# Höherstufige Kalküle

Mark Kaminski
Betreuer: Andreas Rossberg

Seminar Theorie kommunizierender Systeme: Der $\pi$-Kalkül SS 2004
Lehrstuhl Programmiersysteme Prof. Gert Smolka
Universität des Saarlandes

17. Oktober 2004

**Zusammenfassung**

Der $\pi$-Kalkül, entwickelt von Robin Milner, bildet eine formale Grundlage für den Entwurf und die Analyse paralleler, kommunizierender Systeme. Bei seiner Basisvariante handelt es sich um einen Kalkül erster Stufe. Im Folgenden soll zunächst eine höherstufige Erweiterung des Kalküls vorgestellt und mit dem Basiskalkül verglichen werden. Dabei interessieren wir uns vor allem dafür, ob der neue Fomalismus expressiver ist als der $\pi$-Kalkül erster Stufe. Den Vergleich weiten wir anschließend auf den $\lambda$-Kalkül als den wichtigsten Vertreter höherstufiger Formalismen aus und zeigen durch Reduktion der Auswertung im ungetypten $\lambda$-Kalkül auf Berechnung im $\pi$-Kalkül die Turing-Vollständigkeit des letzteren Systems. Abschließend wird kurz aufgezeigt, welche Rolle die gewonnenen Erkenntnisse für die praktische Verwendung des $\pi$-Kalküls spielen.

## 1 Einleitung

Der $\pi$-Kalkül wurde in den späten 80er Jahren von Robin Milner entwickelt. Dabei handelt es sich um eine Erweiterung einer früheren Entwicklung Milners, des Kalküls CCS (a Calculus of Communicating Systems, siehe [3],[4]). Wie auch CCS, dient der $\pi$-Kalkül der Modellierung interaktiver Systeme mit parallelen, kommunizierenden Prozessen. Der $\pi$-Kalkül erweitert die Möglichkeiten von CCS zur Änderung der Verbindungs- und Kommunikationsstruktur innerhalb eines interaktiven Systems. Diese Erweiterung ermöglicht es, unterschiedliche Arten von Mobilität in interaktiven Systemen zu modellieren. Eine tiefergehende Diskussion von Mobilität im $\pi$-Kalkül sowie Beispiele für die Anwendung des $\pi$-Kalküls auf unterschiedliche mobile Systeme können [1] entnommen werden. Blicken wir auf die wesentlich länger erforsche Welt der sequentiellen Berechnung, insbesondere auf den $\lambda$-Kalkül als einen der wichtigsten Vertreter sequentieller Berechnungsmodelle, so können wir einige Parallelen zum $\pi$-Kalkül erkennen. Die Namen im $\pi$-Kalkül sind als primitive Informationseinheiten vergleichbar mit Variablen im $\lambda$-Kalkül. Prozesse, die mit Namen agieren, können wir zusammengesetzten Ausdrücken im $\lambda$-Kalkül zuordnen. Da Prozesse durch Kommunikation von Namen gesteuert werden, handelt es sich dabei sicherlich um funktionale, höherstufige Objekte.

Jegliche Interaktion und Berechnung basiert im $\pi$-Kalkül auf der Kommunikation von Namen. Die Menge von Namen wird einerseits verwendet um Kommunikationskanäle zu bezeichnen, andererseits sind Namen auch der einzig mögliche Gegenstand der Kommunikation. Insofern ist der $\pi$-Kalkül ein Berechungssystem erster Stufe. Der $\lambda$-Kalkül ist dagegen höherstufig. Die Applikation als der grundlegende Berechnungsschritt im $\lambda$-Kalkül akzeptiert sowohl in der Funktor- als auch in der Argumentposition beliebig komplexe $\lambda$-Terme.
Man mag sich fragen, ob ein höherstufiger Kalkül deswegen praktische Vorteile gegenüber einem Kalkül erster Stufe wie dem $\pi$-Kalkül (wir schreiben auch FO$\pi$ für First-order $\pi$-Calculus) besitzt. Gegebenenfalls wollen wir auch den $\pi$-Kalkül zu einem höherstufigen System erweitern können. In dem Zusammenhang interessant ist insbesondere auch, ob wir durch eine höherstufige Erweiterung des Kalküls an Berechnungskraft gewinnen. Wenn dies nicht der Fall ist, muss es einen Weg geben, Terme des höherstufigen Kalküls in den Kalkül erster Stufe zu übersetzen. Umgekehrt kann man, indem man eine solche Übersetzung findet, zeigen, dass die Berechnungskraft von FO$\pi$ der seiner höherstufigen Erweiterung (wir schreiben auch HO$\pi$ für Higher-order $\pi$-Calculus) gleicht.
Nach der Betrachtung von HO$\pi$ wollen wir uns das Verhältnis zwischen FO$\pi$ und dem $\lambda$-Kalkül genauer ansehen. Auch hier konstruieren wir eine Übersetzungsfunktion, die $\lambda$-Terme in entsprechende $\pi$-Terme überführt. So können wir zeigen, dass der $\pi$-Kalkül mindestens die Berechnungskraft des ungetypten $\lambda$-Kalküls hat. Da dieser ein universelles Berechnungsmodell ist, bedeutet das, dass auch FO$\pi$ universelle Berechnungskraft besitzt.

# 2 Höherstufiger $\pi$-Kalkül

## 2.1 Syntax

Wie sieht eine höherstufige Erweiterung des $\pi$-Kalküls nun aus? Um diese Frage zu beantworten sehen wir uns zunächst die Syntax der polyadischen Variante von FO$\pi$ in Tabelle 1 an. Mit $\mathcal{N}$ bezeichnen wir dabei die Menge aller Namen.

Tabelle 1: Polyadischer $\pi$-Kalkül (Syntax)

| | | | |
|---|---|---|---|
| Variablen: | | | |
| $x$ | $\in$ | $\mathcal{N}$ | Namen |
| Agenten: | | | |
| $F$ | ::= | $(\vec{x}).P$ | Abstraktion |
| $C$ | ::= | new $\vec{x}$ $\langle\vec{y}\rangle.P$ | Konkretisierung |
| Prozessausdrücke: | | | |
| $P$ | ::= | $xF \mid \overline{x}C \mid \tau.P \mid \sum_{i\in I} P_i \mid P_1 \mid P_2 \mid$ new $\vec{x}$ $P \mid !P$ | |

Welche Objekte neben Namen soll HO$\pi$ kommunizieren können? In Frage kommen dabei Abstraktionen, Konkretisierungen und Prozessausdrücke. Bedenkt man, dass sowohl Abstraktionen als auch Konkretisierungen Prozessausdrücke als Teilmenge beinhalten, brauchen die letzteren nicht gesondert betrachtet zu werden. Angesichts der weitgehenden Symmetrie zwischen Abstraktionen und

Konkretisierungen müssen wir uns auch nicht beide Typen von Nachrichten ansehen. Wir entscheiden uns für die Betrachtung von Abstraktionen als der im $\pi$-Kalkül gängigeren Art, parametrisierte Prozesse darzustellen.
Um Abstraktionen kommunizieren zu können, müssen wir einen Weg haben, diese an Variablen zu binden. Dazu führen wir eine neue Menge von Variablen über Abstraktionen ein und bezeichnen diese mit $\mathcal{F}$. Eine Variable aus dieser Menge kann genauso verwendet werden wie die entsprechende Abstraktion. Dazu erweitern wir die Syntax von Prozessausdrücken um Applikation von Abstraktionen auf Konkretisierungen. Der Einfachheit halber sollen Variablen über Abstraktionen im Gegensatz zu Namen ausschließlich im Kopf von Abstraktionen gebunden werden können. Um eine Variable an eine Abstraktion zu binden, muss die Abstraktion als Nachricht im Kopf einer Konkretisierung kommuniziert werden. Wir führen den Begriff eines Werts als eines Objekts ein, an den Variablen gebunden werden können, und erlauben Konkretisierungen beliebige Werte zu versenden. In diesem Sinn sind natürlich auch Namen Werte.
Eine Übersicht der geänderten Syntax:

Tabelle 2: HO$\pi$ (Syntax)

Variablen:

| | | | |
|---|---|---|---|
| $x$ | $\in$ | $\mathcal{N}$ | Namen |
| $f$ | $\in$ | $\mathcal{F}$ | Variablen über Abstraktionen |
| $v$ | $\in$ | $\mathcal{N} \cup \mathcal{F}$ | Variablen über Werte |

Werte:

| | | |
|---|---|---|
| $V$ | ::= | $x \mid F$ |

Agenten:

| | | | |
|---|---|---|---|
| $F$ | ::= | $(\vec{v}).P \mid f$ | Abstraktion |
| $C$ | ::= | new $\vec{x}$ $\langle\vec{V}\rangle.P$ | Konkretisierung |

Prozessausdrücke:

$P \quad ::= \quad xF \mid \overline{x}C \mid \tau.P \mid \sum_{i\in I} P_i \mid P_1|P_2 \mid \text{new } \vec{x}\ P \mid !P \mid FC$

## 2.2 Reaktion und Substitution

Das Reaktionsverhalten von HO$\pi$ wollen wir uns an einem einfachen Beispiel ansehen. Dazu betrachten wir zwei Prozessausdrücke:

$$P = \text{new}\, w\, (\overline{x}\langle F\rangle.P') \quad Q = x(f).(f\langle u\rangle \mid f\langle v\rangle \mid Q')$$

Bei $P'$ und $Q'$ handelt es sich ebenfalls um Prozessausdrücke, wogegen $F$ eine einstellige Abstraktion ist. Wir nehmen an, dass $F$ und $P'$ über den Namen $w$ miteinander kommunizieren können und daher $w \in FV(F) \cap FV(P')$ gilt. $f$ ist eine Variable über Abstraktionen. $Q'$ soll $f$ nicht weiter verwenden ($f \notin FV(Q')$).
Entsprechend sieht die von uns erwartete Reaktion aus:

$$P \mid Q \rightarrow \text{new}\, w\, (P' \mid F\langle u\rangle \mid F\langle v\rangle) \mid Q'$$

Wie wir sehen, können in unserem erweiterten Kalkül Variablen durch komplexe Terme substituiert werden. Die praktischen Vorteile, die uns diese Eigenschaft bringt, sind nicht schwer zu erkennen. So lässt sich ein komplexer Ausdruck leicht in einfachere funktional zusammengehörige Komponenten zerlegen, die dann durch den Substitutionsmechanismus wieder zusammengefügt werden. Auch lassen sich Terme bauen, die mit Teiltermen parametrisiert werden können. Will man einen Teilterm an mehreren Stellen verwenden, genügt es, an diesen Stellen die zugehörige Variable einzusetzen. So können wir den Mechanismus der Replikation bei Bedarf durch einen aus höheren Programmiersprachen vertrauten Mechanismus des Prozeduraufrufs ersetzen. Da gebundene Terme Abstraktionen sein können, lassen auch sie sich parametrisieren, was uns zusätzliche Flexibilität bringt.

## 2.3 Expressivität und Übersetzung in FO$\pi$

Nutzt man die Möglichkeiten von HO$\pi$ geschickt aus, so lassen sich viele Terme wesentlich kompakter und übersichtlicher schreiben. Durch die Auslagerung von Teiltermen in getrennte, in ihrer Verwendung an Prozeduren angelehnte Komponenten lassen sich Implementierungsdetails verbergen und auf diese Weise die Modularität und die Wiederverwendbarkeit des resultierenden Codes steigern.
Was sich im höherstufigen Kalkül gegenüber FO$\pi$ allerdings nicht geändert hat, ist die Berechnungskraft. Um dies zu zeigen überlegen wir uns, wie höherstufige Terme nach FO$\pi$ übersetzt werden können. Eine formale Übersetzung von HO$\pi$ nach FO$\pi$ ist auch aus anderen Gründen sinnvoll. So kann der höherstufige Kalkül in vollem Maße von der Vorarbeit profitieren, die für FO$\pi$ geleistet wurde. Denn eine formale Übersetzung ermöglicht es, bewiesene Eigenschaften von FO$\pi$ durch formale Argumentation auf den höherstufigen Fall zu übertragen. Auch Begriffe und Beweistechniken auf erster Stufe können so leichter auf HO$\pi$ ausgeweitet werden. Somit brauchen wir HO$\pi$ nicht als einen neuen, von FO$\pi$ grundsätzlich verschiedenen Kalkül zu sehen, sondern können den höherstufigen Kalkül als eine bequemere abstrahierende Notation für FO$\pi$ nutzen.
Die grundsätzliche Vorgehensweise bei der Übersetzung von HO$\pi$ in FO$\pi$ entlehnen wir der Übersetzung von Quellcode höherer, insbesondere funktionaler Programmiersprachen in Maschinensprache. Natürlich können wir dabei nur von einer konzeptueller Analogie sprechen; die technische Realisierung wird aufgrund der stark unterschiedlichen Berechnungsmodelle kaum Ähnlichkeiten aufweisen können.
Will man ein höherstufiges System in ein System erster Stufe übersetzen, so müssen mehrere Probleme gelöst werden. Zunächst einmal muss Kommunikation höherstufiger Objekte geeignet durch Kommunikation auf erster Stufe simuliert werden. Bei der Übersetzung in Maschinensprache handelt es sich bei den höherstufigen Objekten um Prozeduren. Die Übersetzung in Maschinensprache legt die übersetzten Prozeduren im Programmspeicher so ab, dass sie mehrfach aufgerufen werden können, und ersetzt die Kommunikation von Prozeduren durch die Kommunikation von Zeigern auf die den Prozeduren entsprechenden Fragmente des Programmcodes. Bei der Übersetzung in FO$\pi$ können wir die mehrfache Instanziierbarkeit von Abstraktionen durch Replikation simulieren. Als Zeiger auf Abstraktionen nehmen wir erwartungsgemäß Namen. Die mehrfache Verwendung einer höherstufigen Variablen entspricht dann der mehrfachen Parametrisierung einer replizierten Instanz einer Abstraktion über

den dem Variablennamen eindeutig zugeordenen Kanalnamen. Ebenso wie es notwendig ist, bei der Übersetzung in Maschinensprache die verschiedenen Prozeduren zugeordneten Speicherbereiche zu trennen, ist es bei der Übersetzung in FO$\pi$ von entscheidender Bedeutung, für eine strikte Trennung von Namen zu sorgen. Durch den inhärenten Indeterminismus des $\pi$-Kalküls müssen wir uns sogar noch mehr vorsehen.
Nun wird es Zeit, uns die Übersetzung im Detail anzusehen. Bei der Darstellung in Tabelle 3 handelt es sich um eine Variante der entsprechenden Übersetzungsfunktion aus [2].

