Aπ 計算の Coq による形式化

安武 祥平　渡部 卓雄

アクターモデルは並行計算のモデルのひとつであり，互いに非同期メッセージをやりとりするアクターと呼ばれる計算主体によって計算システムを表現する．アクターモデルのもつ性質には，アクターの名前の一意性と新鮮性，およびアクターの永続性が含まれる．これらの性質を π 計算の型システムとして導入し，アクターモデルを形式化した Aπ 計算が Agha らによって提案されている．A π計算の型システムにおける健全性とは，型付けされた項はアクターモデルとしてのふるまいを示すということであるが，その証明は形式的に与えられていない．本研究では，Aπ 計算およびその型システムの定義をみなおすことで，型システムの健全性について定理証明支援系 Coq を用いた形式的な証明を与える．

The Actor model is a model of concurrent computation based on autonomous computing entities called actors that communicate through asynchronous message passing. The properties of the Actor model include uniqueness and freshness of actor names and persistency of actors. Agha et al. proposed a typed asynchronous variant of π-calculus called Aπ-calculus whose type system imposes these three properties. In Aπ-calculus, the soundness of the type system means that a typed term certainly behaves as an actor configuration. In this paper, by reconsidering the definitions of the terms and types of Aπ-calculus, we give a formal proof of the soundness using Coq.

1　はじめに

アクターモデル [2] は並行計算のモデルのひとつであり，互いに非同期メッセージをやりとりするアクターと呼ばれる計算主体 (computing entity) によって計算システムを表現する．現在，アクターモデルやその発展形である並行オブジェクト計算モデル [12] にもとづく言語は多数提案・実装されてきており，それらの研究成果をもとに現在 Scala や Erlang, Akka 等，アクターモデルを並行計算の基盤としたプログラミング言語やライブラリが実用に供されている．そのため，アクターモデルによって構成されたシステムの形式的検証は喫緊の課題であると考えられる．

形式的検証の研究に先立ち，アクターモデルの形式的意味論については古くから研究が行なわれてきた．例えば Clinger による Powerdomain を用いた表示的意味 [6] や，Agha による操作的意味 [2] とその発展 [3] がある．これらの研究では，公平性やいくつかの等価性についての成果があるものの，プロセス代数を元に発展してきた π 計算等に比べると発展途上であると言わざるを得ない．

一方，形式的検証についてはアクターで記述されたシステムのモデル検査を可能にするモデル記述言語 Rebeca [11]，証明支援系を含むプログラミング言語 Athena を用いた検証例 [10]，および Coq を用いたアクターモデルの定式化と検証例 [8] など，最近になっていくつかの研究成果が出ている．本研究はアクターで構成されたシステムの Coq による形式的検証のための枠組みを提供することを目的としている．

アクターモデルのもつ性質には，アクターの名前は一意であるという性質 (名前の一意性)，メッセージとして送られてきた名前でアクターを作ることはできないという性質 (名前の新鮮性)，アクターにはいつでもメッセージを送ることができるという性質 (ア

A formalization of Aπ-calculus using Coq
Shohei Yasutake, 東京工業大学大学院情報理工学研究科, Dept. of Computer Science, Tokyo Institute of Technology.

クターの永続性) が含まれる．これらの性質をみたす項を構成するよう π 計算に型システムを導入し，アクターモデルを形式化した Aπ 計算が Agha らによって提案されている [4].

Aπ 計算の型システムにおける健全性とは，型付けされた項はアクターモデルとしての振る舞いを示すということであるが，その証明は形式的に与えられていない．本研究では，Aπ 計算およびその型システムの定義を見直すことで，型システムの健全性について定理証明支援系 Coq を用いた形式的な証明を与える．また，本研究は Aπ 計算自体の正しさの検証ということも目的としているため，Aπ 計算の定義は等価なものへの変更でない限り変更しないものとする．

2 背景知識

2.1 アクターモデル

アクターモデルは非同期メッセージ通信に基づいた並行計算のモデルである．アクターと呼ばれる計算主体があり，それらは名前 (またはアドレス) と内部状態をもっている．各アクターは並行に動作し，非同期にメッセージを送り合うことでコミュニケーションをとる．

アクターは，メッセージの内容に応じて一定の動作を行う．これを振る舞い (behavior) という．振る舞いは以下のアクションの組み合わせで表される．

- 他のアクターにメッセージを送信する．
- 新しいアクターを作る．
- 自らの振る舞いを変える．

2.1.1 配置

アクターモデルにおける配置 (configuration) とは，アクターモデルの世界におけるその時点での状態を切り取ったものを表すための概念である．配置はアクターとその振る舞い，まだ受け取られていないメッセージの集合から表される．

配置外からのメッセージを受け取ることのできる配置内のアクターを，窓口 (receptionist) という．アクターは，送り先の名前 (アドレス) を知ることで，その名前のアクターにメッセージを送れるようになる．よって窓口は外部のアクターに自身の名前を知られているアクターと言い換えることができる．また，窓口の集合のことを窓口集合 (receptionist set) という．

