社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

分散 JoinJAVA プログラムの通信エラーに対する型判定システム

佐伯　昌樹†　坂部　俊樹††　酒井　正彦††　草刈　圭一朗††　西田　直樹††

†, †† 名古屋大学大学院情報科学研究科
〒 464-8603 名古屋市千種区不老町
E-mail: †saeki@sakabe.i.is.nagoya-u.ac.jp, ††{sakabe,sakai,kusakari,nishida}@is.nagoya-u.ac.jp

あらまし　分散 JoinJAVA は Join 算法を JAVA の記述が出来るよう拡張した言語である．JAVA プログラムでは複雑となるスレッド間の通信，同期を Join 算法の記述を用いることで簡潔に表現可能となる．Join 算法ではプロセス間の通信が正常に実行されることを保証する型判定システムが与えられている．本稿では，これを分散 JoinJAVA に拡張し，その健全性，すなわち，型整合なプログラムは実行時に通信エラーが起こらないことを示す．
キーワード　Join 算法 型判定システム プロセス間通信

Type Judgement System for Communication Error in Distributed JoinJAVA Programs

Masaki SAEKI†, Toshiki SAKABE††, Masahiko SAKAI††,

Keiichirou KUSAKARI††, and Naoki NISHIDA††

†, †† Graduate School of Information Science, Nagoya University
Furou-chou, Chikusa-ku, Nagoya, 464-8603, Japan
E-mail: †saeki@sakabe.i.is.nagoya-u.ac.jp, ††{sakabe,sakai,kusakari,nishida}@is.nagoya-u.ac.jp

Abstract　The distributed JoinJAVA is the Join-Calculus extended by adding JAVA fragments as sequential processes. It enables us to describe communications between threads concisely by using the features of the Join-Calculus, while thread programming in JAVA is generally complex. The Join-Calculus is equipped with the type judgment system that guarantees that well-typed programs running without any communication errors. In this paper, we extend the type judgment system for the distributed JoinJAVA, and also show that the system is sound, that is, if a program in the distributed JoinJAVA is judged to be well-typed then it never runs into any communication errors.
Key words　Join-Calculus, type judgement, process communication

1. はじめに

一つのプログラム中に同時に複数の処理の流れが存在するようなプログラムをマルチスレッドプログラムと言う．JAVA は，このマルチスレッドプログラミングが可能な言語であり Web ブラウザや携帯電話などにおけるアプリケーションの記述やソフトウェア，システムの開発といった様々な分野で利用されている．しかし，JAVA においてマルチスレッドプログラミングを行うために必要な同期や通信に用いる手続きは，単純なものしか用意されていない．従って，スレッド間で同期，通信を行いながら動作するプログラムは，複雑になりやすいといえる．

一方，Join 算法 [1] は，プロセス通信をモデル化した計算モデルであり，特にプロセス間の同期や通信の記述に優れている．この Join 算法と JAVA 言語を融合することで，JAVA のマルチスレッドプログラミングにおける弱点を補える言語，JoinJAVA [4] が提案されている．しかし，この JoinJAVA は，Join 算法のすべての機能を融合させたわけではなく，JAVA 言語に関しても，一台の計算機上で動作する JAVA プログラムのみを扱う言語であった．また，プログラムの動作中におけるスレッド間の通信，同期が正しく行われているかを判断することも出来なかった．

本稿では，複数の計算機間でプロセスのやり取りをする仕組み [2] を JoinJAVA に導入し，Join 算法のすべての機能を JoinJAVA で扱えるよう拡張を行い，扱えるプログラムの範囲を広げる．以下ではこれを分散 JoinJAVA とする．また，本稿では，[3] に基いて分散 JoinJAVA における型および型判定システムを与え，その健全性を示す．これにより，型判定システムを用いることでプログラムの同期，通信という動作が正しく

行われていることを判定することが可能になる．

2. Join 算法

まず，本稿で扱う分散 JoinJAVA 言語の基礎となる計算モデルの Join 算法の概念について述べる．Join 算法は，プロセス通信をモデル化した計算モデルであるが，その特徴として複数のプロセスを並行して処理するプログラムを記述するのに適した構文を持っており，並行計算に優れた計算モデルといえる．

初期の Join 算法は，プロセス通信を表現するための最低限の要素で構成される計算モデルとして提案された [1]．しかし，現在では Join 算法で表現可能な範囲が拡大され，分散プログラムをモデル化するための概念が導入された [2] [3]．分散プログラムにおいて，プログラムはいくつかの異なる場所に分割されて実行される．この場所を表現するためにロケーションという概念が与えられた．また他の場所に存在するプログラムからは実行できないローカルなプロセスといった概念も必要になってくる．拡張された Join 算法の構文を以下に示す．