Tabelle 3: Übersetzungsfunktion HO$\pi$ $\rightarrow$ FO$\pi$

Abstraktionen:

$$\llbracket f \rrbracket = (\vec{x}).\overline{f}\langle\vec{x}\rangle \quad (3.1)$$
$$\llbracket (\vec{v}).P \rrbracket = (\vec{v}).\llbracket P \rrbracket \quad (3.2)$$

Prozessausdrücke:

$$\llbracket xF \rrbracket = x\llbracket F \rrbracket \quad (3.3)$$
$$\llbracket \overline{x}\langle F_1 ... F_m y_1 ... y_n\rangle.P \rrbracket = \text{new}\, z_1 ... z_m\, (\overline{x}\langle z_1 ... z_m y_1 ... y_n\rangle. (\llbracket P \rrbracket \mid !z_1\llbracket F_1 \rrbracket \mid ... \mid !z_m\llbracket F_m \rrbracket)) \quad (3.4)$$
$$\llbracket \tau.P \rrbracket = \tau.\llbracket P \rrbracket \quad (3.5)$$
$$\llbracket \sum_{i\in I} P_i \rrbracket = \sum_{i\in I} \llbracket P_i \rrbracket \quad (3.6)$$
$$\llbracket P_1 \mid P_2 \rrbracket = \llbracket P_1 \rrbracket \mid \llbracket P_1 \rrbracket \quad (3.7)$$
$$\llbracket \text{new}\, \vec{x}\, P \rrbracket = \text{new}\, \vec{x}\, \llbracket P \rrbracket \quad (3.8)$$
$$\llbracket !P \rrbracket = !\llbracket P \rrbracket \quad (3.9)$$
$$\llbracket F(\text{new}\, \vec{z}\, \langle\vec{y}\rangle.P) \rrbracket = \text{new}\, \vec{z}\, (\llbracket F \rrbracket \langle\vec{y}\rangle \mid \llbracket P \rrbracket) \quad (3.10)$$

Etwas erklärungsbedürftig ist die Übersetzung von Prozessausdrücken, welche die Übermittlung von Abstraktionen beschreiben (Regel 3.4). Diese wird simuliert, indem zunächst ein frischer, d.h. noch nirgends verwendeter Kanalname alloziert wird. Der Name wird dann an Stelle der Abstraktion übermittelt und dient zu ihrer Instanzierung. Um eine Abstraktion mehrfach instanziierbar zu machen, wird sie repliziert. Will der Empfänger des Kanalnamens die entsprechende Abstraktion verwenden, sendet er also die Parameter über den empfangenen Kanal.
Diese Verwendung erklärt auch das Verhalten von Abstraktionen (Regeln 3.1, 3.2). Da nach der Übersetzung Konkretisierungen ausschließlich Namen übermitteln, braucht die Bindung im Kopf von Abstraktionen nicht angepasst zu werden. Allerdings muss man bedenken, dass Variablen, die vorher Abstraktionen beschrieben, nun wie Ausgabekanäle zu verwenden sind. Um die Regel 3.1 anzuwenden muss zusätzlich die Stelligkeit von $f$ bekannt sein.
Die Übersetzung von Applikation (Regel 3.10) lehnt sich an die Definition des Applikationsbegriffes in FO$\pi$ an:

$$F \mathbin{@} \text{new}\, \vec{z}\, \langle y\rangle.P \stackrel{\text{def}}{=} \text{new}\, \vec{z}\, (F\langle y\rangle \mid P)$$

Die Funktionsweise der übrigen Regeln ist offensichtlich.

## 2.4 Übersetzung am Beispiel

Versuchen wir nun die Übersetzungsfunktion auf unser Beispiel aus Abschnitt 2.2 anzuwenden. Dabei machen wir freien Gebrauch von der Assoziativität und Kommutativität der Parallelkomposition.

$$
\begin{array}{lcll}
[\![P]\!] & = & [\![\mathrm{new}\, w\, (\overline{x}\langle F\rangle.P')]\!] & \\
 & = & \mathrm{new}\, w\, [\![(\overline{x}\langle F\rangle.P')]\!] & (3.8) \\
 & = & \mathrm{new}\, w\, (\mathrm{new}\, y\, (\overline{x}\langle y\rangle.([\![P']\!] \mid\, !y[\![F]\!]))) & (3.4) \\
 & = & \mathrm{new}\, wy\, (\overline{x}\langle y\rangle.([\![P']\!] \mid\, !y[\![F]\!])) & \text{Notation} \\
 & = & \mathrm{new}\, wy\, (\overline{x}\langle y\rangle.(\widehat{P'} \mid\, !y\widehat{F})) & \widehat{P'} := [\![P']\!],\ \widehat{F} := [\![F]\!]
\end{array}
$$

$$
\begin{array}{lcll}
[\![Q]\!] & = & [\![x(f).(f\langle u\rangle \mid f\langle v\rangle \mid Q')]\!] & \\
 & = & x[\![(f).(f\langle u\rangle \mid f\langle v\rangle \mid Q')]\!] & (3.3) \\
 & = & x(f).[\![(f\langle u\rangle \mid f\langle v\rangle \mid Q')]\!] & (3.2) \\
 & = & x(f).([\![f\langle u\rangle]\!] \mid [\![f\langle v\rangle]\!] \mid [\![Q']\!]) & (3.7) \\
 & = & x(f).(([\![f]\!]\langle u\rangle \mid [\![0]\!]) \mid ([\![f]\!]\langle v\rangle \mid [\![0]\!]) \mid [\![Q']\!]) & (3.10) \\
 & = & x(f).(([\![f]\!]\langle u\rangle \mid 0) \mid ([\![f]\!]\langle v\rangle \mid 0) \mid [\![Q']\!]) & (3.6) \\
 & \equiv & x(f).([\![f]\!]\langle u\rangle \mid [\![f]\!]\langle v\rangle \mid [\![Q']\!]) & P \mid 0 \equiv P \\
 & = & x(f).(((x).f\langle x\rangle)\langle u\rangle \mid ((x).f\langle x\rangle)\langle v\rangle \mid [\![Q']\!]) & (3.1) \\
 & = & x(f).(f\langle u\rangle \mid f\langle v\rangle \mid [\![Q']\!]) & ((\vec{x}).P)\langle\vec{y}\rangle = \{^{\vec{y}}/_{\vec{x}}\}P \\
 & = & x(f).(f\langle u\rangle \mid f\langle v\rangle \mid \widehat{Q'}) & \widehat{Q'} := [\![Q']\!]
\end{array}
$$

Natürlich muss unsere Übersetzung das Reaktionsverhalten von $P$ und $Q$ erhalten. Allerdings ist es auch klar, dass manche Reaktionsschritte in HO$\pi$ in FO$\pi$ nur durch mehrere Reaktionen zu simulieren sind. Seien $\widehat{P} := [\![P]\!]$, $\widehat{Q} := [\![Q]\!]$. Erinnern wir uns an die ursprüngliche Reaktion

$$P \mid Q \to \mathrm{new}\, w\, (P' \mid F\langle u\rangle \mid F\langle v\rangle) \mid Q'$$

so erwarten wir

$$\widehat{P} \mid \widehat{Q} \to^{*}\sim \mathrm{new}\, w\, (\widehat{P'} \mid \widehat{F}\langle u\rangle \mid \widehat{F}\langle v\rangle) \mid \widehat{Q'}$$

