Integration von Polymorphismus und Subtypen für den π-Kalkül

Dirk Draheim

Institut für Informatik, Freie Universität Berlin draheim@inf.fu-berlin.de

Zusammenfassung In diesem Papier werden π-Kalkül-Typsysteme mit Polymorphismus einerseits [13] und Subtypen andererseits [8] zu einem neuen π-Kalkül mit beschränktem Polymorphismus integriert. Als motivierendes Beispiel wird die natürliche Darstellung einer generischen Compute-Engine im resultierenden Kalkül präsentiert. Technische Beiträge sind zum einen Beweise der notwendigen Eigenschaften des Typsystems, das heißt ein Subjekt-Reduktionstheorem und Theoreme zur Typ-Stärkung und -Schwächung. Außerdem beweisen wir korrektes Laufzeitverhalten wohlgetypter Prozesse.

Keywords: π-Kalkül, σ-Kalkül, Beweistechniken

1 Einleitung

In diesem Papier möchten wir Fortschritte unserer laufenden Arbeit präsentieren, das σ-Kalkül zweiter Stufe in einem geeigneten beschränkt polymorphen π-Kalkül zu interpretieren.

Der π-Kalkül [6] ist eine Prozeßalgebra, die das Konzept der nachrichtenbasierten Kooperation mittels Kommunikation von Kanalnamen formalisiert. Er erlaubt die Beschreibung von Prozeß-Systemen, in denen sich die Kommunikationstopologie dynamisch ändert. Der π-Kalkül eröffnete eine neue Perspektive der Untersuchung von Konzepten objektorientierter Programmiersprachen [15]. Der σ-Kalkül [1] ist als minimale objektbasierte Programmiersprache mit Objekt-Formation, Methoden-Aufruf und Methoden-Update aufzufassen.

Fokus unserer Arbeit ist der Ansatz, den σ-Kalkül im π-Kalkül zu interpretieren. Dieser Ansatz ist für Typsysteme erster Ordnung bereits gründlich ausgearbeitet. Die Motivation liegt im wechselseitig vertieften Verständnis von Objekten und Prozessen einerseits [11], in der Erschließung neuer Beweistechniken andererseits [5]. Vor diesem Hintergrund gilt unser Interesse nun fortgeschrittenen Typsystemen. Die bestehenden Interpretationen sollen konsequent bzgl. Polymorphismus fortgesetzt werden. Die Schwierigkeit ist dabei, daß sich im Kontext von Prozessen existenzquantifizierte Typen als die natürliche Form von Polymorphismus erwiesen haben [12], während Generizität in objektbasierten Sprachen hingegen durch allquantifizierte Typen modelliert wird, in voller Analogie zu parametrischem Polymorphismus funktionaler Sprachen [9].

In Kapitel 2 führen wir den Kalkül $\pi_{\mu,\times,+,:<,\exists:<}$ ein, einen π-Kalkül mit rekursiven Typen, Produkten, Verbundtypen, Subtypen und beschränktem existenzquantifizierten Polymorphismus. Es folgen als Beispiel eine generische Compute-Engine und die Formulierung von Eigenschaften des Typsystems in den Kapiteln 3 und 4. Abschließend wird mit Kapitel 5 die Relevanz der Ergebnisse für die Deutung von Objekten in Prozeßalgebra aufgezeigt.

2 Der Kalkül $\pi_{\mu,\times,+,:<,\exists:<}$

Wir folgen [12] und präsentieren den Kalkül als Erweiterung von CCS-ähnlicher Prozeßalgebra um die Möglichkeit, Kanalnamen zu kommunizieren. Wesentliche Beiträge von Prozeßalgebra sind die Formalisierung des Konzepts nachrichtenbasierter Kommunikation einerseits und die Behandlung von Prozessen als Objekte potentiell unendlichen Verhaltens andererseits. Im π-Kalkül findet sich letzterer Aspekt im Replikationsoperator `bang`. Der π-Kalkül ermöglicht zusätzlich die Modellierung von Systemen mit sich dynamisch ändernder Kommunikationstopologie.

Wir wiederholen zunächst das π-Kalkül mit geeignetem Typsystem für Kanaltypen, Produkten und Verbundtypen und führen anschließend Subtypen und Polymorphismus ein.

