ν計算・並行結合子・プログラミング言語

本田耕平、久保 誠、指野篤司、竹内格、
バスコ バスコンセロス、吉田展子

{kohei,kubo,sashi,kaku,vasco,yoshida}@mt.cs.keio.ac.jp

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

1 はじめに

現在のネットワーク社会において、通信をソフトウェア構成の基礎におく、並行・分散プログラミングは、ますますその重要性を増している。しかし、現在のシステム構成技術は、基本的に従来の単一計算機を基礎にした概念の拡張、つまり複数の逐次計算機とその間のネットワーク通信という枠組に基づいている。これは、プログラミングの基礎理論において、チューリングマシンやλ計算のような逐次・関数型計算の基礎理論にかわる、並行・通信に基づく計算理論が存在しなかったこと、より本質的には、(カール・ヒューイットのアクターモデルなどの先駆的試みを除き) 並行算体間の相互通信を根本にすえて計算・プログラミングの概念を基礎から捉え直す試みがなされてこなかったことが大きな原因であろう。もしも命令型のプログラミング・計算概念、関数型のプログラミング・計算概念が、その基盤レベルにおいて、それぞれきわめて明確な基本操作 (代入・ジャンプ・関数適用等) に的を絞ることによって、体系だち、かつ、有用なソフトウェア構成・実装技術を可能にしてきたとするなら、並行・分散ソフトウェア構成の基本原理も、やはり単純な基本操作の概念、おそらくは通信あるいは相互作用という概念を基礎に据えた探求によって見出されるかもしれない。

本稿では、このような可能性に向けてのわれわれのここ数年の試みの一部を、自由に概括していく。特に、本稿の性質および長さの上での制約から、特にν計算・並行結合子・言語$\mathcal{L}$に中心をおいて、あらましを述べるにとどめ、それらの詳細および他の研究については関連文献を紹介することでこれにかえる。以下、簡単に各節の内容を示す。第2・3節では相互通信に基づく基礎理論体系であるν計算と並行結合子系を紹介する。第4節では通信という概念を計算抽象の根本にすえてプログラミングプリミティブを徹底的に考え直した試みと、その結果うまれた言語$\mathcal{L}$について、その基本概念を示す。第5節では、その他の研究をまとめる。

2 ν計算

いかなる数学的概念も、独立してあるのではなく、他の様々な概念と関連してその意味が明確になり、また有用性が明らかになる。顕著な例として、きわめて異なった形式的体系によって、同等の概念が表現される例が有り、これはその概念が数学的に頑強なものであることを示すと同時に、有用な相互利用の可能性を与えてくれる。顕著なものとして、チューリングマシンの計算可能性

の概念・クリーネの帰納的関数・チャーチのλ定義可能性などの計算可能関数における一致がある。また、λ計算とSK結合子系という対応も、高階関数にもとづく計算概念さらにはそこでの実装枠組における重要な理論的基礎をなしている。

本節と続く3節では、ν計算と並行結合子系という二つの（きわめて単純な）基礎計算理論を紹介する。これは、並行計算における、上記のλ計算とSK結合子系という組合せの対応物とみることもできる。ν計算は、非同期名前通信という単純な通信概念のみを基礎に据えた形式系であり、λ計算とおなじように、(相互作用計算の世界において）単純さと豊かな表現能力とを併せ持つ。いっぽう、後者は有限個の原子とそのあいだの（やはり有限個の）相互作用規則というもののみから構成された形式体系であり、特に「通信」という概念をその枠組みに含まないが、実はν計算と同等な表現能力を持つ。これらの形式系の探求は、それ自体理論的に興味深いものだが、特に、通信と並行性というものを基礎においたソフトウェアというものを考える際の、瑣末でない・かつ原基的な材料を提供する。例えば、言語のデザイン・ソフトウェアの合成・意味的な正当性の検証・最適化などを考えるときに、これらの形式系で考えてみること、あるいは蓄積された成果を用いることで、原理的な指針を得ることができるのである。

