CONCUR'98 と HLCL'98 の報告

sumii

1998 年 9 月 24 日

1 はじめに

今月の 8 日から 12 日にかけてフランスのニースで開かれた、 CONCUR'98 [1] と HLCL'98 [2] にいってきましたので、報告をします。ただ、 CONCUR のほうは発表件数が多く、正直いってほとんど理解できなかったものも多いので (特に verification とか logic とかに関する話題) 、適当に抜粋して紹介したいと思います。

2 CONCUR'98

2.1 Control Flow Analysis for the pi-calculus (Bodei, Degano, Nielson, Nielson)

Scheme などの関数型言語で盛んに行われている control flow analysis を、 π-calculus でやろうという話。基本的なアイディアは非常に簡単で、プログラムの文面において、新たな name が導入される場所 (restriction と input) に marker と呼ばれる印をつけておいて、

- ある name を通して送られる可能性のある値
- ある name に bind される可能性のある値

を、 marker から値の集合への mapping として見積もる。

restriction には $(\nu x^\chi)\ P$ のように、 channel と呼ばれる種類の marker がつけられ、 input には $x(y^\beta).P$ のように、 binder と呼ばれる種類の marker がつけられる。

analysis の結果は

$$(\rho, \kappa) \models_{me} P$$

という形の predicate で表される。 P は対象となるプロセス。その他の記号の意図は次の通り。

me ... P の free name から、それが導入された場所の marker への mapping

ρ ... binder から、それに bind されうる channel の集合への mapping

κ ... channel から、それを通して送られうる channel の集合への mapping

これを formal に定義したのが図 1 のルール。 P に関して syntax-directed かつ compositional になっているので、このルールに素直にしたがえば、プロセス P を与えられたとき、 ρ と κ に関する set constraint を生成できる。この constraint は反復アルゴリズムで簡単に解ける。

analysis の正しさは、 operational semantics (labeled transition system) に関連づけて示されている。応用としては、 name を private なものと public なものにわけ、 private な name が public な name に leak しないことを確かめる、という例が挙げられていた。

[1] 9th International Conference on Concurrency Theory, `http://www.inria.fr/concur98/`

[2] 3rd International Workshop on High-Level Concurrent Languages, `http://www.cs.auc.dk/hlcl98/`

2.2 Minimality and Separation Results on Asynchronous Mobile Processes — Representability Theorems by Concurrent Combinators — (Yoshida)

lambda-calculus が究極的には S と K の二つの combinator だけで実現できるように、*pi*-calculus も有限個の combinator で実現しよう、という話 (の続き)。

parallel composition $P \mid Q$, restriction $(\nu a)\ P$, replication $!\,P$ はそのまま残すが、input $ax\,.\,P$ と output $\overline{a}v$ に関しては、以下の 5 個の combinator (とそれらに injective renaming を行ったもの) が必要十分である。

$$\begin{array}{rlcl}
\mathbf{m}(av) & \text{(message)} & = & \overline{a}v \\
\mathbf{d}(abc) & \text{(duplicator)} & = & ax\,.\,(\overline{b}x \mid \overline{c}x) \\
\mathbf{b_r}(ab) & \text{(right binder)} & = & ax\,.\,\mathbf{fw}(bx) \\
\mathbf{b_l}(ab) & \text{(left binder)} & = & ax\,.\,\mathbf{fw}(xb) \\
\mathbf{s}(abc) & \text{(synchronizer)} & = & ax\,.\,\mathbf{fw}(bc)
\end{array}$$

ただし、 **fw** は以下の通り。

$$\mathbf{fw}(ab) \quad \text{(forwarder)} \quad = \quad (\nu c)\ \mathbf{d}(abc) \quad = \quad (\nu c)\ ax\,.\,(\overline{b}x \mid \overline{c}x) \quad \approx \quad ax\,.\,\overline{b}x$$