2.1.2 アクターモデルの性質

アクターモデルが満たすべき性質として，以下がある．これらの性質はアクターの名前に関するものであり，これらが満たされない場合は，同じ名前を持つ複数のアクターができることやアクターの名前が変わること，消えることがありえるため，アクターモデルとしての一貫性が崩れてしまう．

一意性 (uniqueness property)
: アクターの名前は一意である．

新鮮性 (freshness property)
: アクターが作られるとき，作られるアクターの名前はまだどのアクターの名前でもない．アクターは，メッセージとして送られてきた名前でアクターを作ることはできない．

永続性 (persistence property)
: アクターは消えない．一旦アクターが作られると，いつでもそのアクターにメッセージを送ることができる．

2.2 定理証明支援系 Coq

Coq はフランス国立情報学自動制御研究所で開発されている定理証明支援系である [7]．Coq を用いることで，プログラムがある仕様を満たすということや，数学的な定理などに対して形式的かつ厳密な証明を与えることができる．Coq によって証明されたものとしては，四色定理が有名である [9].

3 Aπ 計算

本節では，アクターモデルの振る舞いを強制するために非同期 π 計算に型を付けた Aπ 計算について説明する．まず Aπ 計算における配置の定義をする．次に型システムを導入するが，その際に仮名関数という関数を導入し，型付け規則を定義する．最後にこの型システムにおける健全性について述べる．また，Aπ 計算の定義および使われている記号は，文献 [4] で用いられているものを用いる．

3.1 記号の定義

これ以降で用いる記号の定義をする．

定義 3.1 (組)**.** $\langle x_1, x_2, ..., x_n \rangle$ と書いたとき，これを $x_1, x_2, ..., x_n$ の組 *(tuple)* とする．単に $\langle\rangle$ と書いたときは要素がない組である．$\tilde{x}$ は名前の組であることを表す．$\{\tilde{x}\}$ と書いたときは，要素として $\tilde{x}$ のみを持つ集合ではなく，$\tilde{x}$ に含まれる名前の集合を表す．また，組を連結する二項演算子を，$\tilde{+}$ と定義する．

定義 3.2 (要素数が 0 または 1 の集合)**.** 空集合であるか，または要素が z のみである集合を $\hat{z}$ と書く．$\langle\hat{z}\rangle$ と書いたときは $\hat{z}$ 一つの組ではなく，$\hat{z} = \emptyset$ であるときは $\langle\rangle$ ，$\hat{z} = z$ であるときは $\langle z \rangle$ とする．

3.2 配置

Aπ 計算における配置の文法は，図 1 のようになっている．

$$
\begin{aligned}
P ::= {} & 0 \mid x(y).P \mid \bar{x}y \mid (\nu x)P \mid P_1|P_2 \\
& \mid \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, ..., y_n : P_n) \\
& \mid B\langle \tilde{x}; \tilde{y} \rangle
\end{aligned}
$$

図 1: Aπ 計算の配置の文法

それぞれの説明を以下に示す．

0	空の配置であることを表す．
$x(y).P$	$(y).P$ という振る舞いを持つアクター x をつくる．ここで振る舞いとは，y というメッセージを受け取って P を実行するという意味である．
$\bar{x}y$	x という名前のアクターに y という名前を送る．
$(\nu x)P$	P 中に現れる x という名前をこの配置の外から隠す．
$P1\|P2$	$P1$ と $P2$ からなる配置であることを表す．
$\texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, ..., y_n : P_n)$	名前 x が y_i と一致したとき P_i になる．一致するものが複数ある場合は，そのなかから非決定的に選ばれたものになる，一致するものがない場合は 0 になる．
$B\langle \tilde{x}; \tilde{y} \rangle$	振る舞いの雛形 B からアクターをつくる．$B \stackrel{\text{def}}{=} (\tilde{x}, \tilde{y})x_1(z).P$ であり，$\tilde{x}$ という名前の組と，$\tilde{y}$ という名前の組を受け取り，$\tilde{x}$ の第一要素 x_1 を名前とし，メッセージ v を受け取って P を実行するようなアクターを返す関数であることを意味する．また，$\tilde{x}$ の要素数は 1 または 2 とし，この意味は 3.4 節で示す．

3.3 型システムの導入

Aπ 計算の文法のみでは，アクターモデルの性質は満たさないような配置も書けてしまう．例えば，$x(u).P|x(v).Q$ は Aπ 計算の文法で書けるが，x という名前のアクターが 2 つ作られるので，一意性を満たしていない．また，$x(u).(u(v).P|x(v).Q)$ も Aπ 計算の文法で書けるが，メッセージとして送られてきた名前でアクターが作られているので，新鮮性に反する．

