プログラミング－言語・基礎・実践－ 5－2
(1991. 11. 21)

並行計算のための抽象アーキテクチャ

久保 誠　　所 真理雄

慶応義塾大学 大学院 計算機科学専攻

本論文では、並行性の理論、特に並行プロセス計算理論の形式表現の一つの考え方として、並行計算の抽象アーキテクチャを提案する。この抽象アーキテクチャは、並行計算に対する抽象的な実行形式を与え、また、抽象アーキテクチャは、計算の実行に関する諸性質を理論レベルで明確にすることができ、並行計算の最適化、実行環境への効果的な写像などに役立つことが期待できる。本論文では、我々の抽象アーキテクチャの基本的な枠組について概略を述べるにとどめ、各種性質に関しては別稿に譲ることとする。

An abstract architecuture for concurrent computation

Makoto Kubo　　Tokoro Mario

kubo@mt.cs.keio.ac.jp　　mario@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1, Hiyoshi, Kohoku-ku, Yokohama, 223, Japen

In this paper, we present an abstract architecuture based on concurrent process calculus. The abstract architecture is executional mechanism for concurrent computation at a high abstraction level.The architecute is expected to be useful in understanding executional properties for concurrent computation, and will be utilized for optimization,efficient mapping to executional environments, and suggestions to real computer architectures.

1 はじめに

本論文で、我々は、並行性の理論、特に並行プロセス計算理論を基礎として、並行計算の抽象アーキテクチャまたは抽象機械を提案する。

抽象機械は、現実の機械を抽象化したモデルである。現在まで、主に逐次計算に関する抽象機械が提案されてきた。1930年代に、チューリングがチューリングマシンを提案し、実行可能なアルゴリズムを、テープの読み書き、移動、状態遷移などによって定義した。この抽象機械の重要な点は、現在の計算機のアーキテクチャの基礎になっただけでなく、計算の複雑さ、コストのモデルとしても基礎となったことである。それと同時期にチャーチなどによって λ 計算が提案された。λ計算は、関数抽象、関数適用を基にし、簡約という概念によって計算を行なう関数型計算モデルである。λ 計算は、簡約(リダクション)という概念に基づき、チューリングマシンと同等の関数の族を表現できる。λ 計算自体は、計算の実行に関するフォーマリズムという側面が少なかったが、その後の様々な抽象アーキテクチャの基礎となった。まず、1960年代になって、ランディンが、λ計算のリダクションを、スタックと環境などによって実行するための機械であるSECDマシンを提案した。このSECDマシンは、必ずしも効率が良かったわけではないが、関数の実行に対し、明確な性格付けを与えた。その後、λ計算の構文構造をグラフで表し、リダクションをグラフの変換によって表現するグラフリダクションモデルが提案された。このモデルは、グラフでの表現によるために、項の共有を表現しやすい。その他に、関数型計算モデルとして、データフローモデルが存在する。このモデルは、関数の引数の入力に注目し、必要なデータがそろったら計算を実行するモデルで、実行順序は、関数の入出力の関係のみの関係のみで表現できる。

しかし、実世界のシュミレーションとしての計算を考える立場から、たがいに相互作用しあう並行分散計算の考え方の方が自然である [12]。だが、そのような計算概念に対しては、明確に並行性の理論や抽象機械は、あまり提案されていなかった。最近、化学反応メタファに基づく化学抽象機械 (Chemical Abstract Machine)[1] が提案された。この抽象機械は、実行メカニズムとして多重集合を使って表現したり、構造と計算の分離等などを導入することによって、並行計算を簡単に表現できたが、、その多重集合をどう実現するかに関してあまり明確になっていなかった。例えば、多重集合間の可換性、重複性をどう実現するかなどである。