2.1 Der Kalkül $\pi_{\mu,\times,+}$ als erweitertes CCS

Als Metavariablen für Kanalnamen werden v, w, x verwendet, in den sich anschließenden Beispielen auch andere Kleinbuchstaben. Außer den Kanalnamen gibt es weitere Basiswerte, die später zusammen mit dem Typsystem eingeführt werden. Prozesse stehen für Aktionssequenzen. Mögliche Aktionen sind Eingabe, Ausgabe und stille Transition. Prozesse werden aus inaktivem Prozeß, Abfolge, Nichtdeterminismus, Nebenläufigkeit, Kanalerzeugung und Vergleich, sowie der Replikation gebildet. Der Prozeß *wrong* repräsentiert einen Laufzeitfehler.

$$
\begin{array}{rcll}
P, Q, R & ::= & 0 & \text{inaktiver Prozeß} \\
& | & \pi.P & \text{Präfix, Abfolge} \\
& | & P + Q & \text{Summe, Nichtdeterminismus} \\
& | & P \mid Q & \text{Komposition, Nebenläufigkeit} \\
& | & (\nu x : T)P & \text{Restriktion, Kanalerzeugung} \\
& | & [x = y]P & \text{Vergleich} \\
& | & !P & \text{bang, Replikation} \\
& | & \mathit{wrong} & \text{Fehler} \\
\\
\pi & ::= & v(x) & \text{Eingabe} \\
& | & \overline{v}w & \text{Ausgabe} \\
& | & \tau & \text{stille Transition}
\end{array}
$$

Die Semantik des Kalküls ist standardmäßig in geeigneter Form als kantenmarkiertes Transitionssystem gegeben.

Als Metavariablen für Typen werden S, T verwendet. Es gibt Werttypen V, Kanaltypen L und den Verhaltenstyp $\lozenge$.

$$\begin{aligned} S, T &::= V \mid L \mid \lozenge \\ V &::= \ \mathit{unit} \mid \mathbb{B} \mid \mathbb{Q} \mid \mathbb{Z} \mid \mathbb{N} \mid \mathbb{N}_0 \mid \ \ldots \end{aligned}$$

Das Typsystem wird als Satz von Ableitungsregeln formalisert, basierend auf geeigneten Typumgebungen zur Typisierung freier Namen.

$$\Gamma \ ::= \emptyset \mid \Gamma, x : L \mid \Gamma, x : V$$

$$\text{TV-Name} \qquad \frac{}{\Gamma, x : T \vdash x : T}$$

Der π-Kalkül geht nun aus dem Basiskalkül durch Gleichsetzung von Wert- und Kanaltypen hervor.

$$V \ ::= \ L$$

Das Typsystem des π-Kalküls definiert die Wohlgetyptheit von Prozessen konstruktionsarm: Prozesse können ausschließlich den Verhaltenstyp besitzen.

$$\begin{array}{rl}
\text{T-Par} & \dfrac{\Gamma \vdash P : \lozenge \quad \Gamma \vdash Q : \lozenge}{\Gamma \vdash P | Q : \lozenge} \\
\text{T-Sum} & \dfrac{\Gamma \vdash P : \lozenge \quad \Gamma \vdash Q : \lozenge}{\Gamma \vdash P + Q : \lozenge} \\
\text{T-Nil} & \dfrac{}{\Gamma \vdash 0 : \lozenge} \\
\text{T-Rep} & \dfrac{\Gamma \vdash P : \lozenge}{\Gamma \vdash !P : \lozenge} \\
\text{T-Tau} & \dfrac{\Gamma \vdash P : \lozenge}{\Gamma \vdash \tau . P : \lozenge} \\
\text{T-Match} & \dfrac{\Gamma \vdash v : \# T \quad \Gamma \vdash w : \# T \quad \Gamma \vdash P : \lozenge}{\Gamma \vdash [v = w] P : \lozenge} \\
\text{T-Res} & \dfrac{\Gamma, x : L \vdash P : \lozenge}{\Gamma \vdash (\nu x : L) P : \lozenge} \\
\text{T-In} & \dfrac{\Gamma \vdash v : \# T \quad \Gamma, x : T \vdash P : \lozenge}{\Gamma \vdash v(x) . P : \lozenge} \\
\text{T-Out} & \dfrac{\Gamma \vdash v : \# T \quad \Gamma \vdash w : T \quad \Gamma \vdash P : \lozenge}{\Gamma \vdash \overline{v} w . P : \lozenge}
\end{array}$$

Für die eingeführten Basisdatentypen werden geeignete Basiswerte angenommen.

$$basval ::= \star \mid true \mid false \mid \ \ldots -1, 0, 1 \ldots \ \mid \ \ldots - \frac{1}{2} \ldots \frac{1}{2} \ldots \mid \ldots$$

$$\text{TV-Base} \qquad \frac{basval \in B}{\Gamma \vdash basval : B}$$

Für Produkte und variante Verbunde wird lediglich die Grammatik präsentiert. Die Typen sind jeweils kovariant in ihren Komponenten. Für variante Verbunde gilt außerdem die bekannte Subtypstruktur entlang längerer Typen.

