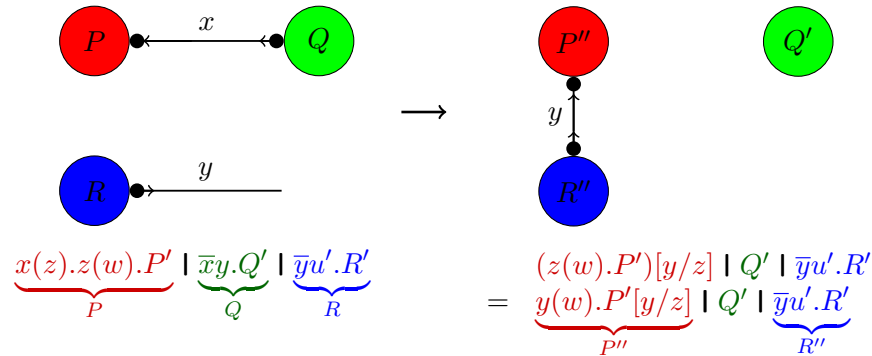
(Ansatz des π -Kalküls, enthält den zweiten Ansatz)



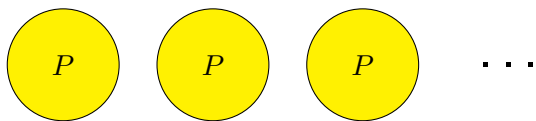
Mobilität (2)

Wie werden Verbindungen bewegt?

⇒ Versende sie als Nachricht!



Replikation



$!P$

“ $!P$ bedeutet $\underbrace{P | P | P | \dots}$ ”
unendlich viele parallele Kopien von P

Private Kommunikation

$\nu x.P$

“Kanal x ist privat für P ”

Beispiel: $\nu x.(x(y).P | \bar{x}z.Q) | \bar{x}z'.R$

- Keine Kommunikation zwischen R und P erlaubt
- äquivalent zu $\nu x'.(x'(y).P | \bar{x}'z.Q) | \bar{x}z'.R$

Synchroner π -Kalkül ohne Summe mit Replikation

Syntax

- \mathcal{N} abzählbar unendliche Menge von **Namen** (ähnlich zu Variablen)
- Syntax für π -Kalkül-**Prozesse** ($x \in \mathcal{N}$)

$P ::= \pi.P$	(Aktion)
$P_1 P_2$	(Parallele Komposition)
$!P$	(Replikation)
$\mathbf{0}$	(Inaktiver Prozess)
$\nu x.P$	(Restriktion)

- Syntax für **Aktionspräfixe** wobei $x, y \in \mathcal{N}$

$\pi ::= x(y)$	Input
$\bar{x}y$	Output

Synchroner π -Kalkül (2)

- $x(y).P$ bedeutet:
Empfange über den **Kanal** namens x einen Namen und binde ihn an y .
Nachdem der Name empfangen wurde: Verhalte dich wie P , wobei für y der empfangene Name eingesetzt wird.
- $\bar{x}y.P$ bedeutet:
Sende über den **Kanal** namens x den Namen y . Danach verhalte dich wie P .
- 0 ist der inaktive Prozess, der nichts tut
- $P_1 \mid P_2$ ist die parallele Komposition von P_1 und P_2 .
- $\nu x.P$ führt P als **Geltungsbereich** für x ein.
- Replikation $!P$ steht für unendlich viele parallele Ausführungen von P , d.h. $\underbrace{P \mid P \mid \dots \mid P}_{\text{unendlich oft}}$.

Alpha-Äquivalenz

Definition (Alpha-Äquivalenz)

Eine **Umbenennung gebundener Namen** eines Prozesses P ist die Ersetzung eines Unterterms

- $x(y).P'$ von P durch $x(z).P'[z/y]$, oder
- $\nu y.P'$ von P durch $\nu z.P'[z/y]$.

Prozesse P_1, P_2 heißen **α -äquivalent**, wenn sie durch Umbenennungen gebundener Namen gleich gemacht werden können.

Wie vorher:

- Alpha-äquivalente Prozesse werden als gleich angesehen.
- Annahme: Distinct Name Convention

Bindungsbereiche

Binder im π -Kalkül sind:

- Restriktion $\nu z.P$ bindet den Namen z mit Geltungsbereich P
- $x(y).P$ bindet y mit Geltungsbereich P
- Beachte: Der Output-Präfix $\bar{x}y$ bindet weder x noch y .

Freie Namen $\text{fn}(P)$

$$\begin{aligned} \text{fn}(x(y).P) &= \{x\} \cup (\text{fn}(P) \setminus \{y\}) \\ \text{fn}(\bar{x}y.P) &= \{x, y\} \cup \text{fn}(P) \\ \text{fn}(P_1 \mid P_2) &= \text{fn}(P_1) \cup \text{fn}(P_2) \\ \text{fn}(0) &= \emptyset \\ \text{fn}(\nu x.P) &= \text{fn}(P) \setminus \{x\} \\ \text{fn}(!P) &= \text{fn}P \end{aligned}$$

Gebundene Namen $\text{bn}(P)$

$$\begin{aligned} \text{bn}(x(y).P) &= \{y\} \cup \text{bn}(P) \\ \text{bn}(\bar{x}y.P) &= \text{bn}(P) \\ \text{bn}(P_1 \mid P_2) &= \text{bn}(P_1) \cup \text{bn}(P_2) \\ \text{bn}(0) &= \emptyset \\ \text{bn}(\nu x.P) &= \{x\} \cup \text{bn}(P) \\ \text{bn}(!P) &= \text{bn}P \end{aligned}$$