また，アクターモデルでは，1 回の通信で複数の名前を送信することができるが，Aπ 計算では 1 回の通信で一つの名前しか送ることはできない．複数の名前を送信するには，その数だけ連続して通信する必要がある．しかし，アクターモデルではこのような複数の名前を含むメッセージの受け渡しはアトミックに行なわれ，即座に次のメッセージを受け取れる状態にならなければいけない．そこで，いつでもメッセージを送信できるという永続性を弱め，アクターはメッセージの送信先のアクターの処理が終わるまで待てるようにする (いつかは必ず送れる)．そしてアトミックに行うことのできない処理の最中はメッセージを送れないということを，正規の名前を隠して一時的な名前を使って処理を行うことで実現させる．ここで，仮の名前をとった後，元の名前とは異なる名前になってしまうと，そのアクターにメッセージを送ろうとしているアクターはメッセージを送れなくなってしまい，永続性に反する．仮の名前をとったときには必ず元の名前に戻るという制約が必要である．

以上のような性質を満たすようにするため，型システムを導入し，Aπ 計算に制約を与える．

3.3.1 仮名関数

仮の名前は必ず正規の名前に戻るという制約を型システムを用いて与えるためには，仮の名前をとったときに正規の名前を覚えておく必要がある．そこで仮の

名前から元の正規の名前にマッピングする関数を導入する．これを 仮名関数 (temporary name mapping function) と呼ぶ．

定義 3.3 (仮名関数)．仮名関数 f は，集合 X から集合 X^* への写像である．ただし，$\mathcal{N}$ を名前全体の集合として，$\bot, * \notin \mathcal{N}, X \subset \mathcal{N}, X^* = X \cup \{\bot, *\}$ である．

仮名関数は以下の通り定義される．

$f(x) = \bot$	x が正規の名前のとき
$f(x) = *$	x が環境から隠されている名前であるとき
$f(x) = y \notin \{\bot, *\}$	アクター y の仮の名前が x であるとき

また，$f^* : X^* \to X^*$ を以下のように定義する．

$$f^*(x) = \begin{cases} f(x) & x \in X \text{ のとき} \\ \bot & x = * \text{ または } x = \bot \text{ のとき} \end{cases}$$

3. 3. 2　仮名関数の構築

仮名関数は以下に示す ch 関数，仮名関数の合成，仮名関数の制限のいずれかで作ることができる．

定義 3.4 (ch)．ch は名前の組をとり，仮名関数を返す関数である．組の長さは任意である．組を $\langle x_1, x_2, ..., x_n \rangle$ としたとき，$ch(\langle x_1, x_2, ..., x_n \rangle)$ は以下のような仮名関数となる．

$$ch(\langle x_1, x_2, ..., x_n \rangle)(x_i) = \begin{cases} x_{i+1} & 1 \leq i < n \text{ のとき} \\ \bot & i = n \text{ のとき} \end{cases}$$

定義 3.5 (仮名関数の合成)．$f_1 : \rho_1 \to \rho_1^*$，$f_2 : \rho_2 \to \rho_2^*$ とする．仮名関数の合成を表す二項演算子 $f_1 \oplus f_2 : \rho_1 \cup \rho_2 \to (\rho_1 \cup \rho_2)^*$ を以下のように定義する．

$$(f_1 \oplus f_2)(x) = \begin{cases} f_1(x) & x \in \rho_1 \text{かつ } f_1(x) \neq \bot, \text{ または} \\ & \text{は } x \in \rho_1 \text{かつ } x \notin \rho_2 \text{のとき} \\ f_2(x) & \text{その他} \end{cases}$$

定義 3.6 (仮名関数の値域の制限)．$f : \rho_1 \to \rho_1^*, \rho \subset \rho_1$ である f, ρ に対して値域の制限を表す二項演算子 $f|\rho : \rho \to \rho^*$ を以下のように定義する．

$$(f|\rho)(x) = \begin{cases} f(x) & f(x) \in \rho \text{のとき} \\ * & \text{その他} \end{cases}$$

3. 3. 3　仮名関数が満たすべき性質

仮名関数 f はその定義から，以下を満たすべきである．

- $f(x) \neq x$
- $f(x) = f(y) \notin \{\bot, *\} \Rightarrow x = y$
- $f^*(f(x)) = \bot$

$f(x) \neq x$ は，仮の名前から正規の名前を得たとき，それは仮の名前と同名ではないという性質である．$f(x) = f(y) \notin \{\bot, *\} \Rightarrow x = y$ は，正規のアクターは 2 個以上の仮の名前をとれないという性質である．$f^*(f(x)) = \bot$ は，仮名のアクターはさらに仮の名前を持つことはできないという性質である．

また，互換性という性質を定義する．

定義 3.7 (互換性 (compatible))．二つの仮名関数 f_1, f_2 が以下の性質を満たすとき，f_1, f_2 は互換性を持つという．また，この性質を満たすという述語を $compatible(f_1, f_2)$ と書く．