「必要」とか「十分」とかは、 π-calculus の二つの subset があったとき、一方に存在するプロセス P に対して、他方に $P \approx Q$ なるプロセス Q が存在するかどうか、という観点から formalize されている。

これらの combinator は一つでも欠けると π-calculus の full set は得られないが、どの combinator をなくすとどのような proper subset が得られるか、という考察がおもしろかった。たとえば、 $\mathbf{b_l}$ をなくすと、local π-calculus ($ax\,.\,P$ の P において x を入力に使えないもの) が得られ、 **d** のかわりに

$$\mathbf{d_1}(abc) = (\nu y)\ ax\,.\,(\overline{b}x \mid \overline{c}y)$$

を用いると、一種の linear π-calculus が得られる。

2.3 From Higher-Order π-Calculus to π-Calculus in the Presence of Static Operators (Vivas, Dam)

Higher-Order π-Calculus というのは、

$$\overline{\mathbf{xfer}}(!\,\overline{a}m) \mid \mathbf{xfer}(P)\,.\,(P \mid a(y)\,.\,Q) \quad \to \quad !\,\overline{a}m \mid a(y)\,.\,Q$$

のように、 channel を通して process も送受信できる計算。これは一見すると普通の π-calculus より強力そうだが、

$$\begin{array}{rl}
 & (\nu b)\ \overline{\mathbf{xfer}}b \mid b\,.\,!\,\overline{a}m \mid \mathbf{xfer}(c)\,.\,(\overline{c} \mid a(y)\,.\,Q) \\
\to & (\nu b)\ b\,.\,!\,\overline{a}m \mid \overline{b} \mid a(y)\,.\,Q \\
\to & !\,\overline{a}m \mid a(y)\,.\,Q
\end{array}$$

とすれば、普通の π-calculus であらわせる (Sangiorgi の D 論)。

しかし、 Higher-Order π-Calculus に blocking を導入した計算は、 Sangiorgi の方法だと普通の π-calculus で正しくあらわせない。 blocking というのは CCS の restriction みたいな operator で、 operational には

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus z \xrightarrow{\alpha} P' \setminus z}\ z \text{ not free } \alpha$$

とあらわされるような効果を持つ (π-calculus の restriction と違って、 z を bind しない) 。たとえば

$$\overline{\mathbf{xfer}}(!\,\overline{a}m) \mid \mathbf{xfer}(P)\,.\,(P \mid a(y)\,.\,Q) \setminus a \quad \to \quad (!\,\overline{a}m \mid a(y)\,.\,Q) \setminus a$$

のように使えば、 a 上の通信を外部から隠すことができる。これを Sangiorgi の方法により普通の π-calculus であらわそうとすると、

$$\begin{array}{rl} & (\nu b)\ \overline{\mathbf{xfer}}b \mid b\,.\,!\,\bar{a}m \mid \mathbf{xfer}(c)\,.\,(\bar{c} \mid a(y)\,.\,Q) \setminus a \\ \to & (\nu b)\ b\,.\,!\,\bar{a}m \mid (\bar{b} \mid a(y)\,.\,Q) \setminus a \\ \to & !\,\bar{a}m \mid (a(y)\,.\,Q) \setminus a \end{array}$$

となってしまい、うまくいかない。

この発表では、送信側ではプロセスを abstract syntax tree の形に encode し、受信側ではそれを interpret するという方法を提示していた。ただ、これはあまりにも naive な方法なので、 π-calculus における higher-order な feature は、もはや単なる syntax sugar とは考えられない、ともいっていた。