Alle Namen eines Prozesses: $\text{n}(P) := \text{fn}(P) \cup \text{bn}(P)$

Prozesskontexte

Prozesskontext D

- Prozess, der an einer Position anstelle eines Prozesses ein Loch $[\cdot]$ hat.
- Grammatik:

$$D ::= [\cdot] \mid \pi.D \mid D \mid P \mid P \mid D \mid !D \mid \nu x.D \quad \text{für } x \in \mathcal{N}$$

Strukturelle Kongruenz

Strukturelle Kongruenz \equiv : Kleinste Kongruenz auf Prozessen, die die folgenden Axiome erfüllt.

$$\begin{aligned}
 P &\equiv Q, \text{ falls } P \text{ und } Q \text{ } \alpha\text{-} \text{äquivalent sind} \\
 P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\
 P_1 \mid P_2 &\equiv P_2 \mid P_1 \\
 P \mid \mathbf{0} &\equiv P \\
 \nu z. \nu w. P &\equiv \nu w. \nu z. P \\
 \nu z. \mathbf{0} &\equiv \mathbf{0} \\
 \nu z. (P_1 \mid P_2) &\equiv P_1 \mid \nu z. P_2, \text{ falls } z \notin \text{fn}(P_1) \\
 !P &\equiv P \mid !P
 \end{aligned}$$

- Umgekehrt ausgedrückt: Kann man P_1 mithilfe der Axiome (angewendet auf Unterprozesse) und α -Umbenennungen in P_2 überführen, dann sind P_1 und P_2 strukturell kongruent (d.h. $P_1 \equiv P_2$)

Operationale Semantik

Reduktionsregeln

$$\begin{aligned}
 (\text{INTERACT}) \quad &x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q \\
 (\text{PAR}) \quad &P \mid Q \rightarrow P' \mid Q, \text{ falls } P \rightarrow P' \\
 (\text{NEW}) \quad &\nu x.P \rightarrow \nu x.P', \text{ falls } P \rightarrow P' \\
 (\text{STRUCTCONGR}) \quad &P \rightarrow P', \text{ falls } Q \rightarrow Q', P \equiv Q \text{ und } P' \equiv Q'
 \end{aligned}$$

Alternative Definition:

- Sei die Menge \mathcal{R} der Reduktionskontexte beschrieben durch $R ::= [\cdot] \mid R \mid P \mid \nu x.R$
- Reduktion \rightarrow :
Wenn $R \in \mathcal{R}$, $P_0 \equiv R[x(y).P \mid \bar{x}v.Q]$, $Q_0 \equiv R[P[v/y] \mid Q]$, dann $P_0 \rightarrow Q_0$.

Strukturelle Kongruenz: Bemerkungen

- Strukturelle Kongruenz ist "eigentlich" dafür gedacht Prozesse als gleich anzusehen, weil sie mehr oder weniger die gleiche Struktur haben.
- Tatsächlich ist bis heute nicht bewiesen, dass \equiv überhaupt **entscheidbar** ist:
Entscheidungsproblem: Gegeben P_1, P_2 . Gilt $P_1 \equiv P_2$?
- Bekannt: \equiv ist EXPSPACE-schwer! (siehe Literatur)
- EXPSPACE-Schwere kommt bereits durch $!$ und \mid
- Zusammen mit ν wird es anscheinend noch schwieriger
- Es gibt entscheidbare Varianten.

Beispiele

$$\text{Betrachte: } P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$$

- P_1 und P_2 kommunizieren über den Kanal x , wobei P_2 an P_1 den Namen z verschickt:

$$P \rightarrow \underbrace{\bar{z}z.\mathbf{0}}_{P'_1} \mid \underbrace{\mathbf{0}}_{P'_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

- P'_1 und P_3 kommunizieren über den Kanal z , wobei P'_1 den Namen z verschickt:

$$P' \rightarrow \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv P''$$

- Ergebnis ist strukturell kongruent zu $\mathbf{0}$ und kann nicht weiter reduziert werden.

Beispiele (2)

- Die Reduktion ist **nicht-deterministisch**
- Betrachte:

$$P \equiv x(y).0 \mid \bar{x}v.0 \mid x(z).\bar{z}w.0$$

$$\swarrow \quad \searrow$$

$$0 \mid 0 \mid x(z).\bar{z}w.0 \qquad x(y).0 \mid 0 \mid \bar{v}w.0$$

Extrusion

- Lokale Namen, können ihren Bindungsbereich durch Kommunikation verlassen.
- Betrachte

$$P \equiv x(y).P_1 \mid \nu z.\bar{x}z.P_2$$

wobei z nicht in P_1 vorkommt.

- Bei Reduktion mit (INTERACT):

$$P \equiv \nu z.(x(y).P_1 \mid \bar{x}z.P_2) \rightarrow \nu z.(P_1[z/y] \mid P_2)$$

- Der scheinbar nur für P_2 bekannte Namen z wird auch für P_1 bekannt
- Dieses Phänomen wird als **Extrusion** bezeichnet.

