Haskell のための非同期局所化 π 計算に基づくネットワークプログラミングフレームワーク

今井敬吾[†] 結縁祥治[†] 阿草清滋[†]

我々はプログラミング言語 Haskell へ，型付き非同期局所化 π 計算（typed Asynchronous Localized π-calculus; ALπ）を，ネットワークプログラミングのためのフレームワークとして埋め込む手法を提案する．本フレームワークは埋め込み言語として実装されることで以下の利点を持つ：(1) Haskell で構築されているため処理系が頑健である，(2) Haskell のリテラル，型や関数といった言語要素をフレームワークに組み込むことができる．さらに，ALπ の mobility に対する制限はフレームワークの実装を簡単にする．本フレームワークでは，ALπ のプロセスは PiMonad と呼ばれる Haskell のモナドとして実装する．結果として，通信により引き起こされる副作用は型の PiMonad タグで区別される．ALπ のサブタイプ関係は Haskell の多引数型クラスで実現する．サブタイプ関係にある型の対は通信の方向を反映した 3 つの 2 引数型クラスに属する．本フレームワークを用いた例として TCP/IP ネットワーク上のインスタント・メッセンジャアプリケーションの構築例を示し，有用性と利点を述べる．

A Network Programming Framework in Haskell Based on Asynchronous Localized π-calculus

Keigo Imai,[†] Shoji Yuen[†] and Kiyoshi Agusa[†]

We propose an embedding of the typed Asynchronous Localized π-calculus (ALπ) into the programming language Haskell as a framework for network programming. The framework has following advantages due to the embedded language nature: (1) the framework is robust due to being built upon the Haskell framework, and (2) the framework can incorporate various Haskell language elements, such as literals, types, and functions, in the framework. Moreover, the limitation of mobility in ALπ simplifies the implementation of the framework. ALπ processes are implemented by means of a Haskell monad called PiMonad. As the result, side-effects caused by communications are distinguished by the tags of PiMonad in typing. The subtyping relations is realized by multi-parameter type classes in Haskell, where a pair of subtype-related types belongs to three binary type classes reflecting the directions of communication. We illustrate the usefulness and benefits of our framework with an example of an implementation of instance messenger application over TCP/IP network.

1. はじめに

ネットワークアプリケーションの普及により，動的な通信リンクの変化をともなうネットワークプログラムはより一般的になりつつある．このようなプログラムにおいては，動的なネットワーク構造を持つ柔軟なソフトウェアの構成が可能である反面，ソースプログラムの記述だけから振舞いを十分に把握することは難しい．

本論文ではこうした動的なネットワーク構造の変化を記述できる体系である π 計算[10] に基づいてプログラミング言語 Haskell（以下，単に ‘Haskell’ と書く）に対する埋め込み言語を定義することによって，ネットワークプログラムの振舞いを直接記述できるフレームワークを提案する．π 計算では，チャネルの送受や生成を行う並行プロセスを柔軟に記述でき，型理論やプロセスの等価性に関する多くの解析技法が開発されている．

我々は Haskell のための型付き非同期局所化 π 計算（typed Asynchronous Localized π-calculus；以下，ALπ と略記する）に基づくネットワークプログラミングフレームワーク PiMonad を実装する．ここで ALπ の型システムと構文を Haskell に埋め込み，PiMonad プログラムのネットワーク機能の動作を Haskell で実装する．

† 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University

ALπ は，π 計算を次の 2 つの点で限定した体系である．(1) 出力プレフィックスの後に **0** 以外のプロセスを認めない（非同期性），(2) 名前の入力チャネルに用いられる名前を他のプロセスに出力しない（局所性）．非同期性は，計算能力を限定することなく，実装を容易にする[12]．局所性によって通信の入力チャネルの移動が制限され，通信が発生するホストを構文的に特定することができる．このような特徴は，IP アドレスやノード番号に基づいて通信を行う多くのネットワークプログラムの概念によくあてはまり，比較的低レベルなネットワークプログラミングのモデルとして適当である．

PiMonad において，ALπ の構文は Haskell のモナドとして表現される．Haskell で入出力に用いられる `IO` モナドと同様に，通信等の副作用をともなう部分を，PiMonad として分離して記述するのが特徴である．純粋関数的部分は Haskell の処理系および標準ライブラリを利用することができる利点がある．さらに，埋め込み言語によって実装することで，Haskell 処理系が持つ厳密さに基づいて言語処理系を構築することができる．

本論文では，PiMonad の構成，Haskell の I/O システムへの適用，さらにネットワーク上での実装を示す．まず I/O システムとの相互作用をともなわない体系，π 計算の記述のための関数および評価器について定義する．型付き言語である ALπ を Haskell に埋め込む際の問題の 1 つは，Haskell の型システムにはサブタイプ機構がないことである．

本論文では，サブタイプ関係を多引数型クラス（multi-parameter type class）を用いて表現する．次に，I/O システムとの相互作用をともなう体系への拡張を示す．`IO` モナドを出力用チャネルとして表し，この実装により一般的な入出力の記述が可能であることを示す．さらに，ネットワーク入出力機能を表す `IO` モナドを提供し，チャネル表現に通信ホストの位置情報を組み込むことで，ネットワークを介した名前通信へ拡張する．このように拡張を行うことで実際的なネットワークプログラムが抽象的なモデルに近い形で直接的に記述できる．

ALπ の利点は，ネットワーク実装において示される．局所性を仮定しない場合，通信がネットワーク上のどのホストで発生するのか特定できず，実装が複雑になる．本フレームワークは文法を ALπ に限定し，記述の対象から複雑な挙動を除外する．このため，処理系としては軽量な実装が可能になっている．