Und tatsächlich

$$
\begin{array}{lcll}
\widehat{P} \mid \widehat{Q} & & & \\
 & = & \mathrm{new}\, wy\, (\overline{x}\langle y\rangle.(\widehat{P'} \mid\, !y\widehat{F})) \mid x(f).(\overline{f}\langle u\rangle \mid \overline{f}\langle v\rangle \mid \widehat{Q'}) & \\
 & \equiv & \mathrm{new}\, wy\, (\overline{x}\langle y\rangle.(\widehat{P'} \mid\, !y\widehat{F}) \mid x(f).(\overline{f}\langle u\rangle \mid \overline{f}\langle v\rangle \mid \widehat{Q'})) & w \notin FV(\widehat{Q'}) \\
 & \to & \mathrm{new}\, wy\, ((\widehat{P'} \mid\, !y\widehat{F}) \mid (\overline{y}\langle u\rangle \mid \overline{y}\langle v\rangle \mid \widehat{Q'})) & \\
 & \equiv & \mathrm{new}\, wy\, (\widehat{P'} \mid\, !y\widehat{F} \mid \overline{y}\langle u\rangle \mid \overline{y}\langle v\rangle) \mid \widehat{Q'} & w, f \notin FV(\widehat{Q'}) \\
 & \equiv & \mathrm{new}\, wy\, (!y\widehat{F} \mid \widehat{P'} \mid \overline{y}\langle u\rangle \mid y\widehat{F} \mid \overline{y}\langle v\rangle \mid y\widehat{F}) \mid \widehat{Q'} & !P \equiv P \mid !P \\
 & \to & \mathrm{new}\, wy\, (!y\widehat{F} \mid \widehat{P'} \mid \widehat{F}\langle u\rangle \mid \widehat{F}\langle v\rangle) \mid \widehat{Q'} & \\
 & \equiv & \mathrm{new}\, w\, (\mathrm{new}\, y\, (!y\widehat{F}) \mid \widehat{P'} \mid \widehat{F}\langle u\rangle \mid \widehat{F}\langle v\rangle) \mid \widehat{Q'} & y \notin FV(\widehat{F}) \cup FV(\widehat{P'}) \\
 & \sim & \mathrm{new}\, w\, (\widehat{P'} \mid \widehat{F}\langle u\rangle \mid \widehat{F}\langle v\rangle) \mid \widehat{Q'} & \mathrm{new}\, y\, (!y\widehat{F}) \sim 0
\end{array}
$$

Die Annahmen $w, f \notin FV(\widehat{Q'})$ und $y \notin FV(\widehat{F}) \cup FV(\widehat{P'})$ sind ohne Beschränkung der Allgemeinheit möglich, da die strukturelle Kongruenz unter $\alpha$-Konversion erhalten bleibt.

# 3 $\lambda$-Kalkül

## 3.1 Berechnung im $\lambda$-Kalkül

Der $\lambda$-Kalkül wurde um 1934 von Alonzo Church entwickelt. Er erlaubt die Modellierung des Berechnungsprozesses als eine Folge von Reduktionen auf funktionalen Termen. Eine wichtige Rolle spielen dabei die Mechanismen der Variablenbindung und der Substitution. Der ungetypte $\lambda$-Kalkül, die älteste Variante des Kalküls, ist in der Lage, alle intuitiv berechenbaren Funktionen darzustellen. Somit handelt es sich dabei um ein universelles Berechnungsmodell. Als solches bildet der ungetypte $\lambda$-Kalkül die Grundlage für alle funktionalen Programmiersprachen. Weitere wichtige Varianten des Kalküls sind der einfach getypte $\lambda$-Kalkül und seine Derivate.
Da wir uns für die Frage interessieren, ob FO$\pi$ universelle Berechnungskraft besitzt, reicht es für uns die ungetypte Variante des $\lambda$-Kalküls zu betrachten. Seine Terme haben die folgende einfache Syntax:

$$M, N ::= x \mid \lambda x.M \mid M\ N$$

Ähnlich einfach verläuft im $\lambda$-Kalkül der Berechnungsprozess. Im Wesentlichen lässt er sich auf die als $\beta$-Reduktion bekannte Termumformungsregel zurückführen:

$$(\lambda x.M)N \xrightarrow{\beta} \{^N/_x\}M$$

Wie schon in der Einleitung festgestellt wurde, handelt es sich beim $\lambda$-Kalkül um ein höherstufiges System. Wollen wir eine Übersetzung von $\lambda$-Termen in FO$\pi$ konstruieren, werden wir daher auf gewisse Probleme stoßen. Dabei wird die Erfahrung, die wir bei der Übersetzung von HO$\pi$ gesammelt haben, sich als nützlich erweisen.
Eine weitere Schwierigkeit bei der Übersetzung ist auf eine Eigenschaft des $\lambda$-Kalküls zurückzuführen, die als Konfluenz bezeichnet wird. Ähnlich wie im $\pi$-Kalkül ist für ein $\lambda$-Term im Allgemeinen mehr als eine Reduktionskette möglich. Konfluenz besagt, dass zwei beliebige Terme, die durch zwei Reduktionsketten aus einem und demselben Ursprungsterm gebildet wurden, durch passende Reduktionsketten wieder auf eine gemeinsame Form gebracht werden können. Beim $\pi$-Kalkül ist diese Eigenschaft nicht erfüllt. Um dies zu sehen, reicht ein einfaches Beispiel:

$$a \quad \leftarrow \quad \overline{x} \mid x.a + x.b \quad \rightarrow \quad b$$

Um bei unserer Übersetzung nicht auf Konfluenz achten zu müssen, wollen wir für jeden $\lambda$-Term bloß eine einzige, dem Term eindeutig zugeordnete Reduktionskette betrachten. Wir spezifizieren diese Kette mittels einer so genannten Auswertungsstrategie. Eine Auswertungsstrategie oder Reduktionsordnung ist ein Regelsystem, welches zu einem Term eine oder mehrere Stellen angibt, an denen die nächste Reduktion stattfinden kann. In unserem Fall soll die Stelle eindeutig sein.
Wir wählen eine einfache Auswertungsstrategie, die mit folgenden zwei Regeln beschrieben werden kann:

$$\beta : \frac{}{(\lambda x.M)N \longrightarrow \{^N/_x\}M} \qquad \mu : \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}$$

Die Regeln sorgen dafür, dass bei einem Term zuerst immer der Redex ausgewertet wird, bei dem der Funktor am weitesten links beginnt. Damit handelt es sich bei unserer Auswertungsstrategie um eine Variante der so genannten normalen Reduktionsordnung. Wie aus den Regeln deutlich wird, verzichten wir auf die in der Praxis eher unübliche Auswertung im Rumpf von Abstraktionen. Als Beispiel sehen wir uns die Auswertung des Terms $(\lambda x.(\lambda y.y)xx)(\lambda x.\lambda y.zy)$ an. Bei jedem Reduktionsschritt markieren wir den aktuellen Redex, indem wir seinen Funktor und das entsprechende Argument mit $F$ bzw. $A$ indizieren.

$$\begin{aligned}
[\lambda x.(\lambda y.y)xx]_F[\lambda x.\lambda y.zy]_A &\rightarrow [\lambda y.y]_F[\lambda x.\lambda y.zy]_A(\lambda x.\lambda y.zy) \\
&\rightarrow [\lambda x.\lambda y.zy]_F[\lambda x.\lambda y.zy]_A \\
&\rightarrow \lambda y.zy
\end{aligned}$$

Vergleicht man unsere Reduktionsordnung mit Auswertungsstrategien in Programmiersprachen, so entspricht sie einer naiver Implementierung bedarfsgesteuerter Berechnung. Naiv deswegen, weil das mehrfache Vorkommen eines Terms in einem größeren Term nicht als solches erkannt wird, so dass ein und derselbe Term unter Umständen mehrfach ausgewertet wird. Der dadurch verursachte Effizienzverlust ist für unsere Betrachtungen aber auch völlig uninteressant.