Beispiele (3)

- ν -Binder verhindern Kommunikationsmöglichkeiten und ermöglichen lokale Kommunikation, die durch den Eingriff von außen geschützt ist.
- Betrachte:

$$P \equiv \nu x.(x(y).0 \mid \bar{x}v.0) \mid x(z).\bar{z}w.0$$

$$\rightarrow \nu x.(0 \mid 0) \mid x(z).\bar{z}w.0$$

- nur eine Reduktion möglich!
- Nach α -Umbenennung von P :

$$P \equiv \nu x'.(x'(y).0 \mid \bar{x}'v.0) \mid x(z).\bar{z}w.0$$

Unendliche Reduktionsfolgen

- Unendlich lange Reduktionsfolgen sind möglich
- Beispiel:

$$P = !(\bar{x}v.0) \mid !(x(z).0)$$

- Anfang der Reduktion

$$\begin{aligned} P &= !(\bar{x}v.0) \mid !(x(z).0) \\ &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid !(x(z).0) \\ &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid !(x(z).0) \\ &\rightarrow !(\bar{x}v.0) \mid 0 \mid 0 \mid !(x(z).0) \\ &\equiv !(\bar{x}v.0) \mid !(x(z).0) \\ &\equiv P \end{aligned}$$

- Bewiesen von Robin Milner, 1992
- Kodiere den call-by-name Lambda Kalkül in den π -Kalkül
- Übersetzung $\llbracket \cdot \rrbracket$
- Für Lambda-Ausdruck s : $\llbracket s \rrbracket u$ ist Prozess
- wobei u ein Name ist
- über u "erhält" s seine Argumente

Kodierung $\llbracket \cdot \rrbracket$ (2)

Übersetzung von Anwendungen

- Bei $(s t)$ wird quasi eine Bindung $x = t$ für das Argument als eigener Prozess angelegt.
- Der Prozess für $x = t$ fordert jedes Mal einen Kanalnamen an, über welchen er seine Argumente erhält
- Abkürzung:

$$\langle x = t \rangle := !(x(w). \llbracket t \rrbracket w)$$

- Replikation: Da t eventuell öfter gebraucht wird
- Übersetzung der Anwendung, wobei x ein neuer Name:

$$\llbracket s t \rrbracket u := \nu v. (\llbracket s \rrbracket v \mid \nu x. \bar{v}x. \bar{v}u. \langle x = t \rangle)$$

Übersetzung von Abstraktionen

$$\llbracket \lambda x. s \rrbracket u := u(x). u(v). \llbracket s \rrbracket v$$

- Über u empfängt die Abstraktion zwei Namen:
- Zum einen x ,
- zum anderen den Namen über den s seine Argumente erhält

Übersetzung von Variablen

$$\llbracket x \rrbracket u := \bar{x}u. \mathbf{0}$$

Beispiel: $\llbracket \lambda x. x \rrbracket u = u(x). u(v). \bar{x}v. \mathbf{0}$

Beispiel

$$\begin{aligned} \llbracket (\lambda x. x) t \rrbracket u &= \nu v. (\llbracket (\lambda x. x) \rrbracket v \mid \nu y. \bar{v}y. \bar{v}u. \langle y = t \rangle) \\ &= \nu v. (v(x). v(w). \llbracket x \rrbracket w \mid \nu y. \bar{v}y. \bar{v}u. \langle y = t \rangle) \\ &= \nu v. (v(x). v(w). \bar{x}w. \mathbf{0} \mid \nu y. \bar{v}y. \bar{v}u. \langle y = t \rangle) \\ &\equiv \nu y. \nu v. (v(x). v(w). \bar{x}w. \mathbf{0} \mid \bar{v}y. \bar{v}u. \langle y = t \rangle) \\ &\rightarrow \nu y. \nu v. (v(w). \bar{y}w. \mathbf{0} \mid \bar{v}u. \langle y = t \rangle) \\ &\rightarrow \nu y. \nu v. (\bar{y}u. \mathbf{0} \mid \langle y = t \rangle) \\ &= \nu y. \nu v. (\bar{y}u. \mathbf{0} \mid !(y(w). \llbracket t \rrbracket w)) \\ &\equiv \nu y. \nu v. (\bar{y}u. \mathbf{0} \mid (y(w). \llbracket t \rrbracket w) \mid !(y(w). \llbracket t \rrbracket w)) \\ &\rightarrow \nu y. \nu v. (\mathbf{0} \mid (\llbracket t \rrbracket u) \mid !(y(w). \llbracket t \rrbracket w)) \\ &\equiv \llbracket t \rrbracket u \mid \nu y. (!(y(w). \llbracket t \rrbracket w)) \end{aligned}$$

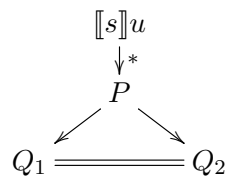
Zusammengefasst:

$$\llbracket (\lambda x.x) t \rrbracket u \xrightarrow{*} \llbracket t \rrbracket u \mid \nu y.(! (y(w). \llbracket t \rrbracket w))$$

- $\nu y.(! (y(w). \llbracket t \rrbracket w))$ kann nicht kommunizieren, daher „Garbage“
- Im call-by-name Lambda Kalkül: $(\lambda x.x) t \rightarrow t$

Lemma

Sei s ein geschlossener Lambda-Ausdruck, dann ist $\llbracket s \rrbracket u$ **deterministisch bzgl. \equiv** , d.h. falls $\llbracket s \rrbracket u \xrightarrow{*} P$ und $P \rightarrow Q_1$ als auch $P \rightarrow Q_2$, dann gilt stets $Q_1 \equiv Q_2$.



Insgesamt folgt:

Theorem

Der synchrone π -Kalkül ist Turing-mächtig.

Proposition

Sei s ein geschlossener Lambda-Ausdruck, dann gilt eine der folgenden Bedingungen:

- 1 $s \downarrow_{name}$ und $\llbracket s \rrbracket u \xrightarrow{*} P$, sodass P irreduzibel ist.
- 2 $s \uparrow_{name}$ und es gibt kein P , so dass $\llbracket s \rrbracket u \xrightarrow{*} P$, wobei P irreduzibel ist.

Informal: Call-by-name Auswertung von Lambda-Ausdrücken kann im π -Kalkül simuliert werden.