本論文の構成は以下のとおりである．2 章で我々が用いる ALπ の構文と操作意味論を紹介する．3 章では非同期 π 計算に対する PiMonad を構成する方法を示す．4 章では π 計算の i/o 型とサブタイプ関係による局所性を型クラスで実現する方法について述べる．5 章では Haskell の入出力システムとの相互作用のために拡張する．6 章でネットワーク実装について述べ，7 章でプログラム例を与える．8 章で関連研究について述べ，9 章でまとめと今後の課題について述べる．

2. 型付き非同期局所化 π 計算

本章では，本論文で用いる ALπ の構文と型システムおよび意味を述べる．表現力の例としてセマフォの記述例をあげる．

2.1 構文と基本動作意味

本フレームワークが対象としている π 計算の構文を**図 1** に示す．値は，通信リンクまたは他の基本的な値からなる．構文に対するラベル付き遷移関係による基本動作意味を**図 2** に示す．$fn(\alpha)$ は α の中で束縛されていない名前の集合，$bn(\alpha)$ は束縛された名前の集合を示す．プロセス式 P に出現する自由な名前の集合を $fn(P)$ とし，P における自由な z の出現の y での置き換えを $P\{y/z\}$ で表す．以上の定義に基づいて，プロセスは以下のような直観的な意味を持つ．

- 入力：プロセス $v(x).P$ は，名前 v を持つ出力プロセスが存在する場合に値を入力し，P 中の x を入力した値に置き換える．
- 非同期出力：プロセス $\overline{v}w.\mathbf{0}$ は，名前 v を持つ入力プロセスが存在する場合に値 w を出力し，終了する．$\overline{v}w$ と略記する．
- 並行合成：$P|Q$ は，並行に動作する 2 つのプロセス P，Q を表す．
- 名前制限：$(\boldsymbol{\nu} x : L)P$ において，型 L を持つ名

Values

v, w	::=	x	channel
	|	$basval$	basic value

Processes

P, Q	::=	$\overline{v}w.\mathbf{0}$	asynchronous output
	|	$v(x).P$	input
	|	$P\|Q$	parallel execution
	|	$(\boldsymbol{\nu} x : L)P$	restriction
	|	$!v(x).P$	replicated input
	|	$\mathbf{0}$	terminated process

図 1 ALπ：プロセスと値の文法
Fig. 1 ALπ: grammar of processes and values.

$$\textsc{AOut}: \frac{}{\overline{x}y.\mathbf{0} \xrightarrow{\overline{x}y} \mathbf{0}} \qquad \textsc{Inp}: \frac{}{x(z).P \xrightarrow{xy} P\{y/z\}}$$

$$\textsc{RepInp}: \frac{}{!x(z).P \xrightarrow{xy} P\{y/z\} \mid !x(z).P}$$

$$\textsc{Par-L}: \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\, bn(\alpha) \cap fn(Q) = \emptyset \qquad \textsc{Par-R}: \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\, bn(\alpha) \cap fn(P) = \emptyset$$

$$\textsc{Comm-L}: \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \textsc{Comm-R}: \frac{P \xrightarrow{xy} P' \quad Q \xrightarrow{\overline{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Res}: \frac{P \xrightarrow{\alpha} P'}{(\boldsymbol{\nu} x : L)\, P \xrightarrow{\alpha} (\boldsymbol{\nu} x : L)\, P'} \text{ if } \alpha \notin \{x, \overline{x}\} \qquad \textsc{Open}: \frac{P \xrightarrow{\overline{x}z} P'}{(\boldsymbol{\nu} z : L) P \xrightarrow{\overline{x}(z)} P'}\, z \neq x$$

$$\textsc{Close-L}: \frac{P \xrightarrow{\overline{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} z : L)(P' \mid Q')}\, z \notin fn(Q) \qquad \textsc{Close-R}: \frac{P \xrightarrow{xz} P' \quad Q \xrightarrow{\overline{x}(z)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} z : L)(P' \mid Q')}\, z \notin fn(P)$$

図 2 ALπ の遷移規則
Fig. 2 ALπ: operational transition rules.

Actions

π	::=	$\overline{x}y$	output action
	|	xy	input action
	|	$\overline{x}(z)$	bound output
	|	τ	internal action

図 3 ALπ：アクション
Fig. 3 ALπ: actions.

前 x のスコープを P に制限する．文脈から明らかな場合，型の記述を省略し $(\boldsymbol{\nu}x)P$ と略記する．

- 複製：$!v(x).P$ 式は，式 $v(x).P$ の無限の並行合成 $v(x).P|v(x).P|\cdots$ と見なされる．
- 終了：$\mathbf{0}$ 式は，終了したプロセスを表す．

以下，$(\boldsymbol{\nu}x)(\overline{v}x|P)$ を，最も弱い結合力の演算子 $\overline{v}(x)P$ として略記する．

ラベルは**図 3** で表される．$\overline{x}y$ は値 y のチャネル x を介した出力を表す．xy は値 y の x を介した入力である．$\overline{x}(z)$ は x を介した制限された名前の出力を表す．ここで z は束縛された名前であり，スコープ射出（scope extrusion）規則によって，他の名前と区別される．τ は他のプロセスから観測できない内部動作を表す．

2.2 非同期性，局所性と表現能力

ALπ は，一般的な π 計算に，非同期性および局所性の制約を加えた体系である．

2.2.1 非 同 期 性

非同期 π 計算[1),6)] は出力プレフィクスの継続が $\mathbf{0}$ に限られる体系である（$\overline{v}w.\mathbf{0}$）．一方，同期 π 計算では，出力プレフィクスの継続に任意のプロセスが許される（$\overline{v}w.P$）．非同期 π 計算は TCP 等の既存のネットワークでの実装が簡易である利点がある．