$v ::= \ < v_1, \ldots, v_n >$	Produktkonstruktion
$P ::= \ with(x_1, \ldots, x_n) = v \ do \ P$	Produktdestruktion
$T ::= \ T_1 \times \ldots \times T_n$	Produkttypen
l, h	variante Marken
$v ::= \ l\_v$	variante Werte
$P ::= case \ v \ of[l_1\_(x_1) \to P_1; \ldots; l_n\_(x_n) \to P_n]$	pattern matching
$T ::= [l_1\_T_1, \ldots, l_n\_T_n]$	variante Typen

Für rekursive Typen wird alternativ zur syntaktischen Einführung von `fold` und `unfold` eine geeignete strukturelle Gleichheit $\sim_{type}$ definiert.

$$\begin{array}{lcl} V & ::= & X \mid \mu X.V \\ L & ::= & \mu X.L \end{array}$$

$$\text{EQ-Unfold} \qquad \frac{}{\mu X.T \sim_{type} T\{\mu X.T/X\}}$$

$$\text{T-EQ} \qquad \frac{\Gamma \vdash v \ : \ s \quad S \sim_{type} T}{\Gamma \vdash v \ : \ T}$$

2.2 Subtypen

Die Subtypstruktur des π-Kalküls basiert auf der Unterscheidung zwischen Eingabe- und Ausgabekanälen.

$$\begin{array}{rl} L ::= iV & \text{Typ für Eingabekanäle} \\ \mid \quad oV & \text{Typ für Ausgabekanäle} \end{array}$$

Die Subtypbeziehung wird in Vorgriff auf die Einführung von Polymorphie in Abhängigkeit von Typumgebungen definiert.

Reflexivität	$$\frac{}{\Gamma \vdash T \leq T}$$
Transitivität	$$\frac{\Gamma \vdash S \leq S' \quad \Gamma \vdash S' \leq T}{\Gamma \vdash S \leq T}$$
SUB-#In	$$\frac{}{\Gamma \vdash \#T \leq iT}$$
SUB-#Out	$$\frac{}{\Gamma \vdash \#T \leq oT}$$
SUB-In	$$\frac{\Gamma \vdash S \leq T}{\Gamma \vdash iS \leq iT}$$
SUB-Out	$$\frac{\Gamma \vdash T \leq S}{\Gamma \vdash oS \leq oT}$$
SUB-Both	$$\frac{\Gamma \vdash T \leq S \quad \Gamma \vdash S \leq T}{\Gamma \vdash \#S \leq \#T}$$

Die Regeln `T-In` und `T-Out` ersetzen gemeinschaftlich die bisherige Regel `T-In`.

T-In	$$\frac{\Gamma \vdash a : iS \quad \Gamma, x : S \vdash P : \Diamond}{\Gamma \vdash a(x).P : \Diamond}$$
T-Out	$$\frac{\Gamma \vdash a : oS \quad \Gamma \vdash w : S \quad \Gamma \vdash P : \Diamond}{\Gamma \vdash \overline{a}(w).P : \Diamond}$$
T-Subsumption	$$\frac{\Gamma \vdash v : S \quad S \leq T}{\Gamma \vdash v : T}$$

Der Ausgabekanaltyp ist kontravariant in seinem einzigen Argument, der Eingabekanaltyp ist kovariant. Zusammen mit der Regel `ST-Subsumption` ergibt sich, daß über einen Ausgabekanal Werte eines beliebigen Subtyps gesendet werden können. Da ein Subtyp immer ein speziellerer Typ in dem Sinn ist, daß seine Objekte mehr Möglichkeiten besitzen bzw. mehr Bedingungen erfüllen, entspricht dies der Intuition, daß der Typ eines Ausgabekanals für einen Prozeß eine Verpflichtung darstellt. Analog ergibt sich für Eingabekanäle, daß empfangene Werte an Namen beliebigen Obertyps gebunden werden können. Dies entspricht der Vorstellung, daß der Typ eines Eingabekanals für einen Prozeß eine Garantie darstellt. Eine weitere Rechtfertigung der genannten Typvarianzen ergibt sich bei der Deutung eines Paares aus Eingabe- und Ausgabekanal als Funktionsaufruf und der vollen Analogie der Varianzeigenschaften des entsprechenden Funktionstyps zu denen der entstehenden Prozeßkontextlücke.