ν計算は、CCS、特にπ-計算 [21, 19] を基盤として、意味を持つ限りでぎりぎりまで単純化された並行プロセス計算である。この形式系は、項とよばれるいわば「プログラム」(実際は並行に走行するプロセスの集合ととらえたほうがよいかもしれない）の構成規則と、それがプロセス同士の通信により、どのように別の「プログラム」(プロセス集合）へと変化していくかを示す簡約則から構成されている。まず、ν-項は、以下の文法によって定義される。要素は、$P,Q,...$によって示されている。

$$P \ ::= \ \leftarrow uv \ \mid \ ux.P \ \mid \ P,Q \ \mid \ a \blacktriangleright P \ \mid \ !_u P \ \mid \ \Lambda$$

ここで、$a,b,c,..$ は「ポート名」(今後簡単に「名前」と呼ばれる）を示し、$x,y,z,..$ はそれに対する変数（「名前変数」）である。また、$u,v,w,..$ は これらのどちらか（識別子と呼ばれる）を示す。

記法からわかるように、“$\leftarrow uv$” は、値である名前 vを u に運ぶメッセージを示している。“$ux.P$” は受容子、つまりメッセージを受けとり、その後、本体 P内の x を受けた値で具体化するようなエージェントを意味する。$ux.P$の x は P内の自由な出現である x を束縛している (この概念は$\lambda x.M$内の x と同等である)。P,Q は、P というプロセスと Q というプロセスが並行に動作していることを表す。 一方、$a \blacktriangleright P$ は、P のなかの a がそのなかだけで局所的な名前ということを示す。$!_u P$ は複製子とよばれ、プロセスを任意回生成するテンプレートである。最後にΛ はエージェントが存在しないことを示す記号である。

ν計算での計算概念は、上記で述べたように「簡約則」によって表現される。これは

$$\begin{array}{ll}
\text{(COM)} & ux.P, \leftarrow uv \ \longrightarrow \ P\{v/x\}. \\
\text{(REP)} & !_u P, \leftarrow uv \ \longrightarrow \ !_u P, \ P. \\
\text{(PAR)} & P \longrightarrow Q \ \Rightarrow \ P,R \longrightarrow Q,R. \\
\text{(RES)} & P \longrightarrow Q \ \Rightarrow \ a \blacktriangleright P \longrightarrow a \blacktriangleright Q. \\
\text{(STR)} & P \equiv P' \ \ P' \longrightarrow Q' \ \ Q \equiv Q' \ \Rightarrow \ P \longrightarrow Q.
\end{array}$$

というようにあたえられ、$\longrightarrow\!\!\!\!\rightarrow$は$\longrightarrow^* \cup \equiv$と定義される。ここで、$\equiv$ は構造的等価性とよばれる、[1] に示唆を得た、基本的な項に対する等価則である。ここでは詳細は述べない。重要な規則は、COM と REP の二つの規則で、COM は通信によって名前置換が起こるという規則であり、REP はやはり通信（ただし値の受渡しを伴わない同期）によって項の複製が起こるという規則である。重要なことは、これら二つの基本操作のみによって、逐次・関数・並行の様々な計算事象を容易に表現できるということである。ここでは以下の基本的な事実を述べるにとどめる。

(i) 部分帰納関数の族を、λ計算と同等の手法で定義可能である。これは、ν計算が、関数として計算表現可能なものを全て表現できることを示す。。

(ii) 様々な相互作用の構造、例えば非同期多値通信・同期通信・同期多値通信・枝わかれ構造を、非同期名前渡しの九去りによって表現できる（これにより、例えばMilnerのπ-計算はν計算によって定義できる）。また、例えば論理変数への代入のような操作も、相互作用の構造として表現することができる。これによって、並行手続き型言語だけではなく、例えば並行論理・制約型言語やUnityなどの言語の基本操作も表現することができる。