- Resultat zeigt nur, dass Lambda-Ausdrücke **ausgeführt** werden können.
- Kein Zusammenhang zwischen **Gleichheitstheorien**
- Milner 1992 zeigt stärkere Aussagen über $\llbracket \cdot \rrbracket$
- Wir haben allerdings (noch) keinen Gleichheitsbegriff für den π -Kalkül definiert
- Es gibt verschiedene Begriffe!
- Milner 1992 gibt zusätzlich eine Kodierung des **call-by-value** Lambda-Kalküls in den π -Kalkül an.

- Bisher: Die Kommunikation findet in einem Schritt statt, d.h. **synchrone** Kommunikation
- Im **asynchronen** π -Kalkül: Fast alles identisch, nur ein Unterschied:
- Nach dem Output-Präfix $\bar{x}y$ darf nur der inaktive Prozess 0 folgen.

Syntax

$$P ::= 0 \mid \bar{x}y.0 \mid x(y).P \mid P_1 \mid P_2 \mid \nu z.P \mid !P$$

Kodierung: Synchroner π -Kalkül in den asynchronen

- Umkehrung klar: Da asynchroner π -Kalkül ein Subkalkül des synchronen ist.
- Sei $P' \mid Q' \equiv (\bar{x}z.P \mid x(y).Q)$ ein Prozess des synchronen π -Kalküls.
- Wir nennen P' den **Sender**, Q' den **Empfänger**
- Ziel: P und Q sind blockiert bis die Kommunikation statt gefunden hat.
- Erste Idee: Übersetze $\bar{x}z.P$ als $\bar{x}z.0 \mid P$
- Funktioniert nicht: P kann weiterreduzieren ohne auf die Kommunikation zu warten!
- Idee: Verwende asynchrone Kommunikation, wobei der Sender auf eine Empfangsbestätigung wartet.

- Der asynchrone Charakter kommt durch die Sichtweise:
- $\bar{x}y.0$ sind **gesendete** Nachrichten, die im Raum „herumschwirren“

$$P \equiv \underbrace{\bar{x}_1y_1.0 \mid \dots \mid \bar{x}_ny_n.0}_{\text{Medium der gesendeten Nachrichten (Puffer)}} \mid \underbrace{Q_1 \mid \dots \mid Q_n}_{\text{„echtes“ Programm}}$$

Kodierung mit Empfangsbestätigung

$$\begin{array}{ll} \text{Sender} & S = \bar{x}z.0 \mid u(v).P \\ \text{Empfänger} & E = x(y).(\bar{u}v.0 \mid Q) \end{array}$$

Auswertung dazu:

$$\begin{aligned} & S \mid E \\ = & (\bar{x}z.0 \mid u(v).P) \mid x(y).(\bar{u}v.0 \mid Q) \\ & \rightarrow u(v).P \mid \bar{u}v.0 \mid Q[z/y] \\ & \rightarrow P \mid Q[z/x] \end{aligned}$$

Kodierung mit Empfangsbestätigung (2)

- Schwachpunkt: Nicht unabhängig vom Kontext:
- Übersetzung von $D[P' \mid Q']$ stellt nicht sicher, dass andere Prozesse aus D über Kanal u kommunizieren.
- Beispiel $D = u(w).\mathbf{0} \mid [\cdot]$
- Dann kann $D[S \mid E]$ wie folgt reduzieren:

$$\begin{aligned} & u(w).\mathbf{0} \mid (\bar{x}z.\mathbf{0} \mid u(v).P) \mid x(y).(\bar{u}v.\mathbf{0} \mid Q) \\ \rightarrow & u(w).\mathbf{0} \mid (u(v).P) \mid \bar{u}v.\mathbf{0} \mid Q[z/y] \\ \rightarrow & (u(v).P) \mid Q[z/y] \end{aligned}$$

- Lösung: zuerst einen **privaten** Kanalnamen zwischen Sender und Empfänger austauschen

Kodierung mit Empfangsbestätigung (4)

Übersetzung $\llbracket \cdot \rrbracket_\pi$ übersetzt synchrone in asynchrone Prozesse

$$\begin{aligned} \llbracket \mathbf{0} \rrbracket_\pi &= \mathbf{0} \\ \llbracket \bar{x}y.P \rrbracket_\pi &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}y.\mathbf{0} \mid \llbracket P \rrbracket_\pi)), \\ &\quad \text{wobei } u, v \notin \text{fn}(P) \\ \llbracket x(y).P \rrbracket_\pi &= x(u).\nu v.(\bar{u}v.\mathbf{0} \mid v(y).\llbracket P \rrbracket_\pi), \text{ wobei } u, v \notin \text{fn}(P) \\ \llbracket P \mid Q \rrbracket_\pi &= \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi \\ \llbracket !P \rrbracket_\pi &= !\llbracket P \rrbracket_\pi \\ \llbracket \nu x.P \rrbracket_\pi &= \nu x.\llbracket P \rrbracket_\pi \end{aligned}$$

Kodierung mit Empfangsbestätigung (3)

- Sender $\nu u.(\bar{x}u.\mathbf{0} \mid u(v)P)$
- Dann kann $u(v)P$ erst dann weiter reduzieren, wenn u über x versendet wurde, da u vorher nach außen **unsichtbar**!
- $u(v)P$ blockiert bis die Kommunikation stattgefunden hat.
- Damit kann man synchronisieren

Beispiel

Auswertung von $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned} & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\ &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid \nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\ &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid \llbracket Q \rrbracket_\pi[z/y])) \\ &\equiv \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi[z/y] \end{aligned}$$

- P kann weiter rechnen bevor z gesendet wurde
- Aber Kommunikation ist sichergestellt, nur Q kennt v'

Notation für geschlossene Prozesse

Definition

Ein **Ausgabe-geschlossener** Prozess des synchronen bzw. asynchronen π -Kalküls ist ein Prozess, so dass für jeden Output-Präfix $\bar{x}y$ gilt: Die Namen x und y sind durch Binder gebunden.