定義 1 Join 算法の構文を BNF 記法を用いて与える．

$$
\begin{array}{rcl}
L & \stackrel{\text{def}}{=} & 0 \\
& | & \texttt{loc}\ \alpha[D{:}P]^{\Delta,I,F} \\
& | & L \| L' \\
P & \stackrel{\text{def}}{=} & 0 \\
& | & P|P' \\
& | & n\langle \tilde{\mathrm{n}} \rangle ; P \\
& | & \texttt{go}\ a; P \\
& | & \texttt{def}\ D\ \texttt{in}\ P \\
& | & \texttt{newch}\ n.P
\end{array}
\qquad
\begin{array}{rcl}
D & \stackrel{\text{def}}{=} & 0 \\
& | & D\ \texttt{and}\ D' \\
& | & J{=}P \\
& | & \texttt{loc}\ a[D{:}P]^{\Delta,I} \\
J & \stackrel{\text{def}}{=} & n\langle \tilde{x} \rangle \\
& | & J|J' \\
\Delta, I & \stackrel{\text{def}}{=} & \emptyset \\
& | & \{n\} \\
& | & \{x\} \\
& | & \Delta \cup \Delta'
\end{array}
$$

□

Join 算法の主な構成要素はロケーション L，プロセス P，定義 D，Join パターン J である．Join パターンは，プロセス通信を行うためのポートであるチャネルを用いて，プロセス間でメッセージの通信を行う為のマッチング規則のことである．特定のパターンが出現したときに別のプロセスを起動するための規則を反応規則と呼び，定義 D で与えられる．プロセス P のアクションとしては，ロケーションを移動するための命令 **go** やローカルなチャネルを生成するための命令 **newch** 等が使える．ロケーション L は分割されたプログラムの一つを表現し，反応規則 D とプロセス P で構成されており，各ロケーションは固有の名前によって識別することが出来る．また，ロケーションは木構造を持っていて，各ロケーションは自らの位置より上の階層のロケーションで定義されている反応規則を利用することが出来る．各ロケーションにおいて，Δ, I はそれぞれ自身とその上のロケーションに反応規則の存在するチャネルの名前を保持している集合で．F は Δ, I を用いてチャネルとロケーションの対応付けをするための関数で以下のように定義される．

定義 2 F はチャネル n に対して，チャネルに対する反応規則の存在するロケーションを返す関数である．F がとる引数は，n が求めたいチャネル名，a は n の存在しているロケーション名，Δ, I はロケーション a に存在する Δ, I である．F' は a が最上位の階層にあるロケーションでなければ一つ上の階層で定義される関数 F であり，最上位の階層のロケーションであればすべて $\perp$ を返す関数である．

$$
F(n, a, F', \Delta, I) = \begin{cases} a & n \in \Delta \\ F'(n) & n \notin \Delta \wedge n \in I \\ \perp & otherwise \end{cases}
$$

□

Join 算法におけるプログラムの動作は大きく二つに分けられ，一つはロケーション間でのプロセスのやりとりである．もう一つはロケーション内での動作であり，これはロケーション中の D で記述される．以下にその動作を簡潔な例を用いて説明するが詳細な意味論については [3] を参照されたい．

例 1 ロケーション内のプロセス通信を行うプログラムについて考える．

$$
\texttt{loc a} \left[\begin{array}{c} P\langle x \rangle | Q\langle y \rangle = R\langle x \rangle \\ \texttt{and}\ R\langle z \rangle = S\langle z \rangle \end{array} : P\langle m \rangle | Q\langle n \rangle \right]^{\{P,Q,R\},\emptyset,F_a}
$$

□

この例ではロケーション a において，プロセスとして $P\langle m \rangle$ と $Q\langle n \rangle$ が存在し，また反応規則として $P\langle x \rangle | Q\langle y \rangle$ という定義がある．よって，この反応規則にプロセスがマッチするので $P\langle m \rangle$ と $Q\langle n \rangle$ が消費され $R\langle m \rangle$ というプロセスが起動される．さらにもう一つの反応規則に，新たに起動された $R\langle m \rangle$ にマッチするので，$R\langle m \rangle$ が消費され，$S\langle m \rangle$ が起動される．

例 2 ロケーション移動を行うプログラムについて考える．

$$
\begin{array}{l}
\texttt{loc a} \left[\begin{array}{c} P\langle x \rangle = P'\langle x \rangle\ \texttt{and} \\ \texttt{loc a'}\,[Q\langle x \rangle = Q'\langle x \rangle : \texttt{go}\ b; P\langle n \rangle]^{\{Q\},\{P\}} : 0 \end{array} \right]^{\{P,Q\},\emptyset,F_a} \\
\Big\| \texttt{loc b} \left[P\langle x \rangle = R\langle x \rangle : 0 \right]^{\{P\},\emptyset,F_b}
\end{array}
$$