## 3.2 Kontrollfluss und Continuation Passing Style

Eine weitere Schwierigkeit bei der Übersetzung von $\lambda$-Termen in Berechnungssysteme erster Stufe besteht in der Modellierung des Kontrollflusses. Tatsächlich kann der Mechanismus der Funktionsapplikation im $\pi$-Kalkül nicht direkt nachgebildet werden. Ein weit verbreiteter Ansatz zur Lösung des Problems besteht in der Verwaltung eines Aufrufstapels. Wir entscheiden uns für einen anderen Weg, indem wir die zu übersetzenden Terme zunächst in eine äquivalente Form überführen, die der Übersetzung in einen anders strukturierten Formalismus wie den $\pi$-Kalkül zugänglicher ist. Diese Termdarstellung wird im Englischen mit dem Begriff continuation passing style (CPS) bezeichnet.
Um die Funktionsweise von CPS zu erklären, sollte man zunächst den Begriff einer Fortsetzung (continuation) klären. Dabei handelt es sich um eine Funktion, die im Kontext ihrer Verwendung den „Rest“ einer Berechnung beschreibt. Funktionen in CPS terminieren niemals mit einem impliztem Rücksprung an die Stelle, an der sie angewendet wurden. Stattdessen bekommen sie ihre Fortsetzung als ein zusätzliches Argument und terminieren, indem sie das Ergebnis ihrer Anwendung an die Fortsetzung übergeben. Da Funktionen niemals zum Aufrufer zurückkehren, benötigt man auch nicht die Hilfsstruktur eines Aufrufstapels. Für die von uns gewählte Auswertungsstrategie sieht die Übersetzungsfunktion von $\lambda$-Termen in CPS wie in Tabelle 4 dargestellt aus. In [2] werden neben dieser auch CPS-Transformationen gemäß anderer Auswertungsstrategien diskutiert.
Nach der Übersetzung nimmt jeder Term einen zusätzlichen Parameter - seine Fortsetzung. Im Falle von Variablen wird die Fortsetzung an die von der Variablen referierte Abstraktion weitergeleitet. Erfreulicherweise ist die resultierende Abstraktion $\eta$-äquivalent zum ursprünglichen Term.
Weil wir durch unsere Auswertungsstrategie Reduktion im Rumpf von Abstraktionen verbieten, sind Abstraktionen Objekte, die für sich alleine genommen nicht weiter ausgewertet werden können. Daher wird bei der Übersetzung von

Tabelle 4: Übersetzung von $\lambda$-Termen in CPS

$$\begin{array}{rcl} [\![x]\!] & = & \lambda k.xk \stackrel{\eta}{=} x \\ [\![\lambda x.M]\!] & = & \lambda k.k(\lambda x.[\![M]\!]) \\ [\![M\,N]\!] & = & \lambda k.[\![M]\!](\lambda v.v[\![N]\!]k) \end{array}$$

Abstraktionen der übersetzte Rumpf sofort an die eigene Fortsetzung übergeben, die, gegeben ein Argument, die Applikation durchführen muss.
Entsprechend sieht die Übersetzung von Applikationen aus. Sie übergibt der Übersetzung des Funktors eine Fortsetzung, die den ausgewerteten Funktor auf die Übersetzung des Arguments appliziert. Da die Auswertung der Applikation somit beendet ist, wird an das Applikationsergebnis die Fortsetzung des Gesamtterms übergeben.
Ohne einen formalen Beweis für die Korrektheit unserer CPS-Transformation angeben zu wollen, verdeutlichen wir uns ihre Funktionsweise an einem Beispiel. Dazu betrachten wir die Reduktion

$$(\lambda x.x)y \to y$$

und zeigen, wie diese durch eine entsprechende Reduktionskette für die CPS-Darstellungen beider Terme simuliert wird:

$$\begin{array}{rcll} [\![(\lambda x.x)y]\!]k & = & [\![\lambda x.x]\!](\lambda v.v[\![y]\!]k) & \\ & = & (\lambda w.w(\lambda x.[\![x]\!]))(\lambda v.v[\![y]\!]k) & \\ & = & (\lambda w.w(\lambda x.\lambda u.xu))(\lambda v.v(\lambda u.yu)k) & (a) \\ & \to & (\lambda v.v(\lambda u.yu)k)(\lambda x.\lambda u.xu) & (b) \\ & \to & (\lambda x.\lambda u.xu)(\lambda u.yu)k & (c) \\ & \to & (\lambda u.(\lambda u.yu)u)k & \\ & \to & (\lambda u.yu)k & \\ & = & [\![y]\!]k & \end{array}$$

Wir beobachten

$(a) \to (b)$ : die Übergabe der übersetzten Abstraktion $\lambda x.x$ an ihre Fortsetzung,

$(b) \to (c)$ : die Applikation der Abstraktion auf ihr Argument durch die Fortsetzung.

## 3.3 Übersetzung in FO$\pi$

Die Übersetzung von CPS-Termen in FO$\pi$ erweist sich als relativ einfach. Die Grundidee ist dabei, Variablen durch Namen und $\lambda$-Abstraktionen durch $\pi$-Abstraktionen zu ersetzen. Da in FO$\pi$ Applikationen ausschließlich Namen als Argumente zulassen, setzen wir bei $\lambda$-Applikationen auf komplexe Terme die uns aus HO$\pi$ bekannte Übersetzungstechnik ein. Unsere Überlegungen fassen wir in der folgenden informellen Darstellung zusammen:

| **CPS** | | **FO$\pi$** | |
|---|---|---|---|
| $x$ | $\rightsquigarrow$ | $x$ | Variablen |
| $\lambda x.M$ | $\rightsquigarrow$ | $(x).M$ | Abstraktion |
| $Mx$ | $\rightsquigarrow$ | $M\langle x\rangle$ | Applikation auf Variablen |
| $M(\lambda x.N)$ | $\rightsquigarrow$ | $\mathrm{new}\, g\,(M\langle g\rangle \mid !g(x).N)$ | Applikation auf Abstraktionen |

Formal definieren wir die Übersetzung mittels einer Funktion $[\_]$, welche zu einem $\lambda$-Term in CPS den entsprechenden FO$\pi$-Term liefert:

$$\begin{array}{rcll} [\lambda k.xk] & = & (k).\overline{x}\langle k\rangle & \\ [\lambda k.k(\lambda x.M)] & = & [\lambda k.k(\lambda xv.Mv)] & \eta \\ & = & (k).k(xv).[M]\langle v\rangle & \\ [\lambda k.M(\lambda v.vNk)] & = & (k).\mathrm{new}\, v\, ([M]\langle v\rangle \mid \mathrm{new}\, x\, (\overline{v}\langle xk\rangle \mid\, !x[N])) & \end{array}$$

Die übersetzung von Variablen in der ersten Zeile erfolgt gemäß unserer Vorüberlegung. Im Fall von Abstraktionen bringen wir den zu übersetzenden Term zunächst in eine etwas andere, $\eta$-äquivalente Form. Dies hat eher ästhetische Gründe und soll uns nicht weiter beschäftigen. Interessanter ist die Übersetzung selbst. In CPS erwartet der Term eine Fortsetzung $k$, deren Aufgabe es ist, bei Bedarf den Rumpf $M$ der Abstraktion mit den Parametern $x$ und $v$ aufzurufen. Im $\pi$-Kalkül erfolgt der Aufruf des Abstraktionsrumpfes wenn $x$ und $v$ über den Kanal $k$ gebunden werden. Der Bindungsmechanismus der $\beta$-Regel wird in diesem Fall durch die im $\pi$-Kalkül gebräuchliche Kommunikation von Namen ersetzt. Dementsprechend wird bei der Übersetzung der Applikation der für die Parametrisierung des Funktors verantwortliche Teilterm $vNk$ ersetzt durch den Ausgabe-Prozess $\mathrm{new}\, x\, (\overline{v}\langle xk\rangle \mid\, !x[N])$. Die Kommunikation des komplexen Arguments $N$ wird wie gewohnt durch die Übermittlung eines assoziierten Kanalnamens $x$ simuliert.
Tabelle 5 fasst die Überführung in CPS und die anschließende Übersetzung in FO$\pi$ in einem Schritt zusammen. Dabei stellen wir fest, dass die $\eta$-Regel des $\lambda$-Kalküls mit gewissen Einschränkungen auch auf $\pi$-Abstraktionen anwendbar ist:

$$(v).((k).\overline{x}\langle k\rangle)\langle v\rangle = (v).\overline{x}\langle v\rangle$$

Diese Beobachtung soll im Folgenden als $\eta'$ bezeichnet werden.