(iii) POOL, CST, ABCLなど、並行オブジェクト指向プログラミング言語[36]、特にアクターモデル[6]と対応する部分が多い（本形式系は本来アクターモデルのπ計算による表現を行なうなかで生まれたものである）。

ν計算は、通信にもとづく並行計算として通常考え得る計算表現の本質的な操作的要素を全て、できるだけ単純で（表現力という点からは）不必要なものを徹底的にそぎおとした構成で表現しようとする試みである。いわば、COMとREPという二つの操作に、相互作用に基づく並行計算の概念を凝縮しようとしている。もちろんこれ自体がプログラミング言語としてよい道具なわけではない。むしろ、このν計算を用いた様々な計算表現を通じて、通信を基礎としたソフトウェア構成のかたちと問題点を最も本来的なかたちで知ることができ、それを通じて本当に有効なプログラミングの抽象化へむけての示唆を得るのである。

ν計算に関する研究の詳細は、[8, 9, 3, 10, 7]などを参照。さらに、[35]では、ν計算の二つの操作を基礎とした関数型計算であるλf計算について述べている。また、ν計算の母体となったπ計算については、[19, 20, 21]を参照のこと。

3 並行結合子

並行結合子[10]は、少数の原子とその間の（通信さえともなわない）単純な相互作用規則のみで計算をすすめていく計算体系である。ある意味で、物理的自然の原子とにたものと考えることもできる。このような原子として、

$$\mathcal{M}^{(2)+},\ \mathcal{D}^{(3)-},\ \mathcal{FW}^{(2)-},\ \mathcal{K}^{(1)-},\ \mathcal{B}_l^{(2)-},\ \mathcal{B}_r^{(2)-},\ \mathcal{S}^{(3)-}$$

を使用する。ここで、$C^{(n)\theta}$において、nおよびθはそれぞれCのアリティーと極性と呼ばれる。この原子のうえに、

$$P \ ::= \ C(\tilde{u}^{(n)}) \ \mid \ P,Q \ \mid \ a \blacktriangleright P \ \mid \ \Lambda$$

という結合子項が与えられる。これらの項が並行計算プログラムあるいはプロセスを表現していることはν計算と同じであるが、受容子の表現がないことに注意してほしい。次に、各エージェントの相互作用規則を与える。

- 複製子（duplicator）: $\mathcal{D}(uww'),\ \mathcal{M}(uv) \longrightarrow \mathcal{M}(wv),\ \mathcal{M}(w'v)$. ここでは、メッセージ／スレッドが複製される。
- 中継子（forwarder）: $\mathcal{FW}(uw),\ \mathcal{M}(uv) \longrightarrow \mathcal{M}(wv)$. 中継子は異なる場所のリンクとして機能する。
- 消去子（killer）: $\mathcal{K}(u),\ \mathcal{M}(uv) \longrightarrow \Lambda$. ここでは、メッセージ／スレッドが消去される。

以上のように、$\mathcal{D}, \mathcal{FW}, \mathcal{K}$はそれぞれ異なった種類のリンクとして振舞う。

このほかに、3つの規則がある。

- 左束縛子（left binder）: $\mathcal{B}_l(uw),\ \mathcal{M}(uv) \longrightarrow \mathcal{FW}(vw)$.
- 右束縛子（right binder）: $\mathcal{B}_r(uw),\ \mathcal{M}(uv) \longrightarrow \mathcal{FW}(wv)$.

- 同期子 (synchroniser): $\mathcal{S}(uww'), \mathcal{M}(uv) \longrightarrow \mathcal{FW}(ww')$. 同期子は、受けとった値によらずリンクを生成する。

- $\mathcal{B}_l, \mathcal{B}_r, \mathcal{S}$はリンク (中継子) を生成することで動的に通信トポロジーを変化させる. これによって複雑な相互作用構造の構成が可能になる.