Für die Basisdatentypen ist ebenfalls eine geeignete Subtypstruktur definiert. Es folgen beispielhafte Ableitungsregeln für die Zahltypen.

$$\frac{}{\Gamma \vdash \mathbb{N} \leq \mathbb{Q}}$$

$$\frac{}{\Gamma \vdash \mathbb{N}_0 \leq \mathbb{N}}$$

2.3 Polymorphismus

Ein existenzquantifizierter Typ ist die unendliche Summe von Objekten, die unter Verwendung eines Repräsentationstypen für einen ansonsten opaken Typen als Implementierung eines Schnittstellentypen gültig sind. Pakete aus Repräsentationstypen und Implementierungen bieten sich deswegen im Kontext von Programmiersprachen zur Modellierung von abstrakten Datenobjekten an. Insgesamt wird somit in [3] und [7] Mechansimen für lose abstrakte Datentypen Bedeutung gegeben. Im Kontext von Prozeßalgebra dienen existenzquantifizierte Typen hingegen natürlicherweise zur Formulierung von Prozessen, die homogen in Abhängigkeit von Typen Dienste erbringen und ermöglichen somit eine Form von parametrischem Polymorphismus. Zu diesem Zweck wird der Kalkül zunächst um das Konzept der beschränkten Typvariablen erweitert.

$$\Gamma ::= \Gamma, X \leq Y \qquad \Leftarrow X \notin dom\Gamma$$

$$\text{SUB-Var} \qquad \frac{}{\Gamma, X \leq Y, \Gamma' \vdash X \leq Y}$$

Werte eines existenzquantifizierten Typen sind Pakete aus Typ und Kanal. Die Pakete werden mit `open` geöffnet. Zur weiteren Destruktion im Körper des `open`-Konstrukts wird eine geeignete `with`-pattern-matching-Syntax angenommen, die in den Beispielen im nächsten Kapitel benötigt wird.

$$\begin{array}{rll} V & ::= \exists X \leq S.T & \text{polymorphe Typen} \\ v & ::= [T; v] & \text{Konstruktion} \\ P & ::= open\ v\ as(X \leq T; x) inP & \text{Destruktion} \end{array}$$

Typsystem und Reduktionssemantik werden wie folgt erweitert.

$$\text{T-Open} \qquad \frac{\Gamma \vdash v : \exists X \leq S.T \quad \Gamma, X \leq S, x{:}T \vdash P\ :\ \Diamond}{\Gamma \vdash open\ v\ (X \leq S; x)\ in\ P : \Diamond}$$

$$\text{T-Poly} \qquad \frac{\Gamma \vdash v\ :\ V(W/X)\Gamma \vdash W \leq S}{\Gamma \vdash [W; v] : \exists X \leq S.V}$$

$$\text{POLY} \qquad \frac{}{open[V; w]as(X \leq S, y)\ in\ P \xrightarrow{\tau} P\{V/X\}\{w/y\}}$$

$$\text{POLY-ERR} \qquad \frac{v\ does\ not\ have\ the\ form\ < V; w >}{open\ v\ as\ (X \leq S, y)\ in\ P \xrightarrow{\tau} wrong}$$

3 Eine generische Compute-Engine

Als Beispiel für die Ausdrücksmächtigkeit des resultierenden Kalküls modellieren wir die Berechnungen einer Compute-Engine. In [16] wird der Begriff der generischen Compute-Engine für einen Server geprägt, der Daten und Methoden als Objekt gekapselt erhält, um diese lokal zur Ausführung zu bringen. In diesem Szenario wird der objektorientierte Subtypmechanismus genutzt, um Minimalanforderungen an die übermittelten Tasks zu spezifizieren. Zur Einführung betrachten wir die Implementierung der polymorphen Identitätsfunktion.

$$ID \overset{DEF}{\equiv} a(t).open\ t\ as(X \leq \mathbb{Q})\ in\ (with(x,y) = t\ do\ \overline{x}y).0$$

Es folgt die Anwendung der Identitätsfunktion auf zwei Pakete, die Daten unterschiedlichen Typs transportieren.