Tabelle 5: Übersetzung von $\lambda$-Termen in FO$\pi$

$$\begin{array}{rcll} [\![x]\!] & = & (k).\overline{x}\langle k\rangle \quad (\overset{\eta'}{=} \quad \overline{x}) & (5.1) \\ [\![\lambda x.M]\!] & = & (k).k(xv).[\![M]\!]\langle v\rangle & (5.2) \\ [\![M\ N]\!] & = & (k).\mathrm{new}\, v\, ([\![M]\!]\langle v\rangle \mid \mathrm{new}\, x\, (\overline{v}\langle xk\rangle \mid\, !x[\![N]\!])) & (5.3) \end{array}$$

## 3.4 Übersetzung am Beispiel

Bevor wir die Korrektheit unserer Übersetzung formal beweisen, probieren wir sie an einem Beispiel aus. Dazu betrachten wir den Term $(\lambda x.x)N$. Um die Übersetzung besser nachzuvollziehen, gehen wir schrittweise vor und beginnen mit dem Teilterm $\lambda x.x$:

$$\begin{array}{rcll} [\![(\lambda x.x)]\!] & = & (v).v(xw).[\![x]\!]\langle w\rangle & (5.2) \\ & = & (v).v(xw).((k).\overline{x}\langle k\rangle)\langle w\rangle & (5.1) \\ & = & (v).v(xw).\overline{x}\langle w\rangle & \eta' \end{array}$$

Applizieren wir die Übersetzung auf eine Fortsetzung, so bleibt nur noch der Abstraktionsrumpf stehen:

$$[\![\lambda x.x]\!]\langle v\rangle \quad = \quad v(xw).\overline{x}\langle w\rangle \qquad (*)$$

Für den Gesamtterm gilt demnach

$$\begin{array}{rcll} [\![(\lambda x.x)N]\!] & = & (u).\mathrm{new}\, v\, ([\![\lambda x.x]\!]\langle v\rangle \mid \mathrm{new}\, x\, (\overline{v}\langle xu\rangle \mid\, !x[\![N]\!])) & (5.3) \\ & = & (u).\mathrm{new}\, v\, (v(xw).\overline{x}\langle w\rangle \mid \mathrm{new}\, x\, (\overline{v}\langle xu\rangle \mid\, !x[\![N]\!])) & (*) \end{array}$$

Nun wissen wir, dass gemäß unserer Auswertungsstrategie im $\lambda$-Kalkül die folgende Reduktion möglich ist:

$$(\lambda x.x)N \to N$$

Offensichtlich ist es sinnvoll zu fordern, dass die Reduktion auch nach der Übersetzung beider Terme in den $\pi$-Kalkül nachvollzogen werden kann:

$$[\![(\lambda x.x)N]\!]\langle u\rangle \to^{*}\sim [\![N]\!]\langle u\rangle$$

Und tatsächlich gilt

$$\begin{array}{rll} & [\![(\lambda x.x)N]\!]\langle u\rangle & \\ = & \mathrm{new}\, v\, (v(xw).\overline{x}\langle w\rangle \mid \mathrm{new}\, x\, (\overline{v}\langle xu\rangle \mid\, !x[\![N]\!])) & \\ \equiv & \mathrm{new}\, vx\, (v(xw).\overline{x}\langle w\rangle \mid \overline{v}\langle xu\rangle \mid\, !x[\![N]\!]) & x \notin FV(v(xw).\overline{x}\langle w\rangle) \\ \to & \mathrm{new}\, vx\, (\overline{x}\langle u\rangle \mid\, !x[\![N]\!]) & \\ \equiv & \mathrm{new}\, vx\, (\overline{x}\langle u\rangle \mid x[\![N]\!] \mid\, !x[\![N]\!]) & !P \equiv P \mid\, !P \\ \to & \mathrm{new}\, vx\, ([\![N]\!]\langle u\rangle \mid\, !x[\![N]\!]) & \\ \equiv & [\![N]\!]\langle u\rangle \mid \mathrm{new}\, vx\, (!x[\![N]\!]) & \text{o.B.d.A. } v, x \notin FV([\![N]\!]) \\ \sim & [\![N]\!]\langle u\rangle & \mathrm{new}\, vx\, (!x[\![N]\!]) \sim 0 \end{array}$$

## 3.5 Korrektheit der Übersetzung

Wir sehen, dass sich die Übersetzung an unserem Beispiel erwartungsgemäß verhält. Um ihre Korrektheit zu zeigen müssen wir uns aber zunächst fragen, was man im Falle einer Übersetzungsfunktion unter dem Korrektheitsbegriff überhaupt verstehen soll. Offensichtlich wollen wir weder im $\lambda$-Kalkül noch im $\pi$-Kalkül alle Terme als äquivalent ansehen. Also können wir die beiden Termmengen in Äquivalenzklassen partitionieren. Wir wählen die Äquivalenzklassen in beiden Fällen so, dass sie unter den jeweiligen Berechnungsoperationen abgeschlossen sind. Im $\lambda$-Kalkül sollen die Äquivalenzklassen daher unter $\beta$-Reduktion, im $\pi$-Kalkül unter Reaktion abgeschlossen sein. Eine korrekte Übersetzung muss zwei äquivalente $\lambda$-Terme auf zwei äquivalente $\pi$-Terme abbilden. Terme, die nicht äquivalent sind, müssen nach ihrer Übersetzung korrekterweise unterschiedlichen Äquivalenzklassen zugehören.
Wann sehen wir zwei Terme als unterschiedlich an? Da für uns allein der Berechnungsprozess interessant ist, muss es zu zwei unterschiedlichen reduzierten Termen Kontexte geben, in denen sich ihr Berechnungsverhalten unterscheidet. Im $\lambda$-Kalkül handelt es sich bei diesem Verhalten um die Kombinierbarkeit eines Terms mit anderen zu $\beta$-Redexen. Im einfachsten Fall ergibt der eine Term eingesetzt in einen Kontext einen $\beta$-Redex und der andere nicht. So sind etwa die Terme $\lambda x.x$ und $g$ unterschiedlich, weil $(\lambda x.x)z$ ein $\beta$-Redex ist und $gz$ nicht. Wir schreiben $(\lambda x.x) \neq_e g$. Andere Fälle lassen sich durch Ausnutzung der Abgeschlossenheit von Äquivalenzklassen unter $\beta$-Reduktion auf den einfachen Fall zurückführen. Zum Beispiel können wir aus $\lambda x.x \neq_e f$ leicht $g \neq_e f$ herleiten:

$$\begin{array}{rrcl} & \lambda x.x & \neq_e & f \\ \overset{\beta}{\Longleftrightarrow} & (\lambda g.g)(\lambda x.x) & \neq_e & (\lambda g.f)(\lambda x.x) \\ \Longrightarrow & g & \neq_e & f \end{array}$$

Die Einsetzung eines Terms in einen Kontext entspricht im $\pi$-Kalkül einem Experiment. Die von uns soeben eingeführte Äquivalenzrelation auf $\lambda$-Termen ist somit gerade die Beobachtungsäquivalenz des $\pi$-Kalküls. Bei unserem Korrektheitsbeweis wollen wir zeigen, dass

$$M =_e N \iff [\![M]\!] \approx [\![N]\!]$$

eine gültige Aussage ist. Dazu reicht es zu zeigen, dass die Abgeschlossenheit der Äquivalenzklassen unter Berechnung von der Übersetzungsfunktion respektiert wird, d.h.

$$[\![(\lambda x.M)N]\!] \approx [\![\{^N/_x\}M]\!]$$

Ist dies gezeigt, so können wir jeden Äquivalenzbeweis im $\lambda$-Kalkül in den $\pi$-Kalkül übertragen, indem wir die Terme durch ihre Übersetzungen und die Relation $=_e$ durch die schwache Äquivalenz ersetzen.
Für den Beweis benötigen wir noch ein einfach zu zeigendes

**Lemma** $x \in FV(M) \iff x \in FV([\![M]\!])$ (L1)

**Beweis** erfolgt über die Struktur von M. Der Beweis ist einfach, weil die Regel (5.1) die einzige ist, bei der auf beiden Seiten eine freie $\lambda$-Variable bzw. ein freier Name eingeführt werden.