このように、各規則は、二つの極性の異なった原子が出会ったときに、どのように計算が起こるか、ということを示している。このうえで、

$$\begin{aligned}
&(\mathrm{PAR}) \quad P \longrightarrow Q \;\Rightarrow\; P,R \longrightarrow Q,R.\\
&(\mathrm{RES}) \quad P \longrightarrow Q \;\Rightarrow\; a\blacktriangleright P \longrightarrow a\blacktriangleright Q.\\
&(\mathrm{STR}) \quad P \equiv P' \;\; P' \longrightarrow Q' \;\; Q \equiv Q' \;\Rightarrow\; P \longrightarrow Q.
\end{aligned}$$

という規則によって、簡約則を定義する。実際にこのような単純な規則のみでν計算の名前通信と同等なことが表現できるのだが、これについては後記文献に譲る。そこには、どのようにして各原子およびその相互作用規則が、並行結合子項の基本的な構成原理から論理的に導出されたかも示されている。

さて、並行結合子系は、必ずしも上記の原子を持つシステムに限定されるわけではない。むしろ、応用に応じて様々な原子とその相互作用則を与えることができる。これらは、ある意味で、並行プログラミング言語における「定数」の概念を与えるのだが、それだけでなく、例えば思考実験として、様々な原子と相互作用規則をどんどん加えていって、表現能力を限りなく高めていくことが考えられる。 興味深いのは、上記に紹介した7この原子をもつシステムの若干の拡張によって、思考実験として考えられる、いかなる一般並行結合子系もシミュレートできるということである。この、ある意味で「完全」な並行結合子系は、第2節で示したν計算と厳密に意味論的な対応を持つ。

並行結合子系については、[12, 13, 14] などを参照。

4 言語 $\mathcal{L}$

ν計算や並行結合子の存在とそれらに関する基本的な理論的結果は、通信に基づく並行計算という枠組が、従来の関数型計算や逐次型計算の世界に還元されない、固有の基礎的な性格をもっていることを示している。このことは、例えば**プログラミング言語 ・ データ型・型理論** というものを考えるうえでも、通信・並行性という概念のみから出発することで、従来のプログラミングイディオムの延長あるいは並行計算への適応というのではない、新たな計算構成原理が得られるのではないかという期待をいだかせる。実際にこのような試みは、新たなプログラミング構成原理と型概念を生み、これらは言語 $\mathcal{L}$ として具体化された。長さの関係からここではその基礎理念を示すにとどめるが、こうしてうまれた並行計算の基礎的構成子の考え方は、ν計算での複雑な並行エージェント記述の経験からわれわれが得た知見 (ここでは詳述しない) と相応するものである。

一般の並行プログラミングのアプリケーション、例えばサーバー・クライアントによる分散環境やロボットのインターフェース等、では、二者間においてかつ連続的になされるときにはじめて意味のあるものとなる場合が多い。例えば複数の並行プロセスからなるアプリケーションでは、構成する複数の並行プロセスがお互いに相互作用しあう。それは時としてなんらかのサーバープロセスとの通信であったり、オペレーティングシステムとの通信であったりし、全体としてアプリケーションシステムの計算が進んでゆく。そのような状況では個々の通信動作では二者間の連続的な相互作用の文脈 (流れ) の維持が重要となる。他にもウィンドウシステムにおける、マウスやキーボードからの入力とそれに対応したディスプレイやその他の機器への出力の繰り返しなどによる、ユーザーとコンピュータとの相互作用などもあげられる。

上の状況を考えた上で、複雑で動的な相互作用の構造を厳密にそして形式的に抽象化する機構は明らかに必要であることがわかる。そしてその機構とは、(call-return や send-and-forget を

含めた) 通常の「単一」の通信事象に加え、複数の相互に連続な通信事象を合成化しひとかたまりの事象としてとらえるようなものであるべきである。またそのような通信事象の合成の概念は何らかの型理論の上に築かれるものであると思われる。