□

この例では 3 つのロケーション a,a' そして b が存在する．ロケーション a' はロケーションが取る木構造において，a の子にあたり，a' 中には go b という命令が存在する，この命令は命令が存在するロケーション全体として，行き先に移動するという動作を行う．従って，ロケーション a' は b の子の位置に移動する，丁度ロケーション a のエージェントの様な役割を果たすことになる．そして移動した先のロケーション b の反応規則に従って，$R\langle n \rangle$ というプロセスが起動される．ここで注意するのはロケーション a においてもプロセス P に対する反応規則が存在するが，go 命令のほうが先に実行されるためこの規則にはプロセス $P\langle n \rangle$ は反応しない．

3. JoinJAVA

Join 算法と JAVA 言語を融合し，JAVA のマルチスレッドプログラミングにおける弱点を補う言語として，JoinJAVA [4] は提案された．この JoinJAVA では JAVA プログラムにおけるスレッドを一つのプロセスとして表現することで，JAVA プ

ログラム中のスレッドの同期，通信を Join 算法で記述することを可能にした．これによってスレッドの同期，通信を簡潔な記述で実現できるようになる．

この JoinJAVA のプログラムの構造は大きく分けると JAVA プログラムと Join 算法プログラムの二つに分類される．そして，それぞれのプログラム中において JAVA から Join 算法のプロセスを起動するための命令，および Join 算法中で JAVA のステートメントを記述できる構文を追加することで，JAVA のスレッドの同期を Join 算法で実現している．

3.1 分散 JoinJAVA

従来の JoinJAVA は分散環境下でのプロセスの通信などを含まない Join 算法を元に作られていた．しかし Join 算法で扱えるプログラムの範囲が現在では広がっており，定義 1 のロケーションやその移動を扱うことができる．本節ではそれに合わせる形で拡張された Join 算法と JAVA 言語を融合し，分散 JoinJAVA を提案する．これにより，複数の計算機に分散している JAVA のプログラム間での通信を簡潔に記述することが出来るようになる．

定義 3 (構文) 分散 JoinJAVA の構文を，定義 1 で示した Join 算法の構文に以下のプロセスを追加することで与える．

$$P \stackrel{\text{def}}{=} \texttt{JAVAstatement}; P$$ □

`JAVAstatement` には JAVA のステートメントを直接記述することが出来る．また，次の命令を JAVA のステートメントに追加する．これは JAVA から Join 算法で定義されたプロセスを実行するための命令である．

$\texttt{spawn}\ P\langle n\rangle[|P'\langle n'\rangle]$

`spawn` 命令では | でつなげられた各プロセスが同時に起動され Join 算法の意味論に従って動作する．また，この JAVA ステートメントの構文の動作を表す意味論を他の Join 算法の構文に対する意味論と同様に定義する．

定義 4 (意味論) JAVA ステートメントは，1 つ以上の Join 算法のプロセスを起動するか，空プロセスを返すことになる．また，$Spawns$ は JAVA ステートメントで起動されるプロセスを返す．

$$[\text{JAVA}]\quad \frac{P = Spawns(\text{JAVAstatement})}{\mathbf{loc}\ \alpha[\mathcal{D} : \mathcal{P} \mid \text{JAVAstatement}; Q]^{\Delta,I,F} \longrightarrow \mathbf{loc}\ \alpha[\mathcal{D} : \mathcal{P} \mid P \mid Q]^{\Delta,I,F}}$$

$$Spawns(\mathcal{E}) = 0$$

$$Spawns(st; sts) = \begin{cases} P \mid Spawns(sts) & \text{if } st = spawn\ P \\ Spawns(sts) & otherwise \end{cases}$$ □

例 3 JAVA のステートメントを含むプログラムについて考える．

$$\texttt{loc a}\left[P\langle x\rangle = Q\langle x\rangle : \begin{array}{c}\texttt{read(String s);}\\ \texttt{spawn}\ P\langle \texttt{s}\rangle;\end{array}\right]^{\{P\},\emptyset,F_a}$$

□

この例において，`read,spawn` で始まる 2 つの文は JAVA ステートメントである．`read` は JAVA のメソッドで標準入力から文字列を読み込むものとする．この `read` メソッドによって読み込まれた文字列は，`spawn` 命令によって Join 算法のプロセスとして起動され，ロケーション内にある反応規則 $P\langle x\rangle = Q\langle x\rangle$ によってプロセス Q に読み込んだ文字列が渡される．

定義 5 (Join 算法におけるデータ型) JoinJAVA では反応規則でマッチした変数を JAVA ステートメント内で使用することが出来る．その際 JAVA ステートメントで使用されるチャネルのデータ型は反応規則で宣言されていなければならない． □

例 4 反応規則に JAVA のデータ型を持つプログラムについて考える．

$$\texttt{loc a}\left[\begin{array}{c} P\langle \text{int } x\rangle | Q\langle \text{int } y\rangle = \\ \texttt{System.out.println}(x+y); 0\end{array} : P\langle 1\rangle | Q\langle 2\rangle\right]^{\{P,Q\},\emptyset,F_a}$$