2.2.2 局 所 性

π 計算における局所性（locality）とは，

- 入力プロセス $a(x).P$ について，束縛された名前 x が継続する P で入力プレフィクスに出現しない，
- チャネルの同一性を検査するマッチング $[\,x = y\,]\,P$ の構文を認めない，

という制約である．このような体系は局所化 π 計算[8),17)]（Localized π）と呼ばれる．

局所化 π 計算では，チャネルの出力能力のみをやりとりする．入力能力は $\boldsymbol{\nu}$ 演算子によってしか導入できない．たとえばプロセス P，Q 間でチャネル x の入力能力を共有する場合，$\boldsymbol{\nu}x(P|Q)$ の形をとらなければならない．このことは，同一の通信ホストに属するプロセス間でのみ入力が共有できることを表して

いる．

ALπ の表現力 Milner による λ 計算から π 計算へのエンコーディングが存在する[9] が，この変形として，ALπ へのエンコーディングも知られている[8]．

他の例として，ALπ における，二値セマフォの例をあげる．ここでは，π 計算で一般的な拡張として，図 1 の Values の定義に対 (x, y) と空値 () を加えた体系を用いる：

$$Semaphore(new) = !new(x).(\boldsymbol{\nu} p, v)\ (\overline{x}\langle p, v\rangle \mid (\boldsymbol{\nu} a)(\ !a().p(r).(\overline{r}\langle\rangle|v(s).(\overline{s}\langle\rangle|\overline{a}\langle\rangle)) \mid \overline{a}\langle\rangle)\ )$$

ここで new はセマフォを生成するためのチャネルである．new を介して得たチャネル x は生成したセマフォの P 操作と V 操作のためのチャネルをユーザプロセスに渡すために用いる．チャネル a はセマフォを初期化するために用いる．内側の $\overline{a}\langle\rangle$ は，V 操作の後，新たに次の P 操作を可能にする．この例は二値セマフォの例だが，さらに外側に 1 つある $\overline{a}\langle\rangle$ を n 個並行合成すれば n 値セマフォも実現できる．

ユーザプロセスにおいて，各操作はそれぞれ $\overline{p}(r)r()$, $\overline{v}(r)r()$ と書く．ここで r は同期のためのチャネルである．たとえば，P 操作の後，チャネル $SomeWork$ に出力し，最後に V 操作を行うユーザプロセスは以下のように書ける：

$$User(new) = \overline{new}(x)x(p, v).\,\overline{p}(r)r().\,\overline{SomeWork}(w)w().\,\overline{v}(s)s()$$

しかしながら局所性は，実装を容易にする一方，明らかに表現力を損なっている．たとえば，以下のようなプロセスは，ALπ には属さない：

$$a(x).x(y).\mathbf{0} \mid \boldsymbol{\nu} z\ (\ \overline{a}z \mid \overline{z}b\ )$$

このようなプロセスでは，制限されない並行合成演算子を越えて，入力チャネルが移動する．このため，入出力両方のチャネルが全体の構造上を自由に移動することになる．このことは，計算の自由度が増す反面，実際の計算資源との対応を複雑にする．

2.3 型付け規則

型のための構文を**図 4** で与える．型はプロセス型と値型，リンク型の 3 つがある．リンク型は，以下のような意味を持つ．

- 入力能力 $\mathsf{i}V$：入力プロセス $x(y).P$ にのみ用いることができる名前の型．
- 出力能力 $\mathsf{o}V$：出力プロセス $\overline{x}v$ にのみ用いることができる名前の型．
- チャネル $\sharp V$：入力および出力の両方に用いることができる名前の型．

値型は，基本型および出力能力のみからなり，ALπ の局所性を特徴付けている．型環境は Γ で表され，Γ のもとでプロセス P が型付け可能であるとき $\Gamma \vdash P : \diamond$ と書く．

Types

S, T	::=	V	value type
	|	L	link type
	|	$\diamond$	behaviour type

Value types

V, W	::=	B	basic type
	|	$\mathsf{o}V$	output capability

Link types

L	::=	$\mathsf{i}V$	input capability
	|	$\mathsf{o}V$	output capability
	|	$\sharp V$	both capability

Type environments

Γ	::=	$\Gamma, x : L$
	|	$\Gamma, x : V$
	|	$\emptyset$

図 4 ALπ：型の文法
Fig. 4 ALπ: grammar of types.

ALπ におけるプロセスの型付け規則およびサブタイプ規則は**図 5** で与えられる[13]．ここで，本論文では基本型の間にサブタイプ関係を定義しない．

π 計算のサブタイプ型システムでは，サブタイプ規則は Sub-Trans, Sub-II, Sub-OO, Sub-BB を含む．しかしながら，本論文では，通信リンクに用いられる値型が $\mathsf{oo}\cdots V$ の形のみであるため，Sub-trans については必要ないため，本論文で用いる ALπ のサブタイプ関係は Sub-refl, Sub-♯I, Sub-♯O のみですべて導出できる．

Proposition 2.1 $mV \leq mW (m \in \{\mathsf{i}, \mathsf{o}, \sharp\})$ が，規則 Sub-refl, Sub-♯I, Sub-♯O, Sub-II, Sub-OO と Sub-BB から導出されるとき，$V = W$．

Proof $\mathsf{o}V \leq \mathsf{o}W$ の場合のみ示す．V が基本型のとき，$W \leq V$ は Sub-refl から導出されるため，$V = W$．一方，$V = \mathsf{o}V'$ and $W = \mathsf{o}W'$ のとき，$\mathsf{o}V' \leq \mathsf{o}W'$ は Sub-OO から導出される．帰納法の仮定より，$V' = W'$，よって $V = W$．$\mathsf{i}V \leq \mathsf{i}W$ と $\sharp V \leq \sharp W$ の場合も同様． □