- $f_1 \oplus f_2 = f_2 \oplus f_1$
- $f_1 \oplus f_2$ が仮名関数の性質を満たす

3. 4　型付け規則

Aπ 計算が先に述べた性質を満たすようにするための型付け規則は図 2 のようになる．配置 P，P の窓口集合 ρ，P 中の仮の名前から正規の名前への写像 $f : \rho \to \rho^*$ で，$\rho; f \vdash P$ が成り立つとき，P は正しく型付けされているという．

それぞれの規則についての説明を以下に示す．

$$\frac{}{\emptyset;\{\}\vdash 0}\ (\textsc{Nil})$$

$$\frac{}{\emptyset;\{\}\vdash \bar{x}y}\ (\textsc{Msg})$$

$$\frac{\rho;f\vdash P \qquad \rho-\{x\}=\hat{z} \qquad y\notin\rho \qquad f=\begin{cases} ch(\langle x\rangle\tilde{+}\langle\hat{z}\rangle) & x\in\rho\text{のとき} \\ ch(\langle\hat{z}\rangle) & \text{その他}\end{cases}}{\{x\}\cup\hat{z};ch(\langle x\rangle\tilde{+}\langle\hat{z}\rangle)\vdash x(y).P}\ (\textsc{Act})$$

$$\frac{\forall 1\le i,j\le n, i\neq j,\ \rho_i;f_i\vdash P_i \qquad compatible(f_i,f_j)}{(\cup_i\rho_i);(f_i\oplus f_2\oplus ...\oplus f_n)\vdash \texttt{case}\ x\ \texttt{of}\ (y_1:P_1,...,y_n:P_n)}\ (\textsc{Case})$$

$$\frac{\rho_1;f_1\vdash P_1 \qquad \rho_2;f_2\vdash P_2 \qquad \rho_1\cap\rho_2=\phi}{\rho_1\cup\rho_2;f_1\oplus f_2\vdash P_1|P_2}\ (\textsc{Comp})$$

$$\frac{\rho;f\vdash P}{\rho-\{x\};f|(\rho-\{x\})\vdash(\nu x)P}\ (\textsc{Res})$$

$$\frac{\{\tilde{x}\};ch(\tilde{y})\vdash B\langle\tilde{x};\tilde{y}\rangle \qquad len(\tilde{x})=2\ \Rightarrow\ x_1\neq x_2}{\{\tilde{x}\};ch(\tilde{x})\vdash B\langle\tilde{x};\tilde{y}\rangle}\ (\textsc{Inst})$$

図 2: 型付け規則

Nil
: 空の配置に関する型付け規則である．0 中にはアクターは存在しないので，窓口集合は空集合，仮名関数は空関数で型付けできる．

Msg
: メッセージの送信についての型付け規則である．$\bar{x}y$ 中にはアクターは存在しないので，Nil と同様に窓口集合は空集合，仮名関数は空関数で型付けできる．

Act
: 新しくアクターを作る構文についての型付け規則である．$y\notin\rho$ は，メッセージの名前でアクターを作れないということを表しているため，これがあることで新鮮性が保証される．また，$\hat{z}=\{z\}$ であるときは，z は x を一時的な名前としてとっていることを表しており，$\hat{z}=\emptyset$ であるときは，x がアクターの正規の名前であることを表している．さらに，$\hat{z}=\emptyset$ かつ $x\notin\rho$ の場合はその x というアクターは消えるが，Aπ 計算では x という名前のアクターはメッセージを受け取っても何もしないアクターになると見なす．こうすることで，永続性を満たすようにする．

Case
: 名前について場合分けを行う際の型付け規則である．相互に互換性を持つという条件によって，一つのアクターが複数の仮の名前を持つということが起こらないことを保証している．

Comp
: P_1 と P_2 からなる配置を作る構文の型付け規則である．$\rho_1\cap\rho_2=\phi$ から，一意性が保証される．

Res
: 名前を外の環境から隠す構文の型付け規則である．P 中に現れるアクター x を外部から隠すので，窓口集合から $\{x\}$ を引き，仮名関数も制限している．

Inst
: 振る舞いの雛形からアクターを作る構文についての型付け規則である．$len(\tilde{x})=2$ であるときは，x_1 は x_2 の仮の名前であることを表している．また，$x_1\neq x_2$ という条件によって，仮の名前が正規の名前と同名にならないようにしている．$len(\tilde{x})=1$ であるときは，x_1 は正規の名前である．

3.5 型システムの健全性

Aπ 計算の型システムにおける健全性は，この型システムを満たした Aπ 計算の配置はアクターモデルとしての振る舞いを示すということを表した性質である．単純型付き λ 計算における型システムの健全性のような，正しく型付けされるならば stuck 状態にならないという性質とは少し異なることに注意されたい．