□

この例では x 及び y の型が反応規則で宣言されている．このプログラムが実行されることで Join 算法の 2 つのプロセスが持っていた整数型の情報が JAVA のステートメントに渡され，標準出力に 3 が出力される．

JAVA においては同期，通信を行いたいスレッドの数が増加するに従って，その構造が複雑になるが，これを Join 算法のプロセス通信のという形で表現することで JAVA のスレッドの同期，通信部分を簡潔な記述を行うことが出来る．

4. 型判定システム

JAVA のプログラムにおいて，スレッドの同期を行いスレッド間で通信を行うことはできるが，そのようなスレッド間の通信がプログラマの意図どおりに動作するかどうかというのは，コンパイル時などに予め判定することは困難である．これは，スレッド間における通信のエラーが，プログラムの動作に従ってスレッドが保持するデータが変化するといった動的な要因により発生するためである．分散 JoinJAVA では，このようなスレッド間の同期，通信を Join 算法の意味論に従って行う．これにより，JAVA のプログラムでは判定の難しいプログラム実行時のスレッド間通信エラーを Join 算法の手法を用いて静的に発見することが出来る．

Join 算法においては，プロセス通信の際に生ずるエラーを判定するための仕組みとして，型判定システム [3] が提案されている．これはまず，Join 算法のプログラムにおけるすべての名前に型をつける．そして，Join 算法の構文毎に型判定規則を与えることで，プログラム中の名前が Join 算法の意味論に従って動作したときに，プロセス通信エラーを起こす可能性があるかどうかを判定するシステムである．

4.1 型

この型判定システムの対象を分散 JoinJAVA にまで広げる．まず，分散 JoinJAVA プログラム中の名前に対する型の定義を与える．この型は，プログラム中の名前がチャネルであるのか，ロケーションであるのかという情報を型として表現したもので

あり，定義は以下の通りである．

定義 6 (分散 JoinJAVA における型) τ を基本型，δ を型スキームとする．分散 JoinJAVA 中のすべての名前は τ または δ の型を持つ．

$$
\begin{array}{rcl}
\tau & ::= & \tilde{\tau} \\
& | & \langle\tau\rangle_{\omega}^{+} \\
& | & \langle\tau\rangle_{\Delta} \\
& | & loc(\Delta) \\
& | & jsta \\
& | & \varphi \\
& | & \rho \\
\sigma & ::= & \forall\tilde{\rho}\,\tilde{\delta}.\tau
\end{array}
\qquad
\begin{array}{rcl}
\omega & ::= & \mathrm{n} \\
& | & \delta \\
\Delta, I & ::= & \emptyset \\
& | & \{\omega\} \\
& | & \Delta \cup \Delta \\
\varphi & ::= & \mathrm{int} \\
& | & \mathrm{String} \\
& \vdots &
\end{array}
$$

□

基本型 τ は Δ などの付随的な型の情報を持つことで，その名前のより詳細な情報を得ることが出来る．$\tilde{\tau}$ は基本型のベクトル表現であり $\tau_1, \tau_2, \cdots, \tau_n$ を省略したものである．$\langle\tau\rangle_{\omega}^{+}$ と $\langle\tau\rangle_{\Delta}$ はチャネルに対してつけられる型であり，前者は反応規則内で出現するプログラムの実行に伴って名前の変化する名前に付き，後者はそれ以外のチャネルを表す名前に付く．また，$\langle\tau\rangle$ という形で再帰的に出現する基本型 τ は，チャネルの引数の型を表している．$loc(\Delta)$ は，ロケーションを表す名前の型で，Δ はロケーション内で定義されているチャネル名の集合を表現している．$jsta$ は，JAVA ステートメントに対する型であり，φ は，JAVA の任意の型を表す型である．また，ρ は型変数であり，反応規則に出現する静的には何の型であるか判定できない場合の型である．これらの型のうち $jsta$ と，φ を除いたものについては Join 算法で既に与えられている型と同様のものである．

例 5 (分散 JoinJAVA の型) 次のプログラムを考える．

$$
\texttt{loc a}\left[\begin{array}{c} P\langle \mathrm{int}\ x\rangle | Q\langle \mathrm{int}\ y\rangle = \\ \texttt{System.out.println}(x+y); 0 \end{array} : P\langle 1\rangle | Q\langle 2\rangle\right]^{\{P,Q\},\emptyset,F_a}
$$

この中に含まれる名前の型は以下の様になる．

$$
\texttt{a} : loc(\{P,Q\}),\ P : \langle int\rangle_{\{P\}},\ Q : \langle int\rangle_{\{Q\}},\ x : int,\ y : int
$$