したがって，Sub-II, Sub-OO と Sub-BB で得られるサブタイプ関係が Sub-Refl のみで得られることがいえる．この性質に基づいて，型クラスによってサブタイプ関係を定義することができるようになる．

Value typing

$$\textsc{Tv-base}: \frac{}{\Gamma \vdash basval : B}\ basval \in B \qquad \textsc{Tv-name}: \frac{}{\Gamma, x : T \vdash x : T}$$

Process typing

$$\textsc{T-Par}: \frac{\Gamma \vdash P : \diamond \quad \Gamma \vdash Q : \diamond}{\Gamma \vdash P \mid Q : \diamond} \qquad \textsc{T-Nil}: \frac{}{\Gamma \vdash \mathbf{0} : \diamond}$$

$$\textsc{T-Res}: \frac{\Gamma, x : L \vdash P : \diamond}{\Gamma \vdash (\boldsymbol{\nu} x : L)P : \diamond} \qquad \textsc{T-AOut}: \frac{\Gamma \vdash a : \mathsf{o}V \quad \Gamma \vdash w : V}{\Gamma \vdash \overline{a}w.\mathbf{0} : \diamond}$$

$$\textsc{T-Inp}: \frac{\Gamma \vdash a : \mathsf{i}V \quad \Gamma, x : V \vdash P : \diamond}{\Gamma \vdash a(x).P : \diamond} \qquad \textsc{T-RepInp}: \frac{\Gamma \vdash a : \mathsf{i}V \quad \Gamma, x : V \vdash P : \diamond}{\Gamma \vdash !a(x).P : \diamond}$$

Subtyping

$$\textsc{Subsumption}: \frac{\Gamma \vdash v : S \quad S \leq T}{\Gamma \vdash v : T}$$

Subtyping rules

$$\textsc{Sub-refl}: \frac{}{T \leq T}$$

$$\textsc{Sub-}\sharp\textsc{I}: \frac{}{\sharp V \leq \mathsf{i}V} \qquad \textsc{Sub-}\sharp\textsc{O}: \frac{}{\sharp V \leq \mathsf{o}V}$$

$$\textsc{Sub-trans}: \frac{S \leq S' \quad S' \leq T}{S \leq T} \qquad \textsc{Sub-BB}: \frac{W \leq V \quad V \leq W}{\sharp V \leq \sharp W}$$

$$\textsc{Sub-II}: \frac{V \leq W}{\mathsf{i}V \leq \mathsf{i}W} \qquad \textsc{Sub-OO}: \frac{W \leq V}{\mathsf{o}V \leq \mathsf{o}W}$$

図 5 ALπ：型付け規則
Fig. 5 ALπ: typing rules.

3. 非同期 π 計算の Haskell でのコーディング

本章で，サブタイプ関係がない非同期 π 計算（Aπ）の Haskell での表現を示す．ここで問題となるのが，$\boldsymbol{\nu}$ 演算子に対応する一意な名前の生成方法である．

ここでは複数のモナドの機能の組合せにより，一意な名前を生成する機構を持つ PiMonad を与え，Aπ を Haskell で表現する．モナドの構造を把握するため，まず，$\boldsymbol{\nu}$ 演算子なしの非同期 π 計算（$\mathrm{A}\pi^{-\nu}$）のプロセスの Haskell での表現を与え，$\mathrm{A}\pi^{-\nu}$ の文脈（context）を表す継続渡しモナド `Ctx` を与える．次に，名前の生成器の状態をモナドに保持させるため，$\boldsymbol{\nu}$ 演算子を含んだ Aπ の文脈を，`Ctx` に状態計算を導入したモナドである PiMonad で与える．

3.1 $\mathrm{A}\pi^{-\nu}$ の Haskell での表現

チャネルの型定義 型 `a` を送受信する通信リンク ♯`a` を表す型を，`Ch a` として表す．たとえば，π 計算において型 ♯`String` で表されるチャネルを `Ch String`，型 ♯♯`()` を `Ch (Ch ())` 等と表す．一方，名前には型の情報は与えず，すべて整数で表す．チャネル型 `Ch a` の定義を以下のように与える．

```
data Ch a = Name Int
```

型 `Ch a` は，定義の左辺に現れる型変数が右辺には現れない幽霊型（phantom type）であり，型情報を忘却することで，名前はすべて `Int` 型の整数表現を

Process type

```
data P =
    Par P P                                          parallel composition P|Q
  | Zero                                             inactive process 0
  | forall t.(Typeable t) => Recv (Ch t) (t -> P)    input x(y).P
  | forall t.(Typeable t) => Send (Ch t) t           asynchronous output x̄y.0
  | forall t.(Typeable t) => ReprRecv (Ch t) (t -> P)  replicated input !x(y).P
```

図 6 $A\pi^{-\nu}$ の Haskell での表現
Fig. 6 Representation of $A\pi^{-\nu}$ in Haskell.

表 1 $A\pi^{-\nu}$ と Haskell の項の対応関係
Table 1 Correspondence between $A\pi^{-\nu}$ and Haskell.