その一方で、並行性をプロセスとその通信でしめしたプロセス間通信モデルが提案されてきた。代表的な例として、CCS[6] がある。CCSは、並行な通信に基づく、単純かつ優雅なフォーマリズムであり、いろいろな興味深い意味論が提案されてきた。しかし、様々なデータ構造・制御構造の表現や実行形式の写像という面では、必ずしもすぐれているとはいえない。最近提案された、CCSにポート名の送信と動的なポート生成の機能を拡張した、π計算 [8, 10, 9] は、動的な計算を表現でき、さまざまなデータ・制御構造で、λ計算や並行オブジェクト計算を表現する能力を持つ。その表現能力に関しては、[10, 9] に示されている。π計算の他に、拡張されたCCSとして、CHOCS[11] やγ計算 [2, 1] などのプロセスそのものを送信するモデルも提案されている。これらは、λ計算を、プロセス渡しに対応させて表現しているが、実現モデルとして考えると、プロセス自身の転送を含むための効率が悪くなるだろう。

このように、π計算は、きわめて高い記述能力をもつ一方で、計算機構の側面から見れば、きわめて単純であり、ポート名というもっとも単純なデータ構造のマッチングと具体化のみによっている。すなわち、π計算は、高い表現能力をもつ並行計算の形式系だけでなく、それ自体 (抽象的な) 実行形式という側面をそなえている。さらに、この実行形式は、きわめて高い潜在的な並行度をもち、様々な並列アーキテクチャへのマッピングを考える上でも興味深い。そこで、そのπ計算を、いろいろな実行環境に対応させるための抽象アーキテクチャとして、πアーキテクチャを提案する。このπアーキテクチャは、並行計算の抽象的な実行形式をあたえ、実行に関する諸性質を明確にする。本論文では、πアーキテクチャの遷移システムによる形式系とπ計算との間の変換の正当性の証明のみについて述べ、その理論的展開・実アーキテクチュアへのマッピング・様々な応用などに関しては、別稿に譲ることとする。

2 π計算

この節では、πアーキテクチャの基であるπ計算に関する構文と意味論について述べる。

π計算は、Nielsen,Engberg[3] により始まり、Milner[8] によって発展したポート自身を通信の引数として授受を行ない、そのポートを生成できるようなプロセス計算モデルである。ポート名の送信と生成により、動的なネットワーク構造が表現が可能である。また、この研究は、[10, 9] で、構造合同性を導入とそれに伴う構造と遷移の分離したことにより、意味論が単純になった。また、π 計算は、同期通信を基本としたのに対し、[4] では、非同期通信に基づいた形式系が提案されている。

2.1 π 計算の構文

π 計算の構文は以下のように定義される。なお、ここでは、[10, 9] で示されているπ計算で示す。

定義 1 π 計算の構文は以下の BNF によって定義される。

$$
\begin{array}{lll}
P ::= & \overline{x}y.P & (\text{メッセージ送信}) \\
| & x(y).P & (\text{メッセージ受信}) \\
| & P|Q & (\text{並行オブジェクト}) \\
| & (y)P & (\text{局所ラベル}) \\
| & A(x_1, x_2, \cdots, x_n) & (\text{定義済みエージェントの生成}) \\
| & !P & (\text{エージェント複製子})
\end{array}
$$

小文字は、名前を示している。大文字は、エージェント (オブジェクト) を示している。なお、ある定義されたエージェント内に、同じエージェント名を使用した場合、それは再帰的な表現となる。再帰とエージェント複製子は、互いを表現することが可能である。この論文では、再帰的な表現の方式を採用している。

次に、[10, 9] で導入された、構造合同性を定義する。この規則は、通信の規則の中にある構造的に等しい性質を定義し、それによって、後述の操作的意味論から、静的な関係に関する規則を分離することが可能になり、意味論が単純になった。

定義 2 構造合同性 (structure congruence) は、以下の等式を満足する最小の合同関係であり、≡で表す。

1. $P \equiv Q$ (Pは Q にα変換可能)
2. $P|0 \equiv P, P|Q \equiv Q|P, P|(Q|R) \equiv (P|Q)|R$
3. $(x)0 \equiv 0, (x)(y)P \equiv (y)(x)$
4. $(x)(P|Q) \equiv P|(x)Q \ (x \notin fn(P))$
5. $A(y) \equiv P[y/x](A(x) \stackrel{def}{=} P)$