$$\begin{array}{ll} & a : \#(\exists X \leq \mathbb{Q}.(X \times oX)\ ,\ b : o\mathbb{N}\ ,\ c : o\mathbb{Q} \\ \vdash & \\ & (\overline{a}[\mathbb{N}; < 3, b >]) \mid \overline{a}[\mathbb{Q}; < \frac{3}{4}, c >] \mid !ID) \\ \xrightarrow{\tau}{}^{*} & \\ & (\overline{b}3.0 \mid \overline{c}\frac{3}{4}.0 \mid !ID) \end{array}$$

Um in unserem Beispiel bessere Analogie zu dem Beispiel in [16] herstellen zu können bedarf es eigentlich eines π-Kalküls höherer Ordnung [10]. Wir begnügen uns aber mit Betrachtung der Verantwortlichkeit der Compute-Engine, unter der die als Dienst zu erbringende Berechnung durchgeführt wird.

$CE \overset{DEF}{\equiv} !a(t).open\ t\ as(X \leq \mathbb{Q})\ in\ (with(v,w,x,y) = t\ do\ \overline{w}v.x(z).\overline{y}z)$
wobei
$v \,\hat{=}$ Wert
$w \,\hat{=}$ Methodenauruf
$x \,\hat{=}$ Methodenausführung
$y \,\hat{=}$ Endergebnis

Es folgt die Anwendung der Compute-Engine auf zwei Tasks verschieden getypten Inhalts.

$$\begin{array}{ll} & a : \#[\exists X \leq \mathbb{Q}.X \times oX \times iX \times oX] \\ \vdash & \\ & (\nu b)(b(x).0 \mid (\nu m)(\nu n)(\overline{a}[\mathbb{N}; < 3, b, m, n >].m(x).\overline{n}x.0)) \\ & \mid (\nu c)(c(x).0 \mid (\nu m)(\nu n)(\overline{a}[\mathbb{N}; < \frac{3}{4}, c, m, n >].m(x).\overline{n}x.0)) \\ & \mid !CE \\ \xrightarrow{\tau}{}^{*} & \\ & (\nu b)(b(x).0 \mid \overline{b}3.0)) \mid (\nu c)(c(x).0 \mid \overline{c}\frac{3}{4}.0) \mid !CE) \\ \xrightarrow{\tau}{}^{*} & \\ & !CE \end{array}$$

Der Task besteht in beiden Fällen in Ausführung der Identitätsfunktion. Es ist zu beachten, wie durch die Verwendung von Restriktionen die Ausführung jeweils in die Verantwortlichkeit der Compute-Engine verbracht wird.

4 Theoreme für den $\pi_{\exists<}$-Kalkül

Wir formulieren nun die grundlegenden Eigenschaften des Typsystems. Sie werden in der fortlaufenden Arbeit, die im anschließenden Kapitel beschrieben wird, benötigt.

Lemma 41 (Korrektheit wohlgetypter Terme) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond$. *Dann folgt :* P *enthält* wrong *nicht als Subterm.* □

Lemma 42 (Schwächung der Umgebung) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \ \wedge \ x \notin dom\Gamma \ \wedge \ S \notin \Gamma$. *Dann folgt :* $\Gamma, x : S \vdash P : \Diamond$. □

Lemma 43 (Stärkung der Umgebung) *Vorausgesetzt :* $\Gamma, x : S \vdash P : \Diamond \ \wedge \ x \notin fn(P)$. *Dann folgt :* $\Gamma \vdash P : \Diamond$. □

Lemma 44 (Typverengung für Werte) *Vorausgesetzt :* $\Gamma, p : S \vdash w : U \ \wedge \ T \leq S$. *Dann folgt :* $\Gamma, p : T \vdash w : U$. □

Lemma 45 (Typverengung für Prozesse) *Vorausgesetzt :* $\Gamma, p : S \vdash P : \Diamond \ \wedge \ T \leq S$. *Dann folgt :* $\Gamma, p : T \vdash P : \Diamond$. □

Lemma 46 (Substitution) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \ \wedge \ \Gamma \vdash x : T \ \wedge \ \Gamma \vdash v : T$. *Dann folgt :* $\Gamma \vdash P\{x/v\} : \Diamond$.