□

このプログラムにおいて a はロケーションであり，このロケーションにはプロセス P, Q に対する反応規則の定義が与えられている．従って，a の型は $\texttt{a} : loc(\{P,Q\})$ となる．また，プロセス P は int 型の引数を 1 つ取るチャネルであり，プロセス中に含まれるチャネルは P のみであるので $P : \langle int\rangle_{\{P\}}$ という型となる．Q も同様である．

4.2 型 判 定

先の例にあるような名前と型の対の集合を型環境 Γ という．そして，この Γ に従って，プログラムの一部の項 $\mathcal{S}$ が Γ で示されている型と合っているかを判定することを型判定と呼ぶ．

この型判定を行う為の型判定規則を与える．この規則は定義 3 で示した分散 JoinJAVA の構文毎に定義されるが，分散 JoinJAVA 固有の構文

$$\texttt{JAVAstatement}; P$$

を除いては [3] で与えられた Join 算法の型判定規則をそのまま用いる．後の例で用いる規則を中心にその一部を示す．また，$\texttt{JAVAstatement}; P$ という構文に対する型判定規則を与える．

$$
[\mathrm{Go}]\ \frac{\Gamma \vdash a : loc(\mathbf{I}) \quad \Delta; \mathbf{I}; \Gamma \vdash P}{\Delta; \mathbf{I}; \Gamma \vdash \mathbf{go}\, a; P}
$$

$$
[\mathrm{Msg}]\ \frac{\Gamma \vdash n : \langle\tilde{\tau}\rangle_{\Delta\cup\mathbf{I}} \quad \Gamma \vdash \tilde{m} : \tilde{\tau} \quad \Delta; \mathbf{I}; \Gamma \vdash \mathrm{P}}{\Delta; \mathrm{I}; \Gamma \vdash n\langle\tilde{m}\rangle; \mathrm{P}}
$$

$$
[\mathrm{Par}]\ \frac{\Delta; \mathbf{I}; \Gamma \vdash \mathcal{P}_1 \quad \Delta; \mathbf{I}; \Gamma \vdash \mathcal{P}_2}{\Delta; \mathbf{I}; \Gamma \vdash \mathcal{P}_1 \mid \mathcal{P}_2}
$$

$$
[\mathrm{Join}]\ \frac{\begin{array}{c} \Delta; \mathbf{I}; \Gamma + \tilde{u_i} : \tilde{\tau_i} \vdash P \quad \Gamma \vdash m_i : \langle\tilde{\tau_i}\rangle_{\omega_i}^{+} \\ \{\tilde{u_i}\} \cap dom(\Gamma) = \emptyset \\ \forall i, j. m_i \neq m_j \Rightarrow fv(\langle\tilde{\tau_i}\rangle_{\omega_i}^{+}) \cap fv(\langle\tilde{\tau_j}\rangle_{\omega_j}^{+}) \subseteq \Theta \end{array}}{\Delta; \mathbf{I}; \Gamma \vdash m_1\langle\tilde{u_1}\rangle \mid \cdots \mid m_i\langle\tilde{u_i}\rangle = P :: \emptyset :: \{m_i\} :: \Theta}
$$

$$
[\mathrm{Soup}]\ \frac{\begin{array}{c} \forall n. n \in \Delta \cup \mathbf{I} \Rightarrow n : \langle\tau\rangle_n^{+} \in \Gamma \quad a : loc(\Delta \cup \mathbf{I}) \in \Gamma \\ \Delta; \mathbf{I}; \Gamma \vdash \mathcal{P} \quad \Delta; \mathbf{I}; \Gamma \vdash \mathcal{D} : \Delta \end{array}}{\Gamma \vdash \mathbf{loc}\ \alpha : a[\mathcal{D} : \mathcal{P}]^{\Delta,\mathbf{I},F}}
$$

それぞれの規則の意味は次のようなものである．規則 [Go] は，a, P が型付けされていて，チャネル名の集合 $\mathbf{I}$ が a で利用できるならば，ロケーション移動命令の型付けが出来るということを示す．規則 [Msg] は，チャネル n，その引数 $\tilde{m}$，プロセス P が型付けされていて，n が要求するチャネル定義 $\Delta, \mathbf{I}$ と P のチャネル定義が一致するならば，n を用いた $\tilde{m}$ を出力するメッセージの型付けができるということを示す．規則 [Par] は，$\mathcal{P}_1, \mathcal{P}_2$ が同一の環境の下で型付けできるならば，それらの並行構造も型付けできることを示す．規則 [Join] は，メッセージの入力に用いるチャネル m_i, 反応規則によって起動されるプロセス P の型付けができて，反応規則に共通に含まれる変数が Θ の部分集合であるならば，反応規則の型付けが出来ることを示す．規則 [Soup] は，ロケーション名とそこで必要なチャネル名の定義がすべて Γ に含まれていて，$\mathcal{D}, \mathcal{P}$ が Γ から型付けできるならば，ロケーションの型付けが出来ることを示す．

$$
[\mathrm{JAVA}]\ \frac{\begin{array}{c} \Gamma \vdash \mathrm{JAVAstatement} : jsta \quad \Delta; \mathbf{I}; \Gamma \vdash P \\ Q = Spawns(\mathrm{JAVAstatement}) \quad \Delta; \mathbf{I}; \Gamma \vdash Q \end{array}}{\Delta; \mathbf{I}; \Gamma \vdash \mathrm{JAVAstatement}; P}
$$