Aπ 計算の型システムの健全性は以下のように定義されている．

定理 3.1 (Aπ 計算の健全性)**.** 配置 P について，

$\rho; f \vdash P$ で型付けできたとき，以下の 3 つの性質を満たす．

- ρ は P 中に現れる自由名の部分集合である (窓口集合の健全性)
- 以下の 3 つを満たす (仮名の取り方の健全性)
  1. $f(x) \neq x$
  2. $f^*(f(x)) = \bot$
  3. $f(x) = f(y) \neq \{\bot, *\} \Rightarrow x = y$
- $\rho'; f' \vdash P$ ならば， $\rho = \rho'$ かつ $f = f'$ (型付けの一意性)

それぞれは以下の意味を持っている．

窓口集合の健全性

窓口集合に属するアクターはメッセージとして送られてきた名前で作られたものではなく，かつ環境から隠されたものでもない．

仮名の取り方の健全性

アクターが仮の名前をとったとき，それは同名のものではなく，さらに仮の名前を持つこともない．また，アクターが複数の仮の名前を持つことはない．

型付けの一意性

配置に型が付いたとき，その型付けは一意である．

4 Coq による形式化

本節では，3 節で述べた Aπ 計算についてどのように Coq で記述していくかについて説明する．まず仮名関数およびその性質についての Coq での表現を述べる．次に型付け規則を Coq で使いやすくための規則の具体化を説明し，最後に型システムの健全性の Coq での証明方針を述べる．また，Coq による定義と証明の全文は GitHub リポジトリ [5] に公開している．

4.1 仮名関数

仮につけた名前から正規の名前を得る 仮名関数の Coq での定義について述べる．

仮名関数 f の型は $X \to X^*$ であり，定義域と値域の集合は $\bot$ と $*$ を除くと等しくなっている．Coq による形式化では簡単にするためにその情報を捨て，任意の `name` 型の値から任意の `option star` 型への関数とした．定義域の範囲に入っていない入力は `None` を返すことを想定している．また，ここで `star` 型は `name` 型に $\bot$ と $*$ という値を加えたような型である．

この変更により，以下の述語が必要となる．$f : X \to X^*$ ということを，$in\_range\_in\_domain(f)$ を満たすことで表している．

定義 4.1 (in_domain)**.** 名前 x ，仮名関数 f について，$x \in domain(f)$ を $in\_domain(f, x)$ で表す．ただし $domain(f)$ は f の定義域である．

定義 4.2 (in_range)**.** 名前 x ，仮名関数 f について，$x \in range(f)$ を $in\_range(f, x)$ で表す．ただし $range(f)$ は f の値域である．

定義 4.3 ($in\_range\_in\_domain$)**.** ある 仮名関数 f が，任意の名前 x について $in\_range(f, x) \Rightarrow in\_domain(f, x)$ を満たすということを $in\_range\_in\_domain(f)$ で表す．

また，2 つの 仮名関数の定義域が重なっていないことを表す述語 $fun\_exclusive$ を定義する．

定義 4.4 ($fun\_exclusive$)**.** 2つの 仮名関数 $f : X \to X^*, g : Y \to Y^*$ について，$X \cap Y = \emptyset$ であるという述語を $fun\_exclusive$ と定義する．

これらの定義は Coq による形式化の `Fun` モジュールで定義されている．

4.1.1 仮名関数の構築

仮名関数は ch 関数，仮名関数の合成，仮名関数の値域の制限，のいずれかで作ることができる．ここで，ch 関数が使われている型付け規則は Act のみであるが，この中で使われている ch 関数は入力の要素数が 0 個，1 個，2 個のものしかない．よって，ch 関数はこの 3 パターンに限定でき，それぞれ ch_0, ch_1, ch_2 と書くこととする．また，仮名関数の値域の制限を行う二項演算子 (|) が現れるのは Res のみで，かつそれはもとの仮名関数の値域からあるひとつの要素を取り去るものとしてしか使われていない．後の証明を簡単にするために，この演算子を，制限後の集合ではなく値域から取り去る要素を右手にとるものとして定義する．ただし，この演算子はもとの演算子 (|) とは定義が異なるため，混乱を避けるために別の記号 (\) を用いることにする．

定義 4.5 (仮名関数の値域の制限)**.** $f : \rho \to \rho^*, x \in \rho$

である f, x に対して値域の制限を表す二項演算子 $f \setminus x : \rho - \{x\} \to (\rho - \{x\})^*$ を以下のように定義する．

$$(f \setminus x)(y) = \begin{cases} * & f(y) = x \text{ のとき} \\ f(y) & \text{その他} \end{cases}$$

また，$ch_0, ch_1, ch_2, \oplus, \setminus$ について仮名関数の性質を満たすことを証明した．その際，ch_2 は引数である 2 つの名前が異なることを，$\oplus$ は $in\_domain\_in\_range, fun\_exclusive$ が成り立つという前提を加えている．

4.2 型付け規則

Aπ 計算の型付け規則には，Coq 上で定義しにくい，またはそのままでは証明を進めにくいものがある．実際に Coq で型付け規則を定義する前に，型付け規則を再定義する．

4.2.1 Act 規則の分割

型付け規則 Act はアクターが作られる際に考えられるパターンを複数まとめたものであり，このままの定義を用いると証明が煩雑になってしまう．これは，Act に関する証明を行う際に，ch 関数の定義にともなって Act に現れる ρ の要素数で場合分けをすることになるが，その都度考えられない場合が現れ，その場合は矛盾を証明する，ということをしなければならないためである．また，Coq では任意の要素数を持つ集合を扱うような証明は煩雑になってしまいやすい．そこで，事前に ρ の要素数によって場合分けを行い，Act を分割しておくことを考える．ρ の場合分けを行うと，以下のように 4 つに分けられることがわかる．

1. 要素数が 0 のとき
   $\rho - \{x\} = \hat{z}$ から，$\hat{z} = \emptyset, f = ch_0$ である．
2. 要素数が 1 のとき
   (a) $x \in \rho$ のとき
   $\rho - \{x\} = \hat{z}$ で $len(\rho) = 1$ なので，$\hat{z} = \emptyset, f = ch_1(x)$ である．
   (b) $x \notin \rho$ のとき
   $\rho - \{x\} = \hat{z}$ であるから，$len(\hat{z}) = 1$ である．$\hat{z} = \{z\}$ とすると，$f = ch_1(z)$ である．
3. 要素数が 2 のとき
   $\rho - \{x\} = \hat{z}$ であり，$\hat{z}$ は要素数が 0 または 1 なので，$x \in \rho$ である．$\hat{z} = \{z\}$ とすると，$f = ch_2(x, z)$ である．
4. 要素数が 3 以上のとき
   $3 \leq len(\rho)$ のとき，$\hat{z}(= \rho - \{x\})$ は要素数が 2 以上である．これは $\hat{z}$ の定義と矛盾する．よって要素数が 3 以上であることはない．

この場合分けをもとに，Act を 4 つに分割すると，図 3 のような定義にできる．Act-empty は，$\rho = \emptyset$ の場合の型付け規則である．メッセージを受け取ったあと何らかの処理をし，その後はメッセージを受け取っても何もしないアクターになるようなアクターの型付け規則となっている．Act-x は，$\rho = x$ の場合の型付け規則である．x は正規の名前で，メッセージを受け取って何らかの処理をした後，また x という名前でメッセージの待ち状態になるようなアクターの型付け規則となっている．Act-z は，$\rho = z$ の場合の型付け規則である．x という仮の名前をとって何らかの処理をした後，正規の名前 z をとるようなアクターの型付け規則となっている．Act-xz は，$\rho = x, z$ の場合の型付け規則である．x という仮の名前をとって何らかの処理をした後さらに同じ仮の名前 x をとるような，z を正規の名前とるアクターの型付け規則となっている．

4.2.2 Case 規則の分割

型付け規則 Case は，前提の数がパターンの数に依存して変動するので，Coq で定義しにくい．よって 図 4 のように Case をパターンが 0 個の場合と $i + 1$ に分割し，帰納的に定義する．これで，Case 規則の前提の数はパターンの数によらず一定となる．Case-nil は `case` 式が空であるような型付け規則で，パターンがないので，空集合と空関数で型付けできる．Case-cons は `case` 式のパターンを一つ追加するような型付け規則である．この際，同じ仮の名前が `case` の枝に現れないようにするために，互換性が必要となる．

また，この定義によって作られる型付け $(...((\rho_1 \cup \rho_2) \cup \rho_3)... \cup \rho_n) \cup \rho; (...((f_1 \oplus f_2) \oplus f_3)... \oplus$

$$\frac{\emptyset; ch_0 \vdash P}{\{x\}; ch_1(x) \vdash x(y).P} \text{ (Act-empty)}$$

$$\frac{\{x\}; ch_1(x) \vdash P \qquad x \neq y}{\{x\}; ch_1(x) \vdash x(y).P} \text{ (Act-x)}$$

$$\frac{\{z\}; ch_1(z) \vdash P \qquad x \neq z \qquad y \neq z}{\{x, z\}; ch_2(x, z) \vdash x(y).P} \text{ (Act-z)}$$

$$\frac{\{x, z\}; ch_2(x, z) \vdash P \qquad x \neq z \qquad x \neq y \qquad y \neq z}{\{x, z\}; ch_2(x, z) \vdash x(y).P} \text{ (Act-xz)}$$

図 3: 型付け規則 Act の分割

$$\frac{}{\emptyset; \{\} \vdash \texttt{case}\ x\ \texttt{of}\ ()} \text{ (Case-nil)}$$

$$\frac{\rho; f \vdash \texttt{case}\ x\ \texttt{of}\ (y_1 : P_1, ..., y_i : P_i) \qquad \rho'; f' \vdash P \qquad compatible(f, f')}{\rho \cup \rho'; f \oplus f' \vdash \texttt{case}\ x\ \texttt{of}\ (y : P, y_1 : P_1, ..., y_i : P_i)} \text{ (Case-cons)}$$

図 4: 型付け規則 Case の分割

$f_n) \oplus f \vdash \texttt{case}\ x\ \texttt{of}\ (y : P, y_1 : P_1, ..., y_n : P_n)$ において，元の定義では $f, f_1, f_2, ..., f_n$ が相互に互換性を持つことを条件としているが，$f, f_1, ..., f_n$ が相互に互換性を持つことは自明ではない．よって以下の補題が必要となる．

補題 4.1. 仮名関数 f, f_1, f_2 について，f と $f_1 \oplus f_2$ が互換性を持ち，f_1 と f_2 もまた互換性を持つとき，f, f_1, f_2 は相互に互換性を持つ．

この補題の Coq による証明は `Fun` モジュールの `fun_plus_compatible` という名前にあるが，以下にその概略を示す．

証明. まず f と f_1 が $\oplus$ について可換であること，f と f_2 が $\oplus$ について可換であることを，f と $f_1 \oplus f_2$ が互換性を持ち f_1 と f_2 もまた互換性を持つという仮定と，関数の外延的同値 (後述) を用いることで，

$$\frac{\rho; f \vdash P}{\rho - \{x\}; f \setminus x \vdash (\nu x)P} \text{ (Res)}$$

図 5: 型付け規則 Res

示すことができる．このことから，f，f_1，f_2 が相互に可換になることがわかる．

次に，f，f_1，f_2 の任意の組み合わせの合成が仮名関数の性質を満たすことを証明する．まずは f と f_1 の合成が仮名関数の性質を満たすことを証明する．f と $f_1 \oplus f_2$ が互換性を持つという仮定から，$f \oplus (f_1 \oplus f_2)$ が仮名関数の性質を満たす．また仮名関数の合成は結合律が成り立つので $(f \oplus f_1) \oplus f_2$ も仮名関数の性質を満たす．合成が仮名関数の性質を満たしているならば合成の左手は仮名関数の性質を満たす，という性質 (`fun_prop_plus_fst` という名前で証明済み) を，先に証明した $(f \oplus f_1) \oplus f_2$ も仮名関数の性質を満たすということに使うと，$f \oplus f_1$ が仮名関数の性質を満たすということを導ける．f と f_2 についても同様である．また，f_1 と f_2 については仮定から明らかである．

以上から，f，f_1，f_2 の任意の組み合わせの合成が仮名関数の性質を満たすことがわかったので，f，f_1，f_2 が相互に可換になることと合わせると，f，f_1，f_2 は相互に互換性を持つことがわかる． □

この補題 4.1 によって，$f, f_1, ..., f_n$ が相互に互換性を持つことが帰納的に保証される．

また，補題 4.1 で用いた関数の外延的同値は以下のような公理である．これは Coq の標準ライブラリで提供されているものを使用した．

公理 4.1 (関数の外延的同値)**.** 関数 f, g が任意の入力に対して等しい結果を返すならば f, g は等しい．

4.2.3 Res 規則

仮名関数の値域の制限を行う二項演算子の定義を変更したことに合わせて，型付け規則 Res は 図 5 のようになる．これは 図 2 から演算子を (
) 変えただけなので，意味としては変わらない．

$$\frac{\{u\}; ch_1(u) \vdash u(z).P}{\{x\}; ch_1(x) \vdash ((\langle u\rangle, \tilde{v})u(z).P)(\langle x\rangle, \tilde{y})} \text{ (INST-1)}$$

$$\frac{\{u_1, u_2\}; ch_2(u_1, u_2) \vdash u_1(z).P \qquad x_1 \neq x_2}{\{x_1, x_2\}; ch_2(x_1, x_2) \vdash ((\langle u_1, u_2\rangle, \tilde{y})u_1(z).P)(\langle x_1, x_2\rangle, \tilde{y})} \text{ (INST-2)}$$

図 6: 型付け規則 INST の分割

4.2.4 Inst 規則の分割

Coq による定義では振る舞いの雛形からアクターを作る配置 $B\langle\tilde{x};\tilde{y}\rangle(B \stackrel{\text{def}}{=} (\tilde{u},\tilde{v})u_1(z).P)$ を，$\tilde{u}$ の要素数が 1 である場合と 2 である場合で二つに分割している．よって型付け規則も二つに分割する必要があり，図 6 のようになる．ただし，ここでは B を使う代わりに B の展開後の項を用いている．INST-1 は，$\tilde{u}$ の要素数が 1 である場合で，振る舞いの雛形から x という正規の名前で作るような型付け規則である．INST-2 は，$\tilde{u}$ の要素数が 2 である場合で，振る舞いの雛形から x_1 という仮の名前，x_2 という正規の名前でアクターを作るような型付け規則である．

4.3 健全性の証明

最後に健全性の証明を行うが，以下の 3 つの補題を証明しておくと，健全性の証明を行いやすいので証明しておく．対応する Coq による証明は，`Typing.v` ファイルの `typing_in_range_in_domain`, `typing_in_domain_1` および `typing_in_domain_2`, `Soundness.v` ファイルの `typing_fun_exclusive` にある．

補題 4.2 (定義域と値域の一致)．$\rho; f \vdash P$ ならば，$in\_range\_in\_domain(f)$ が成り立つ．

補題 4.3 (仮名関数の定義域と窓口集合の一致)．$\rho; f \vdash P$ ならば，任意の名前 x について $in\_domain(f, x)$ であるとき，かつそのときに限り，$x \in \rho$ が成り立つ．

補題 4.4 ($fun\_exclusive$ における集合の一致)．$\rho_1; f_1 \vdash P_1$ かつ $\rho_2; f_2 \vdash P_2$ であり $\rho_1 \cap \rho_2 \neq \emptyset$ ならば，$fun\_exclusive(f_1, f_2)$ を満たす．

健全性は，以上の公理と補題を利用して，型付け規則の構造に関する帰納法で証明できる．対応する Coq の証明は `Soundness.v` ファイルの `Soundness` にある．

5 まとめと今後の課題

本論文では，Aπ 計算における型システムの定義を Coq で証明を行いやすいように再定義し，Aπ 計算の型付けの健全性を形式的に証明した．これによって Aπ 計算の型システムが確かにアクターとしての振る舞いを強制することを示すための第一歩を示した．

Aπ 計算の型システムが確かにアクターとしての振る舞いを強制するということを示すためには，さらに型保存性の証明が必要である．Aπ 計算の操作的意味論は π 計算と同様にラベル付き遷移 (labeled transition system) を用いており，Aπ 計算における型保存性は Aπ 計算の正しく型がついた配置は遷移を行っても正しく型がつく，というものである．この証明がない場合は，遷移の際に型付けできないような Aπ 計算の項になってしまう，つまりアクターとしての性質が成り立たないような項になる可能性が残ってしまう．よってこの操作的意味論の Coq での形式化および型保存性の証明が課題として残っている．

また，この Aπ 計算計算の形式化をアクターの形式検証の土台として用いるためには，Aπ 計算の項の性質を表すための言語内 DSL を Coq 内で構築する必要がある．この手法は π 計算を Coq で形式化し，π 計算の検証ライブラリとした applpi [1] が用いており，π 計算ベースである Aπ 計算にも同様に導入できると考えられる．

謝辞 本研究の一部は JSPS 科研費 24500033 の助成を受けている．

参 考 文 献

[1] Affeldt, R. and Kobayashi, N.: A Coq Library for Verification of Concurrent Programs, *Electronic Notes in Theoretical Computer Science*, Vol. 199, No. 0(2008), pp. 17 – 32. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

[2] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[3] Agha, G., Mason, I. A., Smith, S. F., and Talcott, C.: A foundation for actor computation, *Journal of Functional Programming*, Vol. 7, No. 1(1997),

pp. 1–72.

[4] Agha, G. and Thati, P.: An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language, *From Object-Orientation to Formal Methods*, Lecture Notes in Computer Science, Vol. 2635, Springer-Verlag, 2004, pp. 26–57.

[5] A formalization of Aπ-calculus, `https://github.com/amutake/a-pi`.

[6] Clinger, W. D.: Foundations of Actor Semantics, Technical Report AITR-633, AI Lab., MIT, 1981.

[7] The Coq Proof Assistant, `http://coq.inria.fr/`.

[8] Garnock-Jones, T., Tobin-Hochstadt, S., and Felleisen, M.: The Network as a Language Construct, *Programming Languages and Systems (ESOP 2014)*, Lecture Notes in Computer Science, Vol. 8410, Springer-Verlag, 2014, pp. 473–492.

[9] Gonthier, G.: A computer-checked proof of the Four-Colour Theorem, 2004.

[10] Musser, D. R. and Varela, C. A.: Structured Reasoning About Actor Systems, *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, ACM, oct 2013, pp. 37–48.

[11] Sirjani, M. and Jaghoori, M. M.: Ten Years of Analyzing Actors: Rebeca Experience, *Formal Modeling: Actors, Open Systems, Biological Systems*, Agha, G., Danvy, O., and Meseguer, J.(eds.), Lecture Notes in Computer Science, Vol. 7000, Springer-Verlag, 2011, pp. 20–56.

[12] Yonezawa, A., Briot, J.-P., and Shibayama, E.: Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA '86 Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, 1986, pp. 258–268.