Notation für Antworten

Definition

Ein Prozess P ist eine **Eingabe-Antwort**, wenn gilt $P \equiv \nu x_1 \dots \nu x_n . x(y) . P' \mid Q$

Resultat von Palamidessi

- Synchroner π -Kalkül mit **Summen** ist nicht (sinnvoll) kodierbar in den asynchronen π -Kalkül ohne Summen

Voraussetzungen:

- Übersetzung ist **kompositional** und $\llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket$
- Übersetzung ist wohl-verhaltend bezüglich Umbenennungen: $\llbracket \sigma(P) \rrbracket = \sigma(\llbracket P \rrbracket)$.

Resultat für die Konvergenz:

Theorem

Sei P ein Ausgabe-geschlossener Prozess des synchronen π -Kalküls. Dann gilt:

$$\begin{aligned} &\exists \text{ Eingabe-Antwort } P': P \xrightarrow{*} P' \\ &\text{genau dann, wenn} \\ &\exists \text{ Eingabe-Antwort } P'': \llbracket P \rrbracket_{\pi} \xrightarrow{*} P'' \end{aligned}$$

- Boudol zeigt noch mehr: Adäquatheit bezüglich der kontextuellen Präordnungen

Nichtdeterministische Auswahl

- $P + Q$, verhält sich wie P oder Q
- Oft beschränkt: "guarded choice": Beim Auswählen muss Kommunikation stattfinden, $\pi_1.P + \pi_2.Q$
- Wir: **Synchroner π -Kalkül mit Summen**
- Zusätzlicher Präfix: τ = nicht-beobachtbare Aktion.
- Wir verwenden das Summenzeichen Σ und $+$

$$\pi ::= x(y) \mid \bar{x}y \mid \tau \quad \text{wobei } x, y \in \mathcal{N}$$

$$P ::= \pi.P \mid P_1 \mid P_2 \mid !P \mid \mathbf{0} \mid \nu x.P \mid \sum_i \pi_i.P_i \quad \text{für } x \in \mathcal{N}$$

- Strukturelle Kongruenz, neu: $P + Q \equiv Q + P$, $((P_1 + P_2) + P_3) \equiv (P_1 + (P_2 + P_3))$, $P + P \equiv P$.

- (SILENT) $\tau.P \rightarrow P$
- (SILENTSUM) $\tau.P + M \rightarrow P$
- (INTERACTSUM) $x(y).P + M \mid \bar{x}v.Q + N \rightarrow P[v/y] \mid Q$
- (INTERACT) $x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q$
- (PAR) $P \mid Q \rightarrow P' \mid Q$, falls $P \rightarrow P'$
- (NEW) $\nu x.P \rightarrow \nu x.P'$, falls $P \rightarrow P'$
- (STRUCTCONGR) $P \rightarrow P'$, falls $Q \rightarrow Q'$ und $P \equiv Q$ und $P' \equiv Q'$

Übersetzung: Polyadischer π -Kalkül in monadischen

- Idee: Schicke Elemente aus Tupel (z_1, \dots, z_n) nacheinander
- Übersetze $x(y_1, \dots, y_n).P$ als $x(y_1). \dots x(y_n).P'$
- und übersetze $\bar{x}(z_1, \dots, z_n).Q$ als $\bar{x}z_1. \dots \bar{x}z_n.Q'$

Funktioniert nicht, $P := \bar{x}(z_1, z_2).0 \mid \bar{x}(z_3, z_4).0 \mid x(y_1, y_2).\bar{y}_1y_2.0$
 Möglichkeiten P zu reduzieren:

- $P \rightarrow 0 \mid \bar{x}z_3, z_4.0 \mid \bar{z}_1z_2.0$
- $P \rightarrow \bar{x}(z_1, z_2).0 \mid \bar{z}_3z_4.0$

Übersetzung $Q := \bar{x}z_1.\bar{x}z_2.0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid x(y_1).x(y_2).\bar{y}_1y_2.0$

Reduktionen für Q :

- $Q \xrightarrow{*} 0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid \bar{z}_1z_2.0$
- $Q \xrightarrow{*} \bar{x}z_1.\bar{x}z_2.0 \mid 0 \mid \bar{z}_3z_4.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_1z_3.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_3z_1.0$

- bisher: **monadischer** π -Kalkül: Nur ein Name wird kommuniziert
- **polyadischer** π -Kalkül: Tupel von Namen wird ausgetauscht
- Die Action-Präfixe π im **polyadischen** π -Kalkül:

$$\pi ::= x(\vec{y}) \mid \bar{x}\vec{z} \text{ wobei } x, y_i, z_i \in \mathcal{N}$$

- INTERACT-Regel:

$$(INTERACT) x(\vec{y}).P \mid \bar{x}\vec{z}.Q \rightarrow P[\vec{z}/\vec{y}] \mid Q \text{ falls } |\vec{y}| = |\vec{z}|$$

- $P[\vec{z}/\vec{y}]$ meint hierbei $P[z_1/y_1, \dots, z_n/y_n]$ wenn $|\vec{z}| = n$
- Verboten: Gleicher Kanal mit verschiedenen Tupellängen benutzen, z.B. $x(a, b).P \mid x(a, b, c).Q$
- Formal über Sorten (wie ein Typsystem)

Übersetzung: Polyadischer π -Kalkül in monadischen (2)