2.4 The Tau-Laws of Fusion (Parrow, Victor)

fusion calculus という concurrent calculus における、 weak bisimulation に基づく congruence の axiomization を行った、という話。

fusion calculus というのは、通信が global な影響を及ぼすようなプロセス計算。

$$cx\,.\,P \mid \bar{c}y\,.\,Q \mid R$$

というプロセスを実行すると、 Q の中だけではなく、すべてのプロセスにおいて $x = y$ となる。いいかえると、上のプロセスは fresh な name z に対して

$$(P \mid Q \mid R)\{z/x\}\{z/y\}$$

というプロセスに簡約される。この簡約を fusion といい、 $\{x = y\}$ という action として表す。

fusion を導入するメリットとしては、

- input と output が対称になる
- concurrent constraint を簡単にあらわせる

という点を挙げている（前者はどういう実益があるのかよくわからないが）。

congruence について考える上で、 fusion calculus が π-calculus と異なる点としては、下の二つを挙げていた。

mismatch について。 π-calculus では、

$$[x \neq y]\alpha\,.\,P$$

と

$$[x \neq y]\alpha\,.\,[x \neq y]P$$

は等価である（x, y は α にあらわれないものとする）。しかし fusion calculus では、一個目の mismatch が成り立っていても、だれかが fusion $\{x = y\}$ を行えば、二個目の mismatch は成り立たなくなる。

fusion の合成について。 二つの fusion $\{x = y\}$, $\{u = v\}$ が連続して起こるのは、それらを合成した一つの fusion $\{x = y, u = v\}$ が起こるのと（weak bisimulation では）区別できない。したがって、

$$\{x = y\}\,.\,\{u = v\}\,.\,P \quad \approx \quad \{x = y\}\,.\,\{u = v\}\,.\,P + \{x = y, u = v\}\,.\,P$$

という等価性が成り立つはずである。

これらの相違点に対処するため、 π-calculus における standard な axiom

$$
\begin{array}{lrcl}
\textbf{T1} & \alpha\,.\,\mathbf{1}\,.\,P & = & \alpha\,.\,P \\
\textbf{T2} & P+\mathbf{1}\,.\,P & = & \mathbf{1}\,.\,P \\
\textbf{T3} & \alpha\,.\,(P+\mathbf{1}\,.\,Q) & = & \alpha\,.\,(P+\mathbf{1}\,.\,Q)+\alpha\,.\,Q
\end{array}
$$

のうち、 **T3** を

$$
\begin{array}{lrcl}
\textbf{T3a} & \alpha\,.\,(P+\tilde{M}\mathbf{1}\,.\,Q) & = & \alpha\,.\,(P+\tilde{M}\mathbf{1}\,.\,Q)+\tilde{M}\alpha\,.\,Q \\
\textbf{T3b} & \varphi\,.\,(P+\tilde{M}\psi\,.\,Q) & = & \varphi\,.\,(P+\tilde{M}\psi\,.\,Q)+\tilde{M}(\varphi\wedge\psi)\,.\,Q
\end{array}
$$

の二つに置き換える。ここで、 **1** は空遷移、 $\tilde{M}$ は mismatch の列、 $\varphi\wedge\psi$ は fusion φ, ψ の合成を表す。

2.5 Possible Worlds for Process Algebras (Veglioni, De Nicola)

a を何らかの action として、

$$
\begin{array}{rcl}
P & = & 0 \\
 & & aP \\
 & & P+Q
\end{array}
$$

という形をした non-deterministic なプロセスを、 deterministic なプロセスの集合とみなし、その包含関係でプロセス間の順序や等価性を考えよう、という話。

non-deterministic なプロセスは、 action を入力とみなせば、非決定的オートマトンのように考えられる。そこで、「各状態・各入力に対しどの遷移を選ぶか」をあらかじめすべて決めてしまえば、一つの決定的オートマトンが得られる。そうして得られる決定的オートマトンの集合を、元のプロセス P の possible world と呼び、 $PW(P)$ と書く。

$PW(P) \subseteq PW(Q)$ ならば、 P は Q の possible world refinement であるといい、 $PW(P) = PW(Q)$ ならば、 P と Q は possible world equivalent であるという。このようにして得られる equivalence と、他の様々な equivalence との関係については、図 2, 3 を参照。