$A\pi^{-\nu}$	Haskell
$x(y).P$	`Recv x (\y->p)`
$\overline{x}v$	`Send x v`
$!x(y).P$	`ReprRecv x (\y->p)`
$P \mid Q$	`Par p q`
$\mathbf{0}$	`Zero`

$x(y).(\overline{y}z \mid !z(w).\overline{w}v)$	`Recv x (\y->`
	`  Par (Send y z)`
	`  (ReprRecv z (\w->`
	`  Send w v)))`

持つ一方，送受信のためには違った型を持つ．

プロセスの型定義 次に，$A\pi^{-\nu}$ のプロセスの表現を与える．Haskell で，出力プロセス $\overline{x}v$ を `Send x v`，並行合成プロセス $P|Q$ を `Par p q`，終了したプロセス $\mathbf{0}$ を `Zero` で表す．入力プロセス $x(y).P$ による名前束縛は λ 束縛で表現し，`Recv x (\y->p)` で表す．複製 $!x(y).P$ も同様に `ReprRecv x (\y->p)` と表す．他に例として**表 1** のような対応関係がある．

各データ構築子を，`forall t. Send (Ch t) t` や `forall t. Recv (Ch t) (t->P)` 等と決める．これにより，型 `♯t` のチャネルで送受信可能な値の型が `t` であることを，Haskell の型システムが保証する．データ型の `forall` 構文は Haskell で一般的な拡張構文である．

しかしながら，`forall` で限量した型はそれぞれの値の出現で違った型として扱われるため，τ 遷移の計算に必要な “`Send c1 v` と `Recv c2 f` から `f v` を計算” といった記述は Haskell では型付けできずコンパイルエラーとなる．

ここで，`forall t. (Typeable t) => Send (Ch t) t` のように型 `t` が `Typeable` クラスに属すれば，関数 `cast` による型キャストを行うことができる．これにより，プロセスの τ 遷移を計算する関数を以下のように定義する．

```
tau :: P -> P -> Maybe P
tau (Par (Send (Name i) v)
         (Recv (Name j) f)) =
  if i==j then Just (Par Zero (f (cast v)))
          else Nothing
```

これらのデータ構築子をまとめ，チャネル型とあわせて，$A\pi^{-\nu}$ を表す型の定義を**図 6** で表す．チャネル自身も送受信可能とするため，データ型 `Ch a` も `Typeable` のインスタンスとして宣言する☆：

```
instance Typeable (Ch a) where
    typeof = ...
```

3.2 $A\pi^{-\nu}$ の文脈の継続モナドによる表現

モナドによるプロセスの表現は，後の名前生成を導入した体系において不可欠な役割を果たす．ここでは，まず型 `P` を結果の型とする継続モナド `Ctx` を次のように定義する：

```
data Ctx a = Ctx {runCtx :: ((a -> P) -> P)}
instance Monad Ctx where
  return x = Ctx (\k -> k x)
  (Ctx c) >>= f =
       Ctx (\k -> c (\a -> runCtx (f a) k))
```

型 `Ctx a` を持つモナドは型 `a` を持つある変数を束縛する文脈と見なすことができる．`>>=` 演算子は，ある文脈と同じ型の自由変数を持つ別の文脈から新たな文脈を合成する演算と見なせる．

文脈の穴にプロセス $\mathbf{0}$ を代入しプロセスを得る関数 `ctx2pr` を以下に定義する：

```
ctx2pr :: Ctx () -> P
ctx2pr (Ctx f) = f (\_ -> Zero)
```

さらに，モナド `Ctx` の組合せにより $A\pi^{-\nu}$ の基本的な演算子を表現するため，$A\pi^{-\nu}$ の文脈に対応する関数を与える．

入力 π 計算の入力ガード式 $x(y).P$ から，文脈 $x(y).[\cdot]$ を得る．対応する関数 `recv` を以下に定義する：

☆ 現行の Hugs 98 March 2005 の場合．GHC6.4 では型構築子 `Ch` を型 `Typeable1` のインスタンスとして宣言するか，データ型定義で `deriving Typeable` とする．

Generator for fresh names

```
type Gen  = [Int]
initial  = [1..]                                  infinite list of integer
```

Unrestricted process

```
data APiSub  =
   Par APi APi                                    parallel composition P|Q
 | Zero                                           inactive process 0
 | forall x.(Typeable x) =>
     Recv (Ch x) (x -> Gen -> (APi, Gen))         input x(y).P
 | forall x.(Typeable x) =>
     Send (Ch x) x                                asynchronous output x̄y.0
 | forall x.(Typeable x) =>
     ReprRecv (Ch x) (x -> Gen -> (APi, Gen))     replicated input !x(y).P
```

Process type

```
type APi  =  ([NewChan], APiSub)                  process with restriction
```

Restricted name

```
data NewChan  =  forall x.  NewChan (Ch x)        placeholder of the name
```

図 7 Aπ の Haskell での表現
Fig. 7 Representation of Aπ in Haskell.

```
recv :: (Typeable a) => Ch a -> Ctx a
recv ch = Ctx (\k -> Recv ch k)
```

引数は入力に用いられるチャネルを表す．定義における継続 k は型 a -> P を持ち，入力動作に継続するプロセスを表す．

非同期出力 非同期出力プロセス $\overline{x}v.\mathbf{0}$ には継続するプロセスはない．継続するプロセスを配置するために変数束縛のない文脈（$\overline{x}v.\mathbf{0}|[\cdot]$）を得る．対応する関数 send を以下に定義する：

```
send :: (Typeable a) => Ch a -> a -> Ctx ()
send ch v = Ctx (\k ->
                 (Par (Send ch v) (k ())))
```

第 1 引数は出力に用いられるチャネルを，第 2 引数は出力される値を表す．定義における継続 k は型 () -> P を持ち，並行合成演算の右手側に置かれる．

並行合成 並行合成プロセス（$P|P'$）より文脈（$P|[\cdot]$）を得る．対応する関数 fork を以下に定義する：

```
fork :: Ctx () -> Ctx ()
fork c =
  let p = ctx2pr c
  in Ctx (\k  -> Par p (k ()))
```