Zu vermerken ist die überladene Notation, die wir im Lemma bei $x$ und $FV$ verwenden. Links steht $x$ für eine $\lambda$-Variable und $FV$ liefert die Menge der freien Variablen in $M$ gemäß den Bindungsprinzipien des $\lambda$-Kalküls. Rechts steht $x$ für einen Namen und $FV$ bezieht sich auf die freien Namen eines FO$\pi$-Terms.

**Satz** $[\![(\lambda x.M)N]\!] \approx [\![\{^N/_x\}M]\!]$

**Beweis** Ohne Beschränkung der Allgemeinheit soll $x \notin FV(N)$ gelten. Nach den Übersetzungsregeln gilt

$$\begin{array}{lcll} [\![(\lambda x.M)N]\!]\langle u\rangle & = & \mathrm{new}\, v\,(v(xw).[\![M]\!]\langle w\rangle \mid \mathrm{new}\, x\,(\overline{v}\langle xu\rangle \mid\, !x[\![N]\!])) & \\ & \to & \mathrm{new}\, vx\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) & \\ & \equiv & \mathrm{new}\, x\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) & \text{o.B.d.A. } v \text{ frisch} \end{array}$$

Wir folgern

$$\begin{array}{lcll} [\![(\lambda x.M)N]\!]\langle u\rangle & \sim & \tau.\mathrm{new}\, x\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) & \\ & \approx & \mathrm{new}\, x\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) & \tau.P \approx P \end{array}$$

Es bleibt also zu zeigen

$$\mathrm{new}\, x\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) \approx [\![\{^N/_x\}M]\!]\langle u\rangle$$

Diese Aussage beweisen wir durch Induktion über die Struktur von $M$.

1. $M = x$

$$\begin{array}{lcll} \mathrm{new}\, x\,([\![M]\!]\langle u\rangle \mid\, !x[\![N]\!]) & = & \mathrm{new}\, x\,(\overline{x}\langle u\rangle \mid\, !x[\![N]\!]) & (5.1), \eta' \\ & \sim & \tau.[\![N]\!]\langle u\rangle \mid \mathrm{new}\, x\,(!x[\![N]\!]) & x \notin FV(N), (\mathrm{L1}) \\ & \sim & \tau.[\![N]\!]\langle u\rangle & \mathrm{new}\, x\,(!xF) \sim 0 \\ & \approx & [\![N]\!]\langle u\rangle & \tau.P \approx P \\ & = & [\![\{^N/_x\}M]\!]\langle u\rangle & \end{array}$$

2. $M = y \neq x$

$$\begin{array}{rcll} \operatorname{new} x\,(\llbracket M\rrbracket\langle u\rangle \mid !x\llbracket N\rrbracket) & = & \operatorname{new} x\,(\overline{y}\langle u\rangle \mid !x\llbracket N\rrbracket) & (5.1), \eta' \\ & \sim & \overline{y}\langle u\rangle \mid \operatorname{new} x\,(!x\llbracket N\rrbracket) & \\ & \sim & \overline{y}\langle u\rangle & \operatorname{new} x\,(!xF) \sim 0 \\ & = & \llbracket y\rrbracket\langle u\rangle & (5.1), \eta' \\ & = & \llbracket \{^N/_x\}M\rrbracket\langle u\rangle & \end{array}$$

3. $M = \lambda y.M'$

$$\begin{array}{rcll} \operatorname{new} x\,(\llbracket M\rrbracket\langle u\rangle \mid !x\llbracket N\rrbracket) & = & \operatorname{new} x\,(u(yv).\llbracket M'\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & (5.2) \\ & \sim & u(yv).\operatorname{new} x\,(\llbracket M'\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & \\ & \approx & u(yv).\llbracket \{^N/_x\}M'\rrbracket\langle v\rangle & \text{IA} \\ & = & \llbracket \lambda y.\{^N/_x\}M'\rrbracket\langle u\rangle & (5.2) \\ & = & \llbracket \{^N/_x\}M\rrbracket\langle u\rangle & \end{array}$$

4. $M = M_1 M_2$
Für diesen Fall benötigen wir einen Hilfssatz, der in [1] gezeigt wird.

**Lemma** Seien $P$, $P_1$, $P_2$, $F$ negativ auf $x$, d.h. alle freien Vorkommen von $x$ in $P$, $P_1$, $P_2$, $F$ seien der Form $\overline{x}C$. Dann gilt:

$$\begin{array}{rcll} \operatorname{new} x\,(P_1 \mid P_2 \mid !xF) & \sim & \operatorname{new} x\,(P_1 \mid !xF) \mid \operatorname{new} x\,(P_2 \mid !xF) & \text{(L2)} \\ \operatorname{new} x\,(!P \mid !xF) & \sim & !\operatorname{new} x\,(P \mid !xF) & \text{(L3)} \end{array}$$

Ferner verwenden wir die folgende einfach zu zeigende Äquivalenz:

$$\operatorname{new} x\,(yF \mid !xG) \quad \sim \quad y(w).\operatorname{new} x\,(F\langle w\rangle \mid !xG) \qquad (*)$$

Nun zum eigentlichen Beweis. Auch hier nehmen wir ohne Beschränkung der Allgemeinheit an, dass die nach der Regel (5.3) durch Restriktion eingeführte Namen frisch sind.

$$\begin{array}{rcll} & \operatorname{new} x\,(\llbracket M\rrbracket\langle u\rangle \mid !x\llbracket N\rrbracket) & & \\ = & \operatorname{new} x\,(\operatorname{new} v\,(\llbracket M_1\rrbracket\langle v\rangle \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !y\llbracket M_2\rrbracket)) \mid !x\llbracket N\rrbracket) & & (5.3) \\ \equiv & \operatorname{new} v\,(\operatorname{new} x\,(\llbracket M_1\rrbracket\langle v\rangle \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !y\llbracket M_2\rrbracket) \mid !x\llbracket N\rrbracket)) & & v \text{ frisch} \\ \sim & \operatorname{new} v\,(\operatorname{new} x\,(\llbracket M_1\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & & \\ & \quad \mid \operatorname{new} x\,(\operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !y\llbracket M_2\rrbracket) \mid !x\llbracket N\rrbracket)) & & \text{(L2)} \\ \equiv & \operatorname{new} v\,(\operatorname{new} x\,(\llbracket M_1\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & & \\ & \quad \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid \operatorname{new} x\,(!y\llbracket M_2\rrbracket \mid !x\llbracket N\rrbracket))) & & y \text{ frisch} \\ \sim & \operatorname{new} v\,(\operatorname{new} x\,(\llbracket M_1\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & & \\ & \quad \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !\operatorname{new} x\,(y\llbracket M_2\rrbracket \mid !x\llbracket N\rrbracket))) & & \text{(L3)} \\ \sim & \operatorname{new} v\,(\operatorname{new} x\,(\llbracket M_1\rrbracket\langle v\rangle \mid !x\llbracket N\rrbracket) & & \\ & \quad \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !y(w).\operatorname{new} x\,(\llbracket M_2\rrbracket\langle w\rangle \mid !x\llbracket N\rrbracket))) & & (*) \\ \approx & \operatorname{new} v\,(\llbracket \{^N/_x\}M_1\rrbracket\langle v\rangle & & \\ & \quad \mid \operatorname{new} y\,(\overline{v}\langle yu\rangle \mid !y\llbracket \{^N/_x\}M_2\rrbracket)) & & \text{IA} \\ = & \llbracket \{^N/_x\}M_1\ \{^N/_x\}M_2)\rrbracket\langle u\rangle & & (5.3) \\ = & \llbracket \{^N/_x\}M\rrbracket\langle u\rangle & & \end{array}$$