- Korrektur: Sender und Empfänger tauschen privaten Namen aus

Korrekte Übersetzung

$$[[x(z_1, \dots, z_n).P]]_{poly} = x(w).w(z_1). \dots w(z_n).[[P]]_{poly}$$

$$[[\bar{x}z_1, \dots, z_n.P]]_{poly} = \nu v.\bar{x}v.\bar{v}z_1. \dots \bar{v}z_n. [[P]]_{poly}$$

$$[[\nu x.P]]_{poly} = \nu x. [[P]]_{poly}$$

$$[[P \mid Q]]_{poly} = [[P]]_{poly} \mid [[Q]]_{poly}$$

$$[[!P]]_{poly} = ![P]_{poly}$$

$$[[0]]_{poly} = 0$$

Rekursive Definitionen

- Anstelle der Replikation: rekursive Prozessdefinitionen
- **Konstantenanwendungen** $A(x_1, \dots, x_n)$ hinzu

$$P := \dots | \mathcal{P} | \dots | A(x_1, \dots, x_n) \text{ wobei } x_i \in \mathcal{N}$$

- Definition für A außerhalb des Programms

$$A(x_1, \dots, x_n) = P \text{ wobei } \text{fn}(P) \subseteq \{x_1, \dots, x_n\}$$

- Neue Reduktionsregel:

$$\text{(CONST)} \quad A(y_1, \dots, y_n) \rightarrow P[y_1/x_1, \dots, y_n/x_n] \\ \text{falls } A(x_1, \dots, x_n) = P \text{ die Definition von } A \text{ ist}$$

Übersetzung: Replikation in Rekursion

- Übersetzung $\llbracket \cdot \rrbracket_{repl}$
- Betrachte $!P$ mit $\{x_1, \dots, x_n\} = \text{fn}(P)$.
- Wähle neue Konstante A_P und übersetze:

$$\llbracket !P \rrbracket_{repl} := A_P(x_1, \dots, x_n)$$

- Alle anderen Konstrukte werden homomorph übersetzt.
- Definition für die Konstante A_P :

$$A_P(x_1, \dots, x_n) = \llbracket P \rrbracket_{repl} | A_P(x_1, \dots, x_n)$$

Übersetzung Rekursion in Replikation, polyadisch

- Seien $\{A_1, \dots, A_n\}$ alle definierten Konstanten, mit Definition $A_i(\vec{x}_i) = P_i$.
- Neue Kanalnamen a_1, \dots, a_n
- Jede Konstantenanwendung $A_i(\vec{y}_i)$ wird durch $\langle A_i(\vec{y}_i) \rangle := \bar{a}_i \vec{y}_i.0$ ersetzt
- Definitionen der A_i durch Prozesse darstellen: $A'_i := !a_i(\vec{x}_i).\langle P_i \rangle$.
- Übersetzung:

$$\llbracket Q \rrbracket_{rek} := \nu a_1 \dots \nu a_n. (\langle Q \rangle | A'_1 | \dots | A'_n)$$

- Achtung: Nicht kompositional, z.B. $\llbracket Q_1 | Q_2 \rrbracket_{rek} \neq \llbracket Q_1 \rrbracket_{rek} | \llbracket Q_2 \rrbracket_{rek}$.

Beispiel

$$Q = !\bar{x}z.0 | x(y).0 | x(w).0$$

Auswertung von Q :

$$Q \equiv !\bar{x}z.0 | \bar{x}z.0 | x(y).0 | x(w).0 \\ \rightarrow !\bar{x}z.0 | 0 | 0 | x(w).0 \equiv !\bar{x}z.0 | \bar{x}z.0 | x(w).0 \\ \rightarrow !\bar{x}z.0 | 0 | 0 \equiv !\bar{x}z.0$$

Übersetzung:

$$\llbracket Q \rrbracket_{repl} = A(x, z) | x(y).0 | x(w).0 \\ \text{wobei } A(x, z) = \bar{x}z | A(x, z)$$

Eine Auswertung von $\llbracket Q \rrbracket_{repl}$:

$$\llbracket Q \rrbracket_{repl} = A(x, z) | x(y).0 | x(w).0 \\ \rightarrow (\bar{x}z | A(x, z)) | x(y).0 | x(w).0 \\ \rightarrow A(x, z) | 0 | x(w).0 \\ \rightarrow (\bar{x}z | A(x, z)) | 0 | x(w).0 \\ \rightarrow A(x, z) | 0 | 0$$

- Im π -Kalkül verschiedene Begriffe, kein einheitlicher Begriff
- Kein **Terminierungsbegriff**
- Oft: Terminierung von Prozessen nicht zentral, da verteilte Systeme oft in Endlosschleifen laufen und somit nicht terminieren.
- Gleichheitsbegriff oft: Prozesse haben gleiche Ein- / Ausgabemöglichkeiten
- Statt Reduktion, **labeled transition system**
- Markierungen stellen gerade die Ein- und Ausgaben des Prozesses dar
- Unterscheidung **interne Kommunikation** und **externe Kommunikation**

Labeled Transition System (2)