第 1 引数は別の文脈で，関数 ctx2pr によりプロセスに変換された後，並行合成演算の左手側に置かれる．

複製 複製プロセス $!x(y).P$ から，入力プロセスの場合と同様に文脈 $!x(y).[\cdot]$ が得られる．しかしながら，入力プロセスの場合と違い，継続するプロセスが無限に複製されるため直観にあわない．そこで非同期出力プロセスの場合と同様に，文脈（$!x(y).P|[\cdot]$）から対応する関数 rep を以下に定義する：

```
rep :: (Typeable a) =>
       Ch a -> (a -> Ctx ()) -> Ctx ()
rep ch cont =
  let k'  = \a -> ctx2pr (cont a)
  in Ctx (\k ->
           (Par (ReprRecv ch k') (k ())))
```

第 1 引数は入力に用いられるチャネルを，第 2 引数は入力動作に継続する文脈を表す．文脈は fork の場合と同様に ctx2pr によりプロセスに変換され並行合成される．

3.3 Aπ の Haskell での表現

名前制限演算子を含む Aπ のプロセスの Haskell での表現を**図 7** で与える．以下で，この直観的意味を説明する．

名前生成器 名前は型 Int の整数値で表現する．ν 演算子で新しい名前を生成するための源として，Int の十分に大きいリストを与える．これを名前生成器と

表 2 Aπ と Haskell の項の対応関係
Table 2 Correspondence between Aπ and Haskell.

Aπ	Haskell
$(\boldsymbol{\nu} z : L)x(y).P$	([NewChan z], Recv x (\y->\gen->(([],p),gen)))
$(\boldsymbol{\nu} v : L)\overline{x}v$	([NewChan v], Send x v)
$!x(y).(\boldsymbol{\nu} z : L)P$	([], ReprRecv x (\y->\gen->(([NewChan z],p),gen)))

呼ぶ．名前生成器の型を Gen と名付ける．新しい名前を表す整数はリストの head を用い，生成器の次状態は tail を用いる．生成器の初期状態を initial と名付ける．

プロセスの型 名前制限されたプロセスの Haskell での表現を型 APi で与える．直観的に，そのスコープでの名前制限の（空かもしれない）リストと，名前制限の下のプロセスの対である．

名前制限 1 つの名前制限（$\boldsymbol{\nu}$z: L）を，NewChan z として表す．

名前制限演算子の下のプロセス 名前制限演算子の下のプロセスを表す型を APiSub とする．データは型 P にほぼ対応するが，入力プレフィクスに関わるデータ構築子 Recv，ReprRecv は，入力動作後に継続が適用されプレフィクスの下のプロセスが活性化されるため，引数に生成器を追加する必要がある．さらに，適用した生成器の次の状態を得る必要から，継続の戻り値にも生成器を追加する．よって，Recv および ReprRecv を次のように定義する：

```
...
| Recv (Ch a) (a -> Gen -> (APi, Gen))
| ReprRecv (Ch a) (a -> Gen -> (APi, Gen))
...
```

継続するプロセスがない Send，Par，Zero については P のデータ構築子をそのまま用いる．

例 $(\boldsymbol{\nu} z : \sharp a)\overline{x}z$ を ([NewChan z], Send x z) と表す．継続するプロセスで名前制限があるプロセス $x(y).(\boldsymbol{\nu} z : \sharp a)\overline{y}z$ は，

```
Recv x (\y -> gen ->
  let z = (Name (head gen)) :: Ch a
  in (([NewChan z], Send y z), tail gen)
```

と表す．その他のプロセスの対応関係の例を**表 2** にあげる．

3.4 状態モナドによる名前生成

ここで，Aπ の文脈に対応する一意な名前生成が可能なモナドを，3.2 節で与えた $A\pi^{-\nu}$ の文脈に対応するモナドに，状態をともなう計算を表すモナドの機能を組み合わせ構成する．

モナドで生成器の状態を保持するため，3.2 節で導入した継続モナド Ctx a に，状態モナドを合成する．継続モナドと状態モナドの合成 CS を以下に示す：

```
data ContState a =
  CS ((a-> S -> (R, S)) -> S -> (R, S))
instance Monad ContState where
  return a = CS (\k -> \s -> k a s)
  (CS f) >>= g =
    CS (\k -> \s ->
         f (\a -> \s' ->
             let (CS g') = g a in g' k s')
           s)
```

ここで状態を表す型を S，継続の結果の型を R としている．直観的には，このモナドに型 (a -> S -> (R, S)) を持つ継続と，型 S を持つ初期状態を適用すると，型 (R, S) で表される継続の結果の型と最終状態の対が得られる．

このモナドの状態の型を Gen，結果の型を APi として，Aπ の文脈を表すモナドを**図 8** に示す．これを PiMonad と呼ぶことにする．

本節の目的としていた名前制限の文脈 $(\boldsymbol{\nu} z : L)[\cdot]$ に対応する関数 new および補助関数 addNew を以下に定義する：

```
new :: (Typeable a) => PiMonad (Ch a)
new = PiMonad (\k -> \gen ->
    let newname = Ch (head gen)
        gen'    = tail gen
        (process, gen'') = k newname gen'
    in (addNew newname process, gen'')
  )
  where addNew n (ns, p) = (NewChan n:ns, p)
```

新しい名前は生成器の head からとり，生成器の次状態を tail としている．継続に新しい名前と生成器を

```
newtype PiMonad a =
  PiMonad {runPi::(a -> Gen -> (APi,Gen))
                  -> Gen
                  -> (APi,Gen)}

instance Monad PiMonad where
  return x = PiMonad (\k -> \g -> k x g)
  m >>= f =
    PiMonad (\k -> g ->
             runPi m (\a-> \g' ->
                       runPi (f a) k g') g
```

図 8 PiMonad：Aπ の文脈の Haskell での表現
Fig. 8 PiMonad: representation of Aπ context in Haskell.