5'. $!P \equiv P \,|\, !P$

上記の構造合同性の中で 4 の規則は、全計算システム内で一意なラベルにするために新しいラベルを生成することを意味する。1 は、α同値性を、2,3,5 は、| と () と定義済みエージェントに関する構造的同値性をあらわしている。

2.2 簡約

簡約 (reduction) は、外部に通信することなく、π 項の部分項どうしの通信することを表している関係である。

定義 3 簡約 (reduction) は、以下の規則を満足する最小の関係であり、$\rightarrow$ で表す。

$$\text{COM} \quad x(y).P \mid \overline{x}z.Q \rightarrow P\{z/y\}|Q$$

$$\text{PAR} \quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} \quad \frac{P \rightarrow P'}{(y)P \rightarrow (y)P'}$$

$$\text{STRUCT} \quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

次に、簡約の拡張として、アクションを定義する。アクションは、π 項に対する潜在的な通信、つまりある要求に対する通信に基づいて行なわれる。基本的には、guard された項により生ずる外部との入出力が考えられる。また、前節で述べた簡約は、外部に対してでなく π 項の内部の通信であり、アクションにはこれも含めて考える。これは、外部からみると何も通信を行わないので、τ 遷移とよび、$\xrightarrow{\tau}$ で表現する。このアクションの定義は、[9] に示されている。このように、π 計算の操作的意味意味論は、いくつかの単純な規則によって表現される。次節では、簡約をいくつかのステップに分解し、実行を意識したモデルである π アーキテクチャについて述べる。

3 πアーキテクチャ

3.1 構文

π アーキテクチャの構文は、基本的に π 計算と対応する。違う点として、In,Out が状態を持っている点と、名前が通信のキューを構造として含んでいる点である。文字列ではなく、グラフ表現であることを強調するために、データ構造はポインタの組の形で表現している。

定義 4 抽象構文。
ポート名 N, ポート名の列 $\tilde{N}$, エージェント変数 T, エージェント A は、以下の抽象構文によって与えられる。

$$\begin{array}{lcll}
I & = & x \mid I' & (\text{ポート識別子}) \\
N & = & \{I, \tilde{N}, \tilde{N}, N\} \mid Nil & (\text{ポート名}) \\
\tilde{N} & = & \varepsilon \mid N\tilde{N} & (\text{ポート名の列}) \\
T & = & X \mid T' & (\text{エージェント変数}) \\
\tilde{A} & = & \varepsilon \mid A\tilde{A} & (\text{エージェントの列}) \\
A & ::= & (In, N, N, A, S) & (\text{ポート名受信エージェント}) \\
 & \mid & (Out, N, N, A, S) & (\text{ポート名送信エージェント}) \\
 & \mid & (Com, \tilde{A}) & (\text{並行エージェント生成}) \\
 & \mid & (Def, T, \tilde{N}) & (\text{エージェント生成}) \\
 & \mid & Nil & (\text{ニル}) \\
S & = & Wait \mid Comm \mid Done & (\text{状態}) \quad \blacksquare
\end{array}$$

π アーキテクチャの基本データ構造は、名前とエージェントからなる。π アーキテクチャの名前とエージェントは、π 計算の名前とエージェントに対応している。

名前は、一意性を保証する識別子と、この名前を入力および出力ポートとして使用するエージェントキューへのポインタからなっている。このポインタは、入力と出力の二種類ある。また、変数である場合、その実体の名前を指すポインタを有している。このポインタによって、ポート名の置換と動的なポート生成を実現している。名前は、以下のような形式で表す。

name = {input-queue , output-queue , instance }

このキューにはいっているエージェントは、現在このポート名で通信を行なおうとしているエージェントである。また、

エージェントは、エージェントのそれぞれの型とその型に基づく実体をもつ。In,Out は、通信を行なうエージェントを示し、第一引数の名前のポートを使用し、第二引数の名前を出力するか、その名前に入力した名前を具体化し、第 3 の引数のエージェントになる。state は通信の現在の状態を表し、Wait,Comm, Done の三つの状態によって表される。Com は、agent list にあるエージェントを並行に実行するように生成する。Def は、Agent name で示された、グラフのエージェントを生成する。なお、π計算のここでは示さなかった構成子 (Match,Summation) についても、対応するエージェントが定義されてはいるが、ここでは簡単化のために扱わない。