Für synchronen π -Kalkül:

$$(IN) \quad x(y).P \xrightarrow{x(z)} P[z/y]$$

$$(OUT) \quad \bar{x}y.P \xrightarrow{\bar{x}y} P$$

$$(OPEN) \quad \nu y.P \xrightarrow{\bar{x}(y)} Q, \text{ falls } P \xrightarrow{\bar{x}y} Q \text{ und } x \neq y.$$

$$(INTERACT) \quad x(y).P \mid \bar{x}v.Q \xrightarrow{\tau} P[v/y] \mid Q$$

$$(PAR) \quad P \mid Q \xrightarrow{\alpha} P' \mid Q, \text{ falls } P \xrightarrow{\alpha} P' \text{ und } n(\alpha) \cap \text{fn}(Q) = \emptyset$$

$$(NEW) \quad \nu x.P \xrightarrow{\alpha} \nu x.P', \text{ falls } P \xrightarrow{\alpha} P' \text{ und } x \notin n(\alpha)$$

$$(STRUCTCONGR) \quad P \xrightarrow{\alpha} P', \text{ falls } Q \xrightarrow{\alpha} Q' \text{ und } P \equiv Q \text{ und } P' \equiv Q'$$

- Summen, τ usw. können hinzugefügt werden

- Sichtweise der ext. Kommunikation: Wie kann ein Prozess kommunizieren, wenn man einen Kommunikationspartner hinzufügen würde?
- Prozess der Form $P_0 \equiv \nu z_1 \dots \nu z_n. x(y).P$ mit $x, y \neq z_i$ kann **eine Eingabe empfangen**
- Prozess der Form $P_0 \equiv \nu z_1 \dots \nu z_n. \bar{x}y.P$ mit $x, y \neq z_i$ kann **eine Ausgabe durchführen**
- Andere Variante: Ausgabe eines gebundenen Namens, d.h. für $P_0 = \nu z_1 \dots \nu z_n. \nu y. \bar{x}y.P$ mit $x, y \neq z_i$: Für die Kommunikation muss $\nu y.$ verschoben werden
- Markierungen: $\alpha := \tau \mid x(y) \mid \bar{x}y \mid \bar{x}(y)$
- $\text{fn}(x(y)) = \{x, y\}$ $\text{bn}(x(y)) = \emptyset$
 $\text{fn}(\bar{x}y) = \{x, y\}$ $\text{bn}(\bar{x}y) = \emptyset$
 $\text{fn}(\bar{x}(y)) = \{x\}$ $\text{bn}(\bar{x}(y)) = \{y\}$
 $\text{fn}(\tau) = \emptyset$ $\text{bn}(\tau) = \emptyset$

Beispiel

$$\begin{aligned} & \nu x.(x(u).\bar{u}u.0 \mid \bar{x}w.z(a).\bar{a}b.0) \\ \xrightarrow{\tau} & \bar{w}w.0 \mid \underline{z(a).\bar{a}b.0} \\ \xrightarrow{z(c)} & \bar{w}w.0 \mid \bar{c}b.0 \\ \xrightarrow{\bar{w}w} & \bar{c}b.0 \\ \xrightarrow{\bar{c}b} & 0 \end{aligned}$$

Beachte: Der Name c ist frei wählbar

$P \rightarrow Q$ gdw. $P \xrightarrow{\tau} Q'$ wobei $Q' \equiv Q$

Einschub: Analoge Definition mit Fixpunkten

- Sei $Proc$ die Menge aller Prozesse des π -Kalküls.

Sei $F : (Proc \times Proc) \rightarrow (Proc \times Proc)$ die folgende Funktion:
 $(P, Q) \in F(\eta)$ 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 der **größte Fixpunkt** der Funktion F .
 (die größte Teilmenge X von $Proc$, für die $F(X) = X$ gilt)
- **Prinzip der Coinduktion**: Jede Relation η , die **dicht** bezüglich F ist, ist auch im größten Fixpunkt enthalten.
- Eine Relation η ist **F -dicht**, wenn $\eta \subseteq F(\eta)$
- F -dichte Relation = starke Bisimulation

Definition

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 $(P', Q') \in \mathcal{R}$.

Prozesse P und 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

$\sim_{b, strong}$ ist die **größte** starke Bisimulation.

Starke Bisimulation (2)

Starke Bisimilarity ist eine Äquivalenzrelation, aber keine Kongruenz:

- $P = \bar{z}b.0 \mid a(c).0$
- $Q = \bar{z}b.a(c).0 + a(c).\bar{z}b.0$
- Es gilt $P \sim_{b, strong} Q$:
 $\mathcal{R} = \{(P, Q), (a(c).0, a(c).0), (\bar{z}b.0, \bar{z}b.0), (0, 0)\}$ ist eine starke Bisimulation
- Allerdings gilt $x(z).P \not\sim_{b, strong} x(z).Q$
- 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.

Definition

Zwei Prozesse P, Q sind **stark voll bisimilar** (Notation $P \sim_{b, \text{strong, full}} Q$), wenn für alle Substitutionen σ , die Namen für Namen ersetzen gilt: $\sigma(P) \sim_{b, \text{strong}} \sigma(Q)$.

- Man kann nachweisen, dass $\sim_{b, \text{strong, full}}$ eine Kongruenz ist.

Bisimulation (2)

- Bisimilarity ist keine Kongruenz
- Volle Bisimilarity ist eine Kongruenz.

Satz

Für alle Prozesse P, Q gilt:

- $P \sim_{b, \text{strong}} Q \implies P \sim_b Q$
- $P \sim_{b, \text{strong, full}} Q \implies P \sim_{b, \text{full}} Q$.

Weiterhin gilt $\sim_{b, \text{strong}} \neq \sim_b$ und $\sim_{b, \text{strong, full}} \neq \sim_{b, \text{full}}$.