適用し，プロセスを得，名前制限を関数 addNew によりプロセスに記録する．

3.2 節で定義した，$A\pi^{-\nu}$ の基本的な演算子に対応する send，recv，rep，fork も，$A\pi$ を表す PiMonad の形に変更する必要がある．ここでは send，recv，fork のみ示す：

```
send :: (Typeable a) =>
        Ch a -> a -> PiMonad ()
send ch v = PiMonad (\k -> \gen ->
    let (p, gen') = k () gen
    in (([], Par (Send ch v) p), gen')
  )
recv :: (Typeable a) => Ch a -> PiMonad a
recv ch = PiMonad (\k -> \gen ->
                   (([], Recv ch k), gen))
fork :: PiMonad () -> PiMonad ()
fork (PiMonad f) = PiMonad (\k -> \gen ->
    let (p, gen')  = f (\_->Zero) gen
        (q, gen'') = k () gen'
    in (([], Par p q), gen'')
  )
```

send において，継続 k に生成器 gen を適用し並行合成するプロセス p を得る．一方，recv は生成器を消費しない．入力動作が行われるときに生成器が適用され，名前生成が行われる．rep も同様に定義される．

3.5 抽象評価器による PiMonad の実行

PiMonad () 型の値として定義したプロセスはそのままでは実行されることはない．$A\pi$ の項と同様な実行意味を与えるのは，Haskell で定義した関数 eval :: APi -> [Trans] である．eval の 1 回の適用により 1 ステップ縮約され，$A\pi$ の遷移規則に対応した動作をする．ここで，Trans は $A\pi$ における遷移ラベルと遷移後のプロセスの対 (l, P) に相当する．eval は図 2 の遷移規則に対応した実行として定義されているものとする．

遷移を表す型 Trans の定義 Trans を，以下のように定義する：

```
data Trans =
      TauAct APi
  | forall x. (Typeable x) =>
      SendAct [NewChan] (Ch x) x APi
  | forall x. (Typeable x) =>
      RecvAct (Ch x) (x -> APi)
```

データ構築子 TauAct は，規則 Comm や Close-{L,R} に起因する τ 遷移に相当する．引数は，遷移後のプロセスに相当する．

SendAct は，規則 AOut に起因する出力遷移に相当する．第 2 引数と第 3 引数は，それぞれ遷移ラベルのチャネルと送られる値に対応する．第 4 引数は，遷移後のプロセスに対応する．第 1 引数（[NewChan]）は，送られる名前がチャネル型の場合に束縛された名前のリストに対応する．第 1 引数が空でないときは，規則 Open に対応する実行と見なせる．

RecvAct は，規則 Inp または RepInp に起因する入力遷移に相当する．第 1 引数は遷移ラベルのチャネルに相当する．第 2 引数は，入力値に適用すると遷移後のプロセスを返す関数を表す．

PiMonad の実行 PiMonad の実行は，プロセスを関数 eval の適用結果から 1 つのプロセスを選び，再度，eval を適用することを繰り返して得られる系列となる．本論文における意味論は Early 意味論であるため，入力遷移 RecvAct の場合は，関数から継続するプロセスを得るために外部からの入力が必要である．

4. i/o 型とサブタイプ関係の導入

本章で，ALπ の 3 つのチャネル型構築子 ♯，i，o に対応する型を前章で導入したチャネル型から区別し，さらに値型の概念を型クラスで定める．次に，図 5 で紹介した ALπ のサブタイプ関係を Haskell の多引数型クラス[15] により表現する方法を示す．前章で導入した send，recv，rep，new をこの型クラスを用いた形で修正し，Haskell への ALπ の埋め込みを達成する．

4.1 i/o 型および値型の導入

ALπ の 3 つのチャネル型と Haskell の関係をそれぞれ，入力能力 ia を I a，出力能力 oa を O a，そして入出力チャネル ♯a を L a として与える．

ALπ では基本型および出力能力のみが値型である．値型に対応する型クラス ValueType を定義し，出力能力と，値として送受信可能としたい Haskell の基本型を ValueType のインスタンスとして宣言する．cast 関数のため（3.1 節），ValueType は Typeable のサブクラスとして定義する．

上述 3 つのチャネル型および値型を表す型クラスを**図 9** に与える．これを用い，サブタイプ関係を用いない ALπ を**図 10** に与える．ここで，$A\pi$ に対応する図 7 の型 APiSub のデータ構築子 Send (Ch x) x，Recv (Ch x) ···，ReprRecv (Ch x) ··· をそれぞれ i/o 型のために特化した型 ALPiSub の Send (O x) x，Recv (I x) ···，ReprRecv (I x) ··· とし，さらに型 x の動く範囲を ValueType のインスタンスに制限している．名前制限のために用いられる NewChan は

Channel for both input and output capability

```
data L a = L Integer
```

Input capable channel

```
data I a = I Integer
```

Output capable channel

```
data O a = O Integer
```

Type class for value types

```
class (Typeable a) => ValueType a where
  -- no functions

instance ValueType Int where
instance ValueType Float where
...

instance (ValueType a) => ValueType (O a) where
```

図 9 i/o チャネル型と値型の Haskell での表現
Fig. 9 Refined channel types and value types in PiMonad.

Unrestricted process

```
data ALPiSub  =
   Par ALPi ALPi                                  parallel composition P|Q
 | Zero                                           inactive process 0
 | forall x.(ValueType x) =>
     Recv (I x) (x -> Gen -> (ALPi, Gen))         input x(y).P
 | forall x.(ValueType x) =>
     Send (O x) x                                 asynchronous output x̄y.0
 | forall x.(ValueType x) =>
     ReprRecv (I x) (x -> Gen -> (ALPi, Gen))     replicated input !x(y).P
```

Process type

```
type ALPi  =  ([NewChan], ALPiSub)                process with restriction
```

Restricted name

```
data NewChan  =  forall x.  NewChan (L x)         placeholder of the name
```

図 10 ALπ の Haskell での表現
Fig. 10 Representation of ALπ in Haskell.