3.2 πアーキテクチャの遷移システム

π アーキテクチャの遷移システムは、以下の各遷移規則を実行するシステムの並列実行により実現される。

π アーキテクチャの configuration は、すべての名前の集合と、現在アクティブなエージェントの集合の対から構成され、<N,A>の形で表される。

また、リストは、[A|B] または $[A_1,A_2,\cdots,A_n]$ の形で表す。集合は、<A,B,C>のように表し、その要素の順序は関係ない。また、なお、あるエージェントまたは名前を示す

πアーキテクチャの遷移は、 $< N, A > \rightarrow_A < N', A >$の形で表し、configuration <N,A>が $< N', A' >$に遷移することを示している。

定義 5 *transition* は 以下の規則を満たす最小の関係で、$\rightarrow_A$ で表す。

Communication

$$\frac{a \mapsto \{[i|I_1],[o|O_1],NIL\}}{<< a,x,y|N >,< i,o|A >> \rightarrow_A << a_1,x_1,y|N >,< i_1,o_1|A >>}$$

$$where \begin{cases} a_1 & \mapsto & \{I_a,O_a,NIL\} \\ x & \mapsto & \{I_x,O_x,NIL\} \\ x_1 & \mapsto & \{I_x,O_x,y\} \\ y & \mapsto & \{I_y,O_y,NIL\} \\ i & \mapsto & (In,a,x,P,Comm) \\ i_1 & \mapsto & (In,a,x,P,Done) \\ o & \mapsto & (Out,a,y,Q,Comm) \\ o_1 & \mapsto & (Out,a,y,Q,Done) \end{cases}$$

Synchronization-In

$$\frac{p \mapsto (In,a,x,P,Wait)}{<< a,z|N >,< p|A >> \rightarrow_A << a,z_1|N >,< p_1|A >>}$$

$$where \begin{cases} a & \mapsto & \{I_a,O_a,z\} \\ z & \mapsto & \{I_z,O_z,NIL\} \\ z_1 & \mapsto & \{[I_a|I_z,p_1],[O_z|O_a],NIL\} \\ p_1 & \mapsto & (In,z_1,x,P,Comm) \end{cases}$$

Synchronization-Out

$$\frac{p \mapsto (Out,a,x,P,Wait)}{<< a|N >,< p|A >> \rightarrow_A << a_1|N >,< p_1|A >>}$$

$$where \begin{cases} a & \mapsto & \{I_a, O_a, z\} \\ z & \mapsto & \{I_z, O_z, NIL\} \\ z_1 & \mapsto & \{[I_a|I_z], [O_a|O_z, p_1], NIL\} \\ p_1 & \mapsto & (Out, z_1, x, P, Comm) \end{cases}$$

Transition

$$\frac{p \mapsto (In, a, x, Q, Done) \ or \ p \mapsto (Out, a, x, Q, Done)}{< N, < p|A >> \rightarrow_A < N, < Q|A >>}$$

Composition

$$\frac{p \mapsto (Com, [A_1, \cdots, A_n])}{< N, < p|A >> \rightarrow_A < N, < A_1, \cdots, A_n|A >>}$$