2.6 Tutorial: Type Systems for Concurrent Calculi (Pierce)

simple type, I/O type, linear type などのおなじみの話。特に type を導入することにより、 equivalence がより coarse になる点を強調していた。たとえば

```
def fact[n,r] =
  if n=0 then r![1]
  else new c . (fact![n-1,c] | c?[m] . r![n*m])
```

というプログラムを最適化して、

```
def fact[n,r] =
  if n=0 then r![1]
  else if n=1 then r![1]
  else if n=2 then r![2]
  else if n=3 then r![6]
  else new c . (fact![n-1,c] | c?[m] . r![n*m])
```

とすると、 fact から受信を行うといった異常なプロセスにより、違いが観測できてしまう。 fact に output 専用という型をつければ、そのような プロセスは well-typed でないので、上の変換を正当化することができる。

2.7 Lecture: Stochastic Process Algebras: Benefits for Performance Evaluation and Challenges (Herzog)

CCS などの従来の process algebra は、プロセスがどのような action を行うかなどの、システムの機能的な側面をモデル化していた。 Stochastic Process Algebras は、それに加えて性能的な側面もモデル化しようという話。

具体的には、 Markovian timed prefix (a, λ) というものを導入する。直観的な意味は「パラメータ λ の指数分布にしたがう時間をかけて、 a という action を実行する」ということ。プロセスの遷移も $P \xrightarrow{a,\lambda} Q$ のように、時間を含めた形で表す。 (bi)simulation も時間を考慮して定義する。 P, Q, a が与えられたとき、 $P \xrightarrow{a,\lambda} Q$ なる λ が複数存在しうるのが問題だが、そういう場合は指数分布の性質より総和を考えれば良い。公理としては

$$(a, \lambda) . P + (a, \mu) . P \approx (a, \lambda + \mu) . P$$

のように表せる。

多数の研究者の手により理論はすでに出来上がっていて、大規模で実用的なシステムに応用しようという段階に来ているそうだ（本当か？）。ただ、時間の確率分布が指数分布以外の場合については、現在いろいろな研究が進行中らしい。確かに、この Lecture の後の発表でも、時間の確率分布を一般化しようという話が複数あった。

2.8 Tutorial: Checking Verifications of Protocols and Distributed Systems by Computer (Groote, Monin, van de Pol)

非常に簡単な通信プロトコル（SLIP の一部）を例に、並行システムの formal な検証を（concurrent workbench のような専用ソフトウェアではなく）一般的な proof checker によって行う話。特に難しい内容ではなかった。

プロセス計算の operational semantics も普通の定理として与えるので、大規模で複雑なシステムに対しては非常に効率が悪そうだが、論文には効率を良くするための工夫も述べられているようだ。

2.9 Modeling IP Mobility (Amadio, Prasad)

IPv6 に含まれる mobile IP というプロトコルを formalize して、（人間の手で）正しさを検証しました、という内容。具体的には、 barbed bisimulation により、マクロなモデル（仕様）とミクロなモデル（実装）の等価性を証明していた。 SLIP などに比べると、さすがにかなり複雑なプロトコルのようだが、理論的には特に目新しい点はないように思えた。

3 HLCL'98

3.1 Why networks aren't programmable (or, why programs aren't protocols) (White)

なぜ分散言語は普及しないのか、という話。いろいろと理由をあげていたが、 SMTP とか HTTP とかいった client/server model の text based なプロトコルや、 RPC や RMI などのライブラリを用いる方法に比べると、要するに portability や robustness の点で劣るという話だったと思う。

3.2 Resource Access Control in Systems of Mobile Agents (Hennessy, Riely)

以前に後藤さんがサーベイしたテクニカル・レポートとまったく同じ話。

3.3 Distribution and Mobility with Lexical Scoping in Process Calculi (Vasconcelos, Lopes, Silva)

π-calculus のような並行プロセス計算に、 site の概念を追加して分散プロセス計算に拡張する方法の話。

元となる並行プロセス計算のプロセスを P として、

$$
\begin{array}{rcl}
N & ::= & s : P \\
 & & N \parallel N \\
 & & \nu e N \\
 & & \mathbf{0}
\end{array}
$$