チャネル型 L に特化する．

4.2 サブタイプ関係の型クラスによる表現

サブタイプ関係を型付けに関連付ける SubSumption 規則を，coercion[11] の役割を果たすメソッドを持つ多引数型クラスで表現する．

出力チャネル型 出力チャネルには，型 $\sharp V$ または oV を用いる．$\sharp V$ は oV に coerce する．これを次の多引数型クラス OChan で表す：

```
class (ValueType a) => OChan o a where
  coerceO :: o a -> O a
instance (ValueType a) => OChan O a where
  coerceO = id
```

```
instance (ValueType a) => OChan L a where
  coerceO (L i) = O i
```

出力対象の型はクラス宣言の文脈で値型に制限する．メソッド coerceO は最外の型構築子を O に変換する．インスタンスは O および L について宣言され，それぞれの coerceO の実装は，恒等関数と，名前表現を保持し型のみを L から O に変換する関数で与えられる．

入力チャネル型 ALπ では，入力チャネルには，型 ♯V または iV を用いる．出力チャネルと同様に，次のとおり型クラス IChan を宣言すれば十分である：

```
class (ValueType a) => IChan i a where
  coerceI :: i a -> I a
instance (ValueType a) => IChan I a where
  coerceI = id
instance (ValueType a) => IChan L a where
  coerceI (L i) = I i
```

値型 値型についてはサブタイプ関係を直接多引数型クラスと coercion のための関数を与える．次の型クラス Subtype によりこれを表現する：

```
class (ValueType c2) =>
      Subtype c1 c2 where
  coerce :: c1 -> c2
instance (ValueType v) =>
         Subtype (L v) (O v) where
  coerce (L i) = O i
instance (ValueType v) =>
         Subtype (O v) (O v) where
  coerce = id
instance Subtype Int Int where
  coerce = id
instance Subtype Float Float where
  coerce = id
(and many declarations for other base types)
```

スーパタイプが値型であることを要請するため，クラス宣言の文脈で型の第 2 引数を ValueType に制限する．図 5 に現れるサブタイプ関係のうち，Sub-♯O に対応する coercion 関数のみ，型のみを L から O に変換する関数で与える．残りはすべて Sub-Refl に対応するため，恒等関数で与える．

ALπ の演算子への適用 導入した型クラスを用いて，3 章で導入した基本的な演算子 send，recv を修正し**図 11** に示す．rep も同様に与える．new，fork についてはサブタイプ関係と関連しないため省略する．これらの関数により，ALπ のプロセスが Haskell に埋め込まれる．

例 4.1 PiMonad によるセマフォの記述 図 12

```
send :: (OChan c a, Subtype b a)
        => c a -> b -> PiMonad ()
send ch v = PiMonad (\cont -> \gen ->
    let (cont',gen') = cont () gen
      ch' = coerceO ch
      v' = coerce v
      p = ([], Send ch' v')
    in (([], Par p cont'), gen')
  )

recv :: (IChan i a) => i a -> PiMonad a
recv ch = PiMonad (\f -> \gen ->
      let ch' = coerceI ch
      in (([], Recv ch' f), gen)
  )
```

図 11 サブタイプ関係を表現した型クラスの ALπ への適用
Fig. 11 Use of defined type classes in ALπ operators.

に，2.2 節のセマフォを PiMonad によって記述する．ここで send_sync x は同期のためのプロセス抽象で，$\overline{x}(v)v()$ に対応する．セマフォの最大値の指定に変数 cnt を用い，標準ライブラリ関数 Control．Monad．replicateM_ でプロセス send a () を複製している．

5. Haskell の I/O システムの統合

本章では，PiMonad を Haskell の I/O システムと統合する手法について示す．Haskell の IO アクションは，ALπ の出力動作として定式化する．

まず，ALπ の出力能力型に，Haskell の IO アクションを対応付ける拡張を行う．次に，I/O システムから値を返却するよう拡張することで，I/O システムとの相互作用を可能にする．

5.1 1 方向通信

Haskell の I/O システムとの相互作用の実装として，まずは値を与えて出力操作を実行するのみの 1 方向の通信を実装する．

Haskell の入出力は，IO モナドで記述したアクションを介して行われる（π 計算のアクションと区別するため Haskell アクションと呼ぶことにする）．Haskell アクションを出力チャネルとして記述するため，チャネル型 O を以下のように修正する：

```
data O a =
    O Integer
  | Action String (a->IO ())
```

以後，データ構築子 O のチャネルをローカルチャネル，Action のチャネルをアクションチャネルと呼ぶ．

```
send_sync :: (ValueType a) => O (O a) -> PiMonad a
send_sync ch = do
  (a::L a) <- new
  send ch a
  recv a

semaphore :: (L (O (O (O ()), O (O ()))))) -> Int -> PiMonad ()
semaphore sem cnt = rep sem $ \x -> do
  (p::L (O ())) <- new
  (v::L (O ())) <- new
  send x (p, v)
  (a::L ()) <- new
  rep a $ \_ -> do
    r <- recv p
    send r ()
    s <- recv v
    send s ()
    send a ()
  replicateM_ cnt (send a ())

user :: (L (O (O (O ()), O (O ()))))) -> PiMonad ()
user sem = do
  (x::L (O (O ()), O (O ()))) <- new
  send sem x
  (p, v) <- recv x
  send_sync p
  -- some works here --
  send_sync v

all :: PiMonad ()
all = do
  (sem::(L (O (O (O ()), O (O ())))))) <- new
  semaphore sem 1 -- binary semaphore
  user sem
```

図 12 PiMonad でのセマフォの記述
Fig. 12 Semaphore in PiMonad.