Defined Agent

$$\frac{p \mapsto (Def, A, [x_1, \cdots, x_n]) \quad A \overset{Def}{\mapsto} ([y_1, \cdots, y_n], Q)}{< N, < p|A >> \rightarrow_A << N|y_1, \cdots, y_n >, < Q|A >>}$$

$$where \ y_i \ \mapsto \ \{I_i, O_i, x_i\}$$

3.3 π計算からπアーキテクチャへの変換

π計算の項は、πアーキテクチャ上で、エージェントの木構造に変換される。その変換規則は以下の表に示す。

$$\begin{aligned} a(x).P &\rightarrow (In, a, x, P, Wait) \\ \overline{a}y.P &\rightarrow (Out, a, y, P, Wait) \\ A|B &\rightarrow (Com, [A, B]) \\ A(x_1, x_2, \cdots, x_n) &\rightarrow (Def, A, [x_1, x_2, \cdots, x_n]) \\ A(y_1, y_2, \cdots, y_n) \overset{def}{=} P &\rightarrow A \overset{def}{\mapsto} ([y_1, y_2, \cdots, y_n], P) \end{aligned}$$

また、一番最初 (外側) の項が、木のルートをしめし、最初の configuration のエージェントの要素となる。

ここで、π計算とπアーキテクチャの遷移の等価性を示すために、以下の変換の関数である Load,Unload を定義する。

Load:π項 $\rightarrow$ πアーキテクチャ configuration

$$Load(A|B|C) = << n(A) \cup n(B) \cup n(C) >, < A, B, C >>$$

Unload:πアーキテクチャ configuration $\rightarrow$ π項

$$Unload(< N, < A, B, C, D >>) = A|B|C|D$$

変換の等価性を表す以下の定理が成り立つ。

定理 P,Q があるπ項を、P_A,Q_Aがπアーキテクチャの configuration であり、$P_A = Load(P)$ かつ $Q = Unload(Q_A)$ であるとすると、以下のことが成り立つ。

1. $P \rightarrow Q \Rightarrow P_A \rightarrow_A^* Q_A$
2. $P_A \rightarrow_A^* P'_A \Rightarrow P \rightarrow^* Q$ かつ $P'_A \rightarrow_A^* Q_A$

証明

1の証明

この定理の証明は、πの簡約規則の構造に関する帰納法を使って証明する。

(1) COM の場合は、

$$a(x).P \mid \bar{a}y.Q \rightarrow P\{y/x\}|Q$$

という遷移が存在する。この時、

$$\begin{array}{lcll}
P_A & = & Load(a(x).P|\bar{a}y.Q) & \\
 & = & << a, x, y, N >, < I, O >> & \\
 & \rightarrow_A & << a', x, y, N >, < I', O >> & (Synchronization - In) \\
 & \rightarrow_A & << a'', x, y, N >, < I', O' >> & (Synchronization - Out) \\
 & \rightarrow_A & << a''', x', y, N >, < I'', O'' >> & (Communication) \\
 & \rightarrow_A & << a''', x', y, N >, < P, O >> & (transition - In) \\
 & \rightarrow_A & << a''', x', y, N >, < P, Q >> & (transition - Out)
\end{array}$$

$$where \begin{cases}
a & \mapsto & \{NIL, NIL, NIL\} \\
a' & \mapsto & \{I', NIL, NIL\} \\
a'' & \mapsto & \{I', O', NIL\} \\
a''' & \mapsto & \{NIL, NIL, NIL\} \\
x & \mapsto & \{NIL, NIL, NIL\} \\
x' & \mapsto & \{NIL, NIL, y\} \\
y & \mapsto & \{NIL, NIL, NIL\} \\
I & \mapsto & (In, a, x, P, Wait) \\
I' & \mapsto & (In, a, x, P, Comm) \\
I'' & \mapsto & (In, a, x, P, Done) \\
O & \mapsto & (Out, a, y, Q, Wait) \\
O' & \mapsto & (Out, a, y, Q, Comm) \\
O'' & \mapsto & (Out, a, y, Q, Done)
\end{cases}$$