この規則では $\texttt{JAVAstatement}; P$ という JoinJAVA の構文の型判定を行うためにプロセス P と，`JAVAstatement` を実行した際に得られる Join 算法のプロセス Q が Γ の下でプロセス P と Q に対する型判定規則を用いて型判定が成功し，`JAVAstatement` が Γ の中で確かに JAVA のステートメントであると定義されていると判定されたときに構文全体として，型判定に成功する．ということを定義してある．このように，ある Γ の下で $\Gamma \vdash \mathcal{S}$ と書かれた場合，項 $\mathcal{S}$ が規則に従って正しく型判定されたことを表すが，これは規則を再帰的に用いることにより判定される．

JAVA ステートメントの型判定を行うにあたっては，実際にステートメント内で JAVA のプログラムがどのような動作を行うかについては考慮せず，ステートメントの出力のみに注目するように型判定規則を与えた．これは Join 算法がプロセスの

$$\dfrac{\Gamma \vdash \mathtt{b} : loc(P,Q),\ \dfrac{\Gamma \vdash \text{JAVAprogram} : jsta,\ 0 = Spawns(\text{JAVAprogram})}{\Delta, I, \Gamma \vdash 0}[\text{Nil}],\ \dfrac{\Gamma \vdash P : \langle int, \langle int\rangle_{\{Q\}}\rangle_{\{P,Q\}},\ \Gamma \vdash Q : \langle int\rangle_{\{Q\}},\ \Gamma \vdash 3 : int}{\Delta; I; \Gamma \vdash P\langle 3, Q\rangle; 0}[\text{Msg}]}{\Delta; I; \Gamma \vdash \text{JAVAprogram}; P\langle 3, Q\rangle; 0}[\text{JAVA}]}{\Delta, I, \Gamma \vdash \mathtt{go\ a}; \text{JAVAprogram}; P\langle 3, Q\rangle; 0}[\text{Go}]$$

通信のみの動作に注目しそれ以外の動作については考慮しないという抽象化が行われている計算モデルであり，この計算モデル上で与えられている型判定規則を元に JAVA ステートメントの型判定を行うためには JAVA ステートメントで出力される Join 算法のプロセスにのみ注目するのが適切だと考えたからである．従って，JAVA ステートメントを含む分散 JoinJAVA のプログラムの型判定を正しく行う場合には，スレッドの通信を行うような命令が JAVA ステートメントには含まれていないという制約が必要であるいえる．

例 6 go a; JAVAprogram; $P\langle 3, Q\rangle; 0$ という一連のプロセスについての型判定を上に示す．ここで JAVAprogram はある JAVA のステートメントであり，Join 算法のプロセスを起動する spawn という構文を含まないものとする．従って，$0 = Spawns(\text{JAVAprogram})$ となる．また，この例で用いる Γ は，以下の通りである．

$$\mathtt{a} : loc(\{P,Q\}),\ P : \langle int,\ \langle int\rangle_{\{Q\}}\rangle_{\{P,Q\}},\ Q : \langle int\rangle_{\{Q\}},$$
$$\text{JAVAprogram} : jsta,\ 3 : int,\ \cdots$$

□

4.3 分散 JoinJAVA における通信エラー

プロセス通信におけるエラーの原因となりうる状況を次のように定義する．

定義 7 プログラム $\mathcal{S}$ が以下の条件のいずれかを満たすとき，プログラム $\mathcal{S}$ は通信エラーを起こす可能性を持つ．

（1） チャネルでない名前がチャネルとして扱われている．
（2） 命令 **go** n において，n がロケーションでない．
（3） 同じ名前のチャネルの引数の数が異なる．
（4） すべてのチャネルには対応するチャネル定義が存在する．

□

ここで示した通信エラーの起こりうる条件を満たしてしまうような分散 JoinJAVA プログラムにおいてどのような通信エラーが実際に起こりうるのかを例で紹介する．

例 7 次のプログラムは実行時に通信エラーを生ずる．

$$\mathtt{loc\ a}\,[P\langle x\rangle = P\langle x, y\rangle : \mathtt{go}\ P; P\langle 1\rangle]^{\{P\},\emptyset,F_a}$$