- Vernachlässigung von τ -Transitionen
- $\overset{\tau}{\Rightarrow} :=$ reflexiv-transitive Hülle von $\overset{\tau}{\rightarrow}$
- $\overset{\alpha}{\Rightarrow} := \overset{\tau}{\Rightarrow} \overset{\alpha}{\rightarrow} \overset{\tau}{\Rightarrow}$

Eine binäre Relation \mathcal{R} auf Prozessen ist eine **Bisimulation**, wenn für alle $(P, Q) \in \mathcal{R}$ gilt

- Falls $P \overset{\alpha}{\rightarrow} P'$, dann gibt es einen Prozess Q' , so dass gilt $Q \overset{\alpha}{\Rightarrow} Q'$ und $(P', Q') \in \mathcal{R}$
- Falls $Q \overset{\alpha}{\rightarrow} Q'$, dann gibt es einen Prozess P' , so dass gilt $P \overset{\alpha}{\Rightarrow} P'$ und $(P', Q') \in \mathcal{R}$.

Prozesse P und 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** ($P \sim_{b, \text{full}} Q$) falls $\sigma(P) \sim_b \sigma(Q)$ für alle Substitutionen σ gilt.

Barbs

Für Prozess P schreiben wir

- $P \uparrow^x$, falls $P \xrightarrow{x(y)} P'$
- $P \uparrow^{\bar{x}}$, falls $P \xrightarrow{\bar{x}y} P'$ oder $P \xrightarrow{\bar{x}(y)} P'$
- für $\beta \in \{x, \bar{x}\}$: $P \downarrow_\beta$ falls es einen Prozess Q gibt, so dass $P \xrightarrow{\tau, *}_\beta Q$ und $Q \uparrow^\beta$

Lemma

Für alle Prozesse des synchronen π -Kalküls (mit Summen) gilt:

- $P \dot{\rightarrow}^x$ genau dann, wenn $P \equiv \nu v_1 \dots \nu v_n. (x(y).P' + M \mid Q)$ wobei $x \neq v_i$.
- $P \dot{\rightarrow}^{\bar{x}}$ genau dann, wenn $P \equiv \nu v_1 \dots \nu v_n. (\bar{x}y.P' + M \mid Q)$ wobei $x \neq v_i$.

Deshalb: Man kann auch Reduktion statt labeled Transitionen verwenden!

Barbed Kongruenz

Definition

Zwei Prozesse P, Q sind **barbed kongruent** (geschrieben $P \sim_{c,b,barbed} Q$) falls für alle Kontexte C gilt: $C[P] \sim_{b,barbed} C[Q]$.

Theorem

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.

Definition

Eine binäre Relation \mathcal{R} auf Prozessen ist eine **barbed Bisimulation** falls für alle $(P, Q) \in \mathcal{R}$ gilt:

- Falls $P \dot{\rightarrow}^x$ (bzw. $P \dot{\rightarrow}^{\bar{x}}$), dann $Q \downarrow_x$ (bzw. $(Q \downarrow_{\bar{x}})$)
- Falls $Q \dot{\rightarrow}^x$ (bzw. $Q \dot{\rightarrow}^{\bar{x}}$), dann $P \downarrow_x$ (bzw. $(P \downarrow_{\bar{x}})$)
- Falls $P \xrightarrow{\tau} P'$, dann existiert Q' mit $Q \xrightarrow{\tau} Q'$ und $(P', Q') \in \mathcal{R}$
- Falls $Q \xrightarrow{\tau} Q'$, dann existiert P' mit $P \xrightarrow{\tau} P'$ 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.

May-Testing

Die **may-testing** Präordnung \leq_{may} :

$P \leq_{may} Q$ gdw. für alle Kontexte C , für alle Namen x :
 $C[P] \downarrow_x \implies C[Q] \downarrow_x$ und $C[P] \downarrow_{\bar{x}} \implies C[Q] \downarrow_{\bar{x}}$.

May-testing Äquivalenz \sim_{may} :

$P \sim_{may} Q$ gdw. $P \leq_{may} Q$ und $Q \leq_{may} P$

May-testing Äquivalenz

Satz

Die may-testing Äquivalenz ist eine Kongruenz

\sim_{may} ist sehr grob-körnig:

Satz

Für alle Prozesse P, Q gilt: $P \sim_{c,b,barbed} Q \implies P \sim_{may} Q$.

Die Umkehrung gilt nicht, z.B.

$a(x).0 \oplus (b(x).0 \oplus c(x).0)$ und $(a(x).0 \oplus b(x).0) \oplus c(x).0$
wobei $P \oplus Q := \tau.P + \tau.Q$

Should-Testing

Should-Barb

Für $\mu \in \{x, \bar{x}\}$ schreiben wir $P \Downarrow_\mu$ (P muss einen Barb μ haben), falls

$$\forall Q : P \xrightarrow{*} Q \implies Q \Downarrow_\mu.$$

should-testing Präordnung \leq_{should} :

$P \leq_{should} Q$ gdw. für alle Kontexte C , für alle Namen x :
 $C[P] \Downarrow_x \implies C[Q] \Downarrow_x$ und $C[P] \Downarrow_{\bar{x}} \implies C[Q] \Downarrow_{\bar{x}}$.

Should-testing Äquivalenz \sim_{should} :

$P \sim_{should} Q$ gdw. $P \leq_{should} Q$ und $Q \leq_{should} P$

May-testing Äquivalenz

Zu grob:

$x(y).0 \oplus 0$ und $x(y).0$ sind may-testing äquivalent!

Should-Testing

Satz

Für alle Prozesse P, Q gilt:
 $P \sim_{c,b,barbed} Q \implies P \sim_{should} Q$.

Satz

Für alle Prozesse P, Q gilt:
 $P \sim_{should} Q \implies P \sim_{may} Q$.

Theorem

$$\sim_{b, \text{strong, full}} \subseteq \sim_{b, \text{full}} \subseteq \sim_{b, \text{c, barbed}} \subseteq \sim_{\text{should}} \subseteq \sim_{\text{may}}$$