Unload(<< a''', x', y, N >, < P, Q >>) = P{y/x} — であり、1 が成り立つ。

(2) PAR の場合は、$P \rightarrow P'$ ならば、 $P_A \rightarrow_A^* P'_A$であると仮定すると、

$$\begin{array}{lcl}
Load(P|Q) & = & < n(P) \cup n(Q), < P, Q >> \\
 & \rightarrow_A^* & < n(P)' \cup n(Q), < P', Q >> \quad \text{(帰納法の仮定)}
\end{array}$$

$Unload(< n(P)' \cup n(Q), < P', Q >>) = P'|Q$ となり、1 が成り立つ。

(3) RES の場合は、πアーキテクチャでは、一意の名前をコンパイラが生成する。また、動的に生成している def の場合は、グラフを生成時に他のものと識別できるような識別子が生成しているので、この規則は、満たされている。

(4) STRUCT の場合は、STRUCT の 1 は、識別子の使用により自明である。2 は、πアーキテクチャの configuration が集合を表していることから自明である。3,4 は Restriction のための規則であるため、(3) の場合と同様に満たさ

れる。5 の場合は、π アーキテクチャの Defined Agent の規則により満たされることがわかる。

2 の証明

この定理の証明は、πアーキテクチャの構造に関する帰納法をつかって証明する。P, P_Aは以下の形であると仮定する。

$$P = A_1|\cdots|A_n$$

$$P_A =< N, < A_1, \cdots, A_n >>$$

(A_iの型は、In,Out,Def のいずれか)

$\rightarrow_A$の数が 0 の時は、最初の状態と変わっていないので、π計算においても遷移の行なわれていない状態に対応する。

$\rightarrow_A$の数が 1 以上の時に、以下の場合を考えると、

(1) P'_Aのエージェントの型が、In または Out で状態が wait である場合は、

$P_A \rightarrow^k P'_A$ かつ、$Unload(P'_A)$ が、a(x).P または$\bar{a}y.Q$ の形をしている。これは定理を満たす。

(2) P'_Aのエージェントの型が、In または Out で状態が comm の場合は、

$$\begin{array}{lll}
P'_A = & << a, x, N >, < I >> & (AgentA \text{の状態が} comm \text{で、型が} In/Out) \\
\rightarrow_A & << a', x, y, N >, < I, O >> & (Synchronization - Out/In) \\
\rightarrow_A & << a'', x', y, N >, < I', O' >> & (Communication) \\
\rightarrow_A & << a'', x', y, N >, < P, O >> & (transition - In) \\
\rightarrow_A & << a'', x', y, N >, < P, Q >> & (transition - Out)
\end{array}$$

$$where \begin{cases}
a & \mapsto \{I, NIL, NIL\} \\
a' & \mapsto \{I', O', NIL\} \\
a'' & \mapsto \{NIL, NIL, NIL\} \\
x & \mapsto \{NIL, NIL, NIL\} \\
x' & \mapsto \{NIL, NIL, y\} \\
y & \mapsto \{NIL, NIL, NIL\} \\
I & \mapsto (In, a, x, P, Comm) \\
I' & \mapsto (In, a, x, P, Done) \\
O & \mapsto (Out, a, y, Q, Comm) \\
O' & \mapsto (Out, a, y, Q, Done)
\end{cases}$$

Unload($<< a'', x', y, N >, < P, Q >>$) = P{y/x} — Q となって、(2) が成り立つ。

ここで、通信するべき相手がいない場合は、πアーキテクチャは状態が遷移しないが、π計算でも遷移は存在しないことに注意すること。

(3) P'_Aのエージェントの型が、In または Out で状態が Done の場合は、

$$\begin{array}{lll}
P'_A = & < N, < A >> & (AgentI \text{の状態が} Done \text{で、型が} In/Out) \\
\rightarrow_A & < N, < P >> & (transition) \\
& A \mapsto (In/Out, z_1, x, P, Done) &
\end{array}$$