□

この例中に存在するプロセス P は，引数を 1 つ取るチャネルであるが，反応規則では 2 つの引数のプロセス P が生成される規則となってしまっている．これでは y の値が定まらずプロセスの通信エラーとなってしまう．これは条件の (3) にあたる．同様に go 命令で書かれる行き先はロケーションでなければいけないはずだが P というプロセス名が書かれているため行き先にいけず通信エラーとなってしまう．これは条件の (2) に当てはまる．このように条件を満たす場合には通信エラーとなる．

以降では分散 JoinJAVA のプログラムが型判定規則を用いることで型付けに成功したときには，上記のような条件を満たさない事，つまり通信エラーが起こらないという．型判定規則の健全性についての証明を行う．

5. 拡張された型判定システムの健全性

まず，型判定規則の健全性を求めるにあたって必要となる補題についての証明を行う．分散 JoinJAVA における型判定規則は多くの部分を Join 算法の型判定規則を元に作られているため，その健全性の証明は Join 算法で行われていた型判定規則の健全性の証明 [3] に準じて行える．

補題 1 プログラム $\mathcal{S}$ と，well-formed な型環境 Γ に対して $\Gamma \vdash \mathcal{S}$ とする．このときプログラム $\mathcal{S}$ に含まれる各ロケーション a に含まれるいかなるチャネル $n \in \Delta_a \cup I_a$ についても，対応するロケーションが必ず 1 つ存在する．

［証明］ ロケーション系列の長さに関する帰納法による．証明は Join 算法における証明と同様である．詳細な証明は [3] を参照されたい． □

補題 2 プログラム $\mathcal{S}$ と，well-formed な型環境 Γ に対して $\Gamma \vdash \mathcal{S}$ とする．このとき $\mathcal{S} \rightleftharpoons \mathcal{S}'$ であるような $\mathcal{S}'$ に対して well-formed な Γ' が存在して $\Gamma' \vdash \mathcal{S}'$ となる．但し，$\rightleftharpoons$ は意味論において相互に遷移可能な関係を表す．

［証明］ 分散 JoinJAVA の意味論において $\rightleftharpoons$ で表される規則は [α-STR] と [TREE] の 2 つである．従って，まずこれらの規則のいずれか 1 つを用いて 1 ステップでつながる $\mathcal{S}$ と $\mathcal{S}'$ について Γ' の存在することを示す．これが証明できれば，k ステップでつながるものに関しては容易に求めることが出来る．これらはいずれも Join 算法における証明と同様に証明できる，詳細は [3] を参照されたい． □

補題 3 $\mathcal{S}$ をプログラムとし，Γ を well-formed な型環境とする．もし $\Gamma \vdash \mathcal{S}$ かつ $\mathcal{S} \to \mathcal{S}'$ ならば，well-formed な Γ' が存在して $\Gamma' \vdash \mathcal{S}'$ となる．

［証明］ 意味論に従って変更される前後の $\mathcal{S}$ と $\mathcal{S}'$ の 2 つの構

造を元に $\mathcal{S}$ で型判定に成功した Γ と等しい Γ を用いて $\mathcal{S}'$ でも型判定が成功することを，意味論のすべての規則で証明する．意味論の規則は JAVA ステートメントに対するものを除いては Join 算法ですでに証明が存在するため，JAVA ステートメントを処理する意味論の規則についてのみ証明を行う．

分散 JoinJAVA の意味論の規則を使ってプログラムの動作を行う前後のプログラムが以下のような形をしていたとする．

$$\begin{aligned}&\texttt{loc a}\,[\mathcal{D}:\mathcal{P}|\texttt{JAVAstatement};P]^{\Delta,I,F_a}\\ \longrightarrow\ &\texttt{loc a}\,[\mathcal{D}:\mathcal{P}|Q|P]^{\Delta,I,F_a}\end{aligned}$$

ただし，プロセス Q は関数 $Spawns$ を用いて得られるプロセスである．このとき，動作前のプログラムが型環境 Γ によって型判定に成功していたとすると，以下のような型判定の構造をしていることがプログラムの構造からわかる．

(1) $\forall n.n \in \Delta \cup I \Rightarrow n : \langle\tau\rangle_n^+ \in \Gamma$

(2) $\Gamma \vdash a : loc(\Delta \cup I)$

(3) $\Delta; I; \Gamma \vdash \mathcal{D} :: \Delta$ (4) $\Delta; I; \Gamma \vdash \mathcal{P}$

(5) $\Delta; I; \Gamma \vdash P$ (6) $\Delta; I; \Gamma \vdash Q$

(7) $\Gamma \vdash \texttt{JAVAstatement} : jsta$

(8) $Q = Spawns(\texttt{JAVAstatement})$

$$\dfrac{(1,2,3),\ \dfrac{(4),\ \dfrac{(5),(6),(7),(8)}{\Delta; I; \Gamma \vdash \texttt{JAVAstatement}; P}[\text{JAVA}]}{\Delta; I; \Gamma \vdash \mathcal{P} \mid \texttt{JAVAstatement}; P}[\text{Par}]}{\Gamma \vdash \texttt{loc a}\,[\mathcal{D}:\mathcal{P} \mid \texttt{JAVAstatement};P]^{\Delta,I,F_a}}[\text{Soup}]$$