という風に syntax を拡張する。ただし、 s は site を、 e は located name（site と name の組）をあらわす。

operational semantics は、 structural congruence を

$$
\begin{array}{rcll}
s : \mathbf{0} & \equiv & \mathbf{0} & \\
s : P_1 \parallel s : P_2 & \equiv & s : (P_1 \mid P_2) & \\
s : \nu a P & \equiv & \nu (s \cdot a)\ (s : P) & \\
N_1 \parallel \nu e N_2 & \equiv & \nu e (N_1 \parallel N_2) & \quad \text{if } e \notin \mathrm{fn}(N_1)
\end{array}
$$

のように、 reduction relation を

$$
r : (s \cdot a) C \to s{:}\, a(C\sigma_{rs})
$$

$$
\frac{P \to Q}{s : P \to s : Q}
$$

のように拡張して得る。ただし $C\sigma_{rs}$ は、プロセス C に現れる name a について、 a は $r : a$ に、 $s : a$ は a に変換したプロセス。

この拡張の元となる言語にいろいろなものがありうるが、 Vasconcelos らは例として、 TyCO という処理系を拡張する実装について説明していた。

3.4 Compiling Join-Patterns (Le Fessant, Maranget)

以前に遠藤さんがサーベイした join calculus の、 essence となる join pattern を実装する方法の話。

たとえば

```
let A(n) | B() = P(n)
and A(n) | C() = Q(n)
;;
```

という join definition を考える。 A, B, C の 3 つのメッセージの有無を考えると、プログラムの状態遷移図は図 4 のようになる。ここで fairness を捨て去って、簡約できる状態になったら即座に簡約することにすると、図 5 を得る。

このオートマトンを実装する方法は二通り述べられていたが、ここではそのうちの一つを紹介する。

コンパイル時には、各状態から可能な各遷移に対応するコードを生成する。実行時には、 name を index とし、message queue と message handler へのポインタを entry とする array が存在する。 handler へのポインタは、そのときの状態に応じて異なったコードを指す。メッセージを送信するときは、引数を queue に入れてから、handler を呼び出す。 handler は、遷移先の状態に応じて handler へのポインタを書き換えてから、（matching が起こっていれば）queue から引数を取り出し、しかるべきプロセスを実行する。

この方法だと状態数やコードサイズの爆発が心配だが、オートマトンにおいていくつかの state を一つにまとめてしまい、そのかわりに runtime のチェックは行うことにより、問題を回避している。

3.5 A Generalized Deadlock-Free Process Calculus (Sumii, Kobayashi)

略。

3.6 A Concurrent Object Calculus: Reduction and Typing (Gordon, Hankin)

去年の冬学期の院の授業でもやった、 Abadi と Cardelli の object calculus に、 parallel composition と mutual exclusion を導入し、そうして得られた calculus の operational semantics や type system を構築したという話。 technical にはわりと straightforward で、特に変わった点や難しいところはなかったように思う。

3.7 λ_S: an implicitly parallel λ-calculus with recursive bindings, synchronization and side effects (Arvind, Maessen, Nikhil, Stoy)

pH ないしは Id という、 Haskell を Parallel に拡張した言語の基礎を得るために、 λ-calculus をいろいろと拡張した計算を作ったという話。と論文には書いてあるのだが、正直いって全体的に内容が複雑でよくわからなかった。特に発表では non-termination の違い (?) について延々と述べていたのだが、何が問題なのか最後まで理解できなかった。[3]

4 おわりに

計 5 日に渡る発表をすべて集中して聞くのは、体力的に無理だということがよくわかりました...。

[3]他の発表と違って質問応答がほとんどなかったので、おそらくみんな理解できなかったんだと思う。