Beweis: Dieses Lemma folgt mittels Analyse der möglichen Ableitungssequenzen aus Lemma 42 . □

Theorem 1 (Subjekt-Reduktion) *Vorausgesetzt :*

$$\Gamma \vdash P : \Diamond \ \wedge \ P \xrightarrow{\alpha}_E P'$$

Dann folgt :

1. $\alpha = \tau \Rightarrow \Gamma \vdash P' : \Diamond$
2. $\alpha = av \Rightarrow \exists T.$
   (a) $\Gamma \vdash a : iT$
   (b) $\Gamma \vdash vT \Rightarrow \Gamma \vdash P' : \Diamond$
3. $\alpha = (\nu \tilde{x} : \tilde{S})\overline{a}v \Rightarrow \exists T.$
   (a) $\Gamma \vdash a : oT$
   (b) $\Gamma, \tilde{x} : \tilde{S} \vdash v : T$
   (c) $\Gamma, \tilde{x} : \tilde{S} \vdash P' : \Diamond$
   (d) *jede Komponente von* $\tilde{S}$ *ist ein Kanaltyp*

Beweis: Der Beweis erfolgt durch strukturelle Induktion über die Länge der Mehrschritt-Transition bzgl. der gewählten operationellen Semantik mittels Lemma 42 bis Lemma 46. □

Korollar 1 (Laufzeit-Korrektheit) *Vorausgesetzt :* $\Gamma \vdash P : \Diamond \ \wedge \ P \xrightarrow{\alpha} P'$. *Dann folgt :* P' *enthält* wrong *nicht als Subterm.*

Beweis: Das Korollar folgt aus Lemma 41 und Theorem 1. □

5 Ausblick

Der σ-Kalkül [1] ist eine minimale objektbasierte Programmiersprache mit Objekt-Formation, Methoden-Aufruf und Methoden-Update. Er hat wesentlich zum Verständnis zentraler Konzepte objektorientierter Sprachen und deren Typsysteme, wie z.B. Selbstreferenz und kovariante Selbst-Typen, beigetragen.

Ziel der fortlaufenden Arbeit ist es, den σ-Kalkül zweiter Stufe im π-Kalkül zu interpretieren. Für den σ-Kalkül erster Stufe ist die Interpretation bereits erfolgreich ausgearbeitet worden. Sie benötigt als Zielsprache $\pi_{\mu,+,:<}$, den π-Kalkül mit rekursiven Typen, varianten Verbunden und Subtypen.

Die existierende Interpretation ist allerdings nicht vollständig abstrakt. Die laufende Arbeit versucht deshalb konsequent, Ergebnisse aus dem Bereich der Deutung von Objekten mittels prozeduraler Konzepte nutzbar zu machen [4] [2] [14].

Literatur

1. Martin Abadi, Luca Cardelli. A Theory of Objects. Monographs in Computer Science, Springer-Verlag, 1996
2. Martin Abadi, Luca Cardelli. An Interpretation of Objects and Object Types. Proceedings of POPL '96, ACM, 1996
3. Luca Cardelli, Peter Wegener. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol. 17, No. 4, December 1985
4. Samuel Kamin. Inheritance in smalltalk-80: A denotational definition. In: 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 80–87, January 1988.
5. Josva Kleist. Reasoning about Objects using Process Techniques, Ph.D. Thesis. Aalborg University, 2000
6. Robin Milner, Joachim Parrow, David Walker. A Calculus for Mobile Processes, Part I/II. Information and Computation 100, pp. 1-40/41-77, 1992
7. John C. Mitchell, Gordon D. Plotkin. Abstract Types have Existential Type. ACM Transaction on Programming Languages and Systems,vol. 10, no. 3, 1988
8. Benjamin Pierce, Davide Sangiorgi. Typing and Subtyping for Mobile Processes. In: Mathematical Structures in Computer Science, pp. 409-454, 1996
9. John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In: Information Processing 83, Elsevier Science Publishers, 1983
10. Davide Sangiorgi. From pi-calculus to Higher-order pi-calculus and back. In: TAPSOFT, Springer Verlag, LNCS 668, 1993.
11. Davide Sangiorgi. An Interpretation of Typed Objects into π-Calculus. In: Information and Computation, pp. 34-73, Academic Press, 1998
12. Davide Sangiorgi. David Walker. The π-calculus: a Theory of Mobile Processes. Cambridge University Press. June 2001, to appear
13. David N. Turner. The Polymorphic π-Calculus: Theory and Implementation. Ph.D. Thesis. University of Edinburgh. 1996
14. Ramesh Vismanathan. Full Abstraction for First-Order Objects with Recursive Types and Subtyping. In: Logic of Computer Science, 1998
15. David Walker. π-calculus Semantics of Object-Oriented Programming Languages. In: Theoretical Aspects of Computer Science, pp.532-547. Springer-Verlag, 1991
16. Jim Waldo, Ann Wollrath. RMI Tutorial. Sun Microsystems, 1999