「型」はものごとを系統立てて分類したり、抽象化するなどの際に便利な概念である。とくに、計算機科学における型の理論では多くの利点が発見されている。関数の基礎理論として研究されてきた型付き λ 計算では、λ 項の型付け可能性 (typability) は強正規化性 (Strogly Normalizing)、つまり項の停止性を保証することができる。また、ML のような関数プログラミングでは、型システムにより実行時の型エラーの回避を保証している。プログラマにとっても、型を用いることにより、プログラムモジュールの属性や振舞いを分類することが可能となる。そして、型検査により、データに対する意図した解釈を守ることが可能となる。特に、同時にプログラムモジュールのパラメータや返り値の型などの整合性も検査するので、大規模なアプリケーション作成やプログラムモジュールの再利用などにおいて非常に便利である。また、コンパイラでは型を利用した最適化を行うことが可能となる。このように、型の概念はプログラム作成の支援機構としても、また、実行時の処理系の型検査に関する処理のコストを軽減できるなど、さまざまな利点がある。

今まで、「型」はプログラミングで用いられるデータ・値 (およびその格納場所)、レコードや配列、プログラムモジュールや関数、そして抽象データなどのようなものに対して用いられてきた。さらに、型は通信事象の分類・抽象化にも活用できるであろうと考えられる。通信事象のための型概念があれば、通信を多用したプログラムの通信事象に関する整合性を検査することが可能となるであろう。つまり、通信事象のための型システムがあれば、型付け可能なプログラムは不整合な通信事象により生じるランタイムエラーを引き起こさないということを保証することができるであろう。このことは、POOL や Actors [6] のような並行オブジェクト指向パラダイムやあるいは CML において行われる、動的な通信ポートの転送を含んだ相互作用において特に重要である。

言語 $\mathcal{L}$ ではこのような相互作用の抽象化のためのプログラミング言語プリミティブや型概念の提案を行っている。これらは一つ一つは単純なものでありながら、複雑な、そして連続的な相互作用の構造を持つプログラムの記述を、柔軟な形で可能にする。また、$\mathcal{L}$ は型なしプログラミング言語だが、型システムにより、効果的な型付けがなされる。それはいわば ML [4] の型システムが関数適用の一貫性を検査するように、相互作用の一貫性を検査する方法を提供する。柔軟な相互作用のプログラミングおよび型抽象双方についての重要なプリミティブは「セッション」の概念であり、二者間での私的なチャネルを用いて、この二者間の一連の相互作用を保護する機構である。また、いわゆるポートは複数のプロセスから参照されうるが、チャネルはこれとは異なり、セッションの開始により二者間のみから参照される通信の点 (session-point ともいう) である。このチャネルと通常のポートの区別は、相互作用の柔軟な合成と型抽象の双方に対して本質的なものとなる。またこのことは、プログラムの型付け可能性が二者間の通信パターンの不一致を含むランタイムエラーの回避を保証することにより示される。この問題は並行・分散環境における通信を基礎とするプログラミングにおいて重要であると思われる。

言語 $\mathcal{L}$ およびその型システム・型チェックアルゴリズムの詳細については、[31, 32] を参照。また、$\mathcal{L}$ 以外の近年のプロセス間の通信事象の型抽象に関する研究としては、[26] [29] [2] [24] [28]、および第 5 節で述べる TyCO がある。

5 TYCO, 簡約意味論、Interaction Machine

以上の研究の他に、名前渡しプロセス計算における型推論系・型アルゴリズム・プログラミング環境に関する総合的研究として、ヴァスコ・ヴァスコンセロスによる TyCO [33, 34], 相互作用を基盤とするプログラムの意味論的な基礎を扱った研究として吉田らによる簡約意味論 [10, 11]、環境あるいは結合子概念を基礎にした実装枠組として、久保・指野らによる Interaction Machines [15, 30] の研究などがある。

References

[1] Berry, G. and Boudol, G., The Chemical Abstract Machine. *Theoretical Computer Science*, vol 96, pp. 217–248, 1992.

[2] Dave Berry, Robin Milner and David Turner. A semantics for ML concurrency primitives. In *Proceedings of ACM Principle of Programming Languages*, pages 119–129, 1992.