Unload($< N, P >$) = P となって、(2) が成り立つ。

(4) P'_Aのエージェントの型が、Com の場合は、

$$\begin{array}{rcll} P'_A & = & < N, < A >> & (AgentA \text{ が } Com) \\ & \rightarrow_A & < N, < A_1, \cdots, A_n >> & (Composition) \\ & & A \mapsto (Com, [A_1, \cdots, A_n]) & \end{array}$$

Unload$(< N, < A_1, \cdots, A_n >) = A_1 | \cdots | A_n$ となって、(2) が成り立つ。
(5) P'_Aが、(1)-(3) までの集合 $< P'_1, \cdots, P'_n >$である場合は、

$$\begin{array}{rcll} P'_A & = & < N, < P'_1, \cdots, P'_n >> & \\ & \rightarrow^*_A & < N, < P''_1, \cdots, P''_m >> & \\ & & P''_i \mapsto (In/Out, a_i, x_i, P, Comm) & \\ & \rightarrow^*_A & < N, < P'''_1, \cdots, P'''_m, Q_1, \cdots, Q_r >> & (Communication \text{ の列}) \\ & & P'''_i \mapsto (In/Out, a_i, x_i, A_i, Done) & \\ & & Q_i \mapsto (In/Out, a_i, x_i, P, Wait) & \\ & \rightarrow^*_A & < N, < A_1, \cdots, A_m, Q_1, \cdots, Q_r >> & (Transition \text{ の列}) \end{array}$$

Unload$(< N, < A_1, \cdots, A_m, Q_1, \cdots, Q_r >) = A_1 | \cdots | A_m | \cdots | Q_r$ となって、(2) が成り立つ。
この定理より、π計算を実行するための形式系としてπアーキテクチャが正当であることが証明された。 ■

4 結論

本論文では、並行計算のためのアーキテクチャの一つとしてπアーキテクチャを提案した。πアーキテクチャの遷移は、π計算の遷移を細かく分別することで、計算の実行概念をより直接的に表現している。例えば、計算に内在する並行性や衝突などの性質が明示的に記述されている。一般的な計算の実行形式として見れば、計算の順序を保証するのは、通信による因果関係あるいは依存性 (dependency) のみで、あとは並行性を最大限生じさせるような構成になっている。このようなπアーキテクチャの性質を利用した各種言語の実行形式理論、最適化理論がきわめて興味深い研究課題である。しかし、これらに関しては、別稿に譲ることにする。

参考文献

[1] Berry, G. and Boudol, G., *The Chemical Abstract Machine.* in POPL 1990, 1990.

[2] Boudol, G., *Towards a Lambda Calculus for Concurrent and Communicating.* in TAPSOFT 1989,LNCS 351,1989.

[3] Enberg,U., and Nielsen,N., A Calculus of Communicating Systems with Label-Passing. report DAIMI PB-208, Computer Science Department,University of Aarhus, 1986.

[4] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication, *Proc. of European Conference on Object-Oriented Programming*, LNCS, Springer-Verlag, July 1991.

[5] Honda, K. and Tokoro, M., On Asynchronous Communication Semantics, *Proc. of Workshop on Object-based Concurrent Computing*, LNCS, Springer-Verlag, 1992.

[6] Milner,R.. *A Calculus of Communicating Systems.* Volume 92 of *LNCS*, Springer Verlag, 1980.

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

[8] Milner,R., Parrow,J., and Walker,D., *A Calculus of Mobile Processes. Part I and II.* ECS-LFCS-89-85/86, Edinburgh University, 1989.

[9] Milner,R., *Function as Processes.* Rapports de Recherche No.1154, INRIA-Sophia Antipolis, 1990.

[10] Milner.R., Function as processes. In ICALP 90. LNCS 443, Springer Verlag, 1990.

[11] Thomsen, B., *A calculus of higher order communicating systems.* in POPL Proceedings, 1989.

[12] Tokoro, M., *Issues In Object-Oriented Distributed Computing.* in: *Proceedings of 4th Conference of Japan Society for Software Science and Technology*, September 1988.(in Japanese)