□

### 3.6 Sortendisziplin der Übersetzung

Nachdem wir die Korrektheit unserer Übersetzungsfunktion und damit die universelle Berechnungskraft des $\pi$-Kalküls bewiesen haben, fragen wir uns, ob ein FO$\pi$-Term, der durch die Übersetzung aus einem gegebenen $\lambda$-Term entsteht, grundsätzlich eine Sortierung respektiert. Auf der einen Seite ist bekannt, dass Terme des ungetypten $\lambda$-Kalküls im Allgemeinen keine Typdisziplin respektieren. Auf der anderen Seite kennen wir keine sinnvollen Beispiele von nicht typbaren $\pi$-Termen. Zu jedem nicht typbaren Term können wir leicht einen stark äquivalenten typbaren Term konstruieren. Daher vermuten wir, dass auch unsere Übersetzung nur typbare Terme produziert. Um diese Vermutung zu bestätigen genügt ein aufmerksamer Blick auf die Übersetzungsfunktion in Tabelle 5. Da eine Sortierung lediglich eine Aussage über die Verwendung von Namen als Kanäle macht, reicht es gewisse Teilterme zu betrachten. Zunächst bemerken wir, dass jeder $\lambda$-Term in eine Abstraktion übersetzt wird, welche die Sorte von $k$ besitzt. Aus der Verwendung $[\![M]\!]\langle v\rangle$ schließen wir dann, dass $v$ die gleiche Sorte wie $k$ haben muss. Die beiden Terme $\overline{x}\langle k\rangle$ und $x[\![N]\!]$ machen klar, dass $x$ ein Kanal ist, der Nachrichten der Sorte von $k$ übermittelt. Diese Sorte können wir näher bestimmen, indem wir die Verwendung von $k$ und $v$ in $k(xv)$ und in $\overline{v}\langle xk\rangle$ analysieren.
Wählen wir für die Sorte von $k$ den Namen FUN, so gilt nach unseren Überlegungen $k, v : \mathrm{FUN}$, $x : \mathrm{CHAN}(\mathrm{FUN})$. Die von unserer Übersetzung respektierte Sortierung sieht dann wie folgt aus:

$$\begin{aligned} \mathrm{CHAN}\ s &\mapsto s \\ \mathrm{FUN} &\mapsto \mathrm{CHAN}(\mathrm{FUN}),\ \mathrm{FUN} \end{aligned}$$

Bei der ersten Regel handelt es sich um einen Spezialfall der Definition des Sortenkonstruktors CHAN. Da wir CHAN ausschließlich mit dem Argument FUN verwenden, könnten wir für $x$ alternativ einen neuen Sortennamen einführen und so auf die Verwendung polymorpher Sortenkonstruktoren verzichten.
Die Tatsache, dass wir für die Übersetzung ungetypter $\lambda$-Terme in FO$\pi$ eine Sortierung finden konnten, ist bei näherer Betrachtung wenig überraschend. Denn es ist wohlbekannt, dass bei der Auswertung wohlgeformter $\lambda$-Terme niemals Fehler auftreten können. Für die CPS-Darstellung des Kontrollflusses, die der Übersetzung zugrundeliegt, bedeutet dies, dass wir bei der Auswertung immer einen gültigen CPS-Ergebnisterm erreichen müssen. Insbesondere erwarten wir eine konsistente Verwendung freier Variablen, was eine entscheidende Voraussetzung für die Sortierbarkeit des resultierenden $\pi$-Terms ist. Einen formalen Bewes für die Einhaltung der angegebenden Sortierung durch die Übersetzung findet man in [1].

# 4 Zusammenfassung und Ausblick

Die Motivation hinter der Entwicklung des $\pi$-Kalküls war, eine formale Grundlage für die Analyse paralleler Prozesse und ihres Interaktionsverhaltens vor dem Hintergrund einer frei änderbaren Verbindungsstruktur zu schaffen. Zum heutigen Zeitpunkt existieren keine anderen Formalismen zur Modellierung kommunizierender Systeme, deren formale Zugänglichkeit der des $\pi$-Kalküls nahekommt.

Da, wie wir gesehen haben, der $\pi$-Kalkül Turing-vollständig ist, kann er prinzipiell zur Beschreibung und zur Analyse beliebig komplexer Systeme eingesetzt werden.
In der Praxis aber stehen seinem Einsatz gewisse Unzulänglichkeiten entgegen. So fehlen dem Basiskalkül jegliche Abstraktions- und Strukturierungsmöglichkeiten, die ein unverzichtbares Hilfsmittel bei der Analyse größerer Systeme sind. Der Kalkül verfügt weder über prozedurale noch über modulare Konzepte, was dazu führt, dass schon bei relativ einfachen Anwendungen die entsprechenden Terme kaum noch zu verstehen sind.
Will man alle Berechnungsprozesse mit den Mitteln von FO$\pi$ beschreiben, sieht man schnell eine weitere Unzulänglichkeit des Kalküls. Während die Kommunikation von Namen einige Prozesse auf eine intuitive Weise modellieren können, gibt es Anwendungsfelder, bei denen spezielle Formalismen wesentlich kompaktere und verständlichere Modelle liefern.
Um komplexe Anwendungen zu erschließen, müssen die aufgezeigten Schwierigkeiten bei der Verwendung des $\pi$-Kalküls überwunden, seine Vorteile mit denen höherstufiger, modularer Formalismen verbunden werden. Wie wir im Rahmen dieser Ausarbeitung gesehen haben, sind wir in der Lage, formale Zusammenhänge zwischen dem $\pi$-Kalkül und anderen, teilweise vollkommen anders strukturierten Kalkülen herzustellen. Damit haben wir die Möglichkeit, abstraktere, modular aufgebaute Prozessbeschreibungssprachen zu entwickeln, die trotzdem detaillierter Analyse durch formale Techniken des $\pi$-Kalküls zugänglich sind. Das wichtigste Beispiel für einen solchen Ansatz ist die von Benjamin Pierce und David Turner entwickelte, auf Konzepten von HO$\pi$ aufgebaute Programmiersprache Pict (siehe [5]).

# Literatur


[1] R. Milner, Communicating and Mobile Systems: the $\pi$-Calculus, Cambridge University Press, 1999.

[2] D. Sangiorgi and D. Walker, The $\pi$-calculus: A Theory of Mobile Processes, Cambridge University Press, 2001.

[3] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol 92, Springer-Verlag, 1980.

[4] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[5] B. Pierce and D. Turner, Pict: A programming language based on the Pi-calculus, Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in Proof, Language and Interaction: Essays in Honour of Robin Milner, eds. G. Plotkin, C. Stirling and M. Tofte, MIT Press, 1998.