[3] Boudol, G., *Asynchrony and π-calculus.* INRIA Report 1702, INRIA, Sophia Antipolis, 1991.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] Gay, S. J., A sort inference algorithm for the poliadic π-calculus. *Proc. 20th. Annual Symposium on Principles of Programming Languages*, pp.429–438, ACM press, 1993.

[6] Hewitt, C., Bishop, P., and Steiger, R., A Universal Modular ACTOR Formalism for Artificial Intelligence. *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, pp.235–245, 1973.

[7] Kohei Honda, *ν-calculus and its combinatory representation*, in preparation.

[8] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, pp.133–147, Springer-Verlag 1991.

[9] 本田耕平 所真理雄, 並行オブジェクト計算のための形式系, コンピュータソフトウエア, 9(2), pp.75–89, March 1992.

[10] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics, *Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, pp.371–387, Springer-Verlag, 1993.

[11] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics (the full version), submitted for publication.

[12] Honda, K. and Yoshida, N., Combinatory Representation of Mobile Processes, *Proc. 21st. Annual Symposium on Principles of Programming Languages*, pp.348–360, ACM press, 1994.

[13] Honda, K. and Yoshida, N., Replication in Concurrent Combinators, *TACS '94*, LNCS, Springer-Verlag, 1994.

[14] Honda, K. Kubo, M., and Yoshida, Combinatory Representation of Mobile Processes, in preparation.

[15] Makoto Kubo. π-architecture, *Manuscript*, 1991.

[16] Robin Milner, J. Parrow and David Walker. A calculus of Mobile processes, (Parts I and II). *Information and Computation*, 100:1-77, 1992.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[18] Milner, R., *Communication and Concurency*, Prentice hall, 1989.

[19] Milner, R., Functions as Processes. *Mathematical Structure in Computer Science*, 2(2), pp.119–146, 1992.

[20] Milner, R., Polyadic π-Calculus. *Logic and Algebra of Specification*, Springer-Verlag, 1992.

[21] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Information and Computation* 100(1), pp.1–77, 1992.

[22] Milner, R. and Sangiorgi, D., Barbed Bisimulation. *Proc. of ICALP'92*, LNCS 623, pp.685–695, Springer-Verlag, 1992.

[23] Mitchell, J., Type Systems for Programming Languages, *Handbook of Theoretical Computer Science*, pp.367-458, Elsevier Science Publishers B.V., 1990.

[24] Hanne Riis Nielson and Flemming Nielson. Highr-Order Concurrent Programs with Finite Communication Topology. In *Proceedings of ACM Symposium on Principles of Programming Languages*, to appear, 1994.

[25] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[26] Atsushi Ohori and Kazuhiko Kato. Semantics for Communication Primitives in a Polymorphic Language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1993.

[27] Benjamin C. Pierce, Dider Remy and David N. Turner. A Typed Higher-Order Programming Language Based on the Pi-Calculus. *Manuscript*, 1993.

[28] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *1993 IEEE Symposium on Logic in Computer Science*, 1993

[29] John H. Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 294–305, 1991.

[30] Atsushi Sashino, Makoto Kubo, On Some Interaction Machines, in preparation.

[31] Kaku Takeuchi, *An Interaction-based Language and Its Typing System.* Master Thesis, Keio University, 1993 (in Japanese).

[32] Kaku Takeuchi, et al. *An Interaction-based Language and Its Typing System.* To appear.

[33] Vasco T. Vasconcelos, Typed Concurrent Objects, to appear in *ECOOP '94*, LNCS, Springer-Verlag, 1994.

[34] Vasco T. Vasconcelos, Recursive Types in a Calculus of Objects, Transactions of Information Processing Society of Japan, 1994. To appear.

[35] Nobuko Yoshida, *Optimal Reduction in Weak λ-calculus with Shared Environemnts*, Proc. FPCA '93, ACM Press, 1993.

[36] Yonezawa, A., and Tokoro, M., (ed.) : *Object-Oriented Concurrent Programming.* MIT Press, 1986.