従って，この型判定に用いた条件を用いて，動作後のプログラムの型判定を作ることが出来ればよい．

$$\dfrac{(1,2,3),\ \dfrac{(4),\ \dfrac{(5),(6)}{\Delta; I; \Gamma \vdash Q \mid P}[\text{Par}]}{\Delta; I; \Gamma \vdash \mathcal{P} \mid Q \mid P}[\text{Par}]}{\Gamma \vdash \texttt{loc a}\,[\mathcal{D}:\mathcal{P} \mid Q \mid P]^{\Delta,I,F_a}}[\text{Soup}]$$

以上より意味論の規則 [JAVA] での証明が出来た． □

定理 1 $\mathcal{S}$ をプログラムとし，Γ を well-formed な型定義とする．このとき，$\Gamma \vdash \mathcal{S}$ であるならば，$\mathcal{S}$ は定義 7 の条件を満たさない．

[証明] 定義 7 で与えられた条件それぞれについて $\Gamma \vdash \mathcal{S}$ であるときに条件のような状態に $\mathcal{S}$ がならないことを示す．

（1） チャネルでない名前がチャネルとして扱われている．
プログラム $\mathcal{S}$ 中に含まれる任意の名前 n において，n がチャネルとして扱われている場合，それらは必ず，型判定規則 [Msg],[R-Msg] のどちらかで型判定されている．これらの規則を用いて型判定に成功しているのであれば，その前提条件として Γ において n がチャネル型であることが判る．また，この Γ はプログラム $\mathcal{S}$ を型判定している型環境 Γ であり Γ は健全であるので題意は満たされる．

（2） 命令 **go** n において，n がロケーションでない．
プログラム $\mathcal{S}$ 中に含まれるすべての命令 **go** n において，これらは必ず規則 [Go] で型判定されている．この規則を用いて型判定に成功しているのであれば，その前提条件として Γ において n がロケーションであることが判る．また，この Γ はプログラム $\mathcal{S}$ を型判定している型環境 Γ であり Γ は健全であるので題意は満たされる．

（3） 同じ名前のチャネルの引数の数が異なる．
まず，意味論と型判定において，チャネルの引数の数を変化させるような規則が存在しないことを確認しておく．また，反応規則を型判定するにあたって，反応規則に出現するチャネルの引数の数と，そのチャネルの型環境中に定義されている引数の数が一致することが規則 [Join] から，そして，このチャネルがプロセスとして型判定される場合にも，チャネルの引数の数と型環境中に定義されている引数の数が一致することが規則 [Msg],[R-Msg] から判る．従って，同名のチャネルであれば型環境 Γ によってその引数の数が一意に定まることが保障される．

（4） すべてのチャネルには対応するチャネル定義が存在する．
補題 1 より明らかである． □

プログラムの型判定に成功した場合には，通信エラーの条件を満たさないことが，定理 1 からいえた．そして，プログラムを意味論に従って動作させた場合においても，補題 2 および，補題 3 より型判定がくずれないことが言える．従って，プログラムを動かした場合にも通信エラーが起こらないことがいえる．

6. ま と め

本稿では，Join 算法に基づいた言語である分散 JoinJAVA を与え，この言語における実行時エラーの一つである通信エラーを引き起こす原因となる条件を定義すると共に，分散 JoinJAVA における型と型判定規則を定義することで，型判定に成功することで通信エラーを起こさないことを保障する型判定システムを与えた．また，この型判定システムの健全性の証明を行った．

今後の課題は，この型判定システムによって判定できる実行時エラーの対象を拡大し，それにあわせて型判定規則を改良することがあげられる．また，分散 JoinJAVA の実装および，通信エラーを判定する型判定の自動化も今後の課題である．

謝辞 本研究は一部，科研費#15500007，#16300005，#17700009 ならびに名古屋大学 21 世紀 COE プログラム（社会情報基盤のための音声・映像の知的統合）の補助を受けている．

文 献

[1] C. Fournet, G. Gonthier : The reflexive CHAM and the join-calculus, Principles of Programing Language, 1995.
[2] C. Fournet G. Gonthier J.J.Levy L.Maranget D.Remy : A Calculus of Mobile Agents, CONCUR, 1996.
[3] A. Schmitt : Safe Dynamic Binding in the Join Calculus, IFIP TCS, pp.563–575, 2002.
[4] 尾関 嘉一郎: Join 算法による並行計算の簡潔な記述が可能な JAVA 言語, 名古屋大学 修士学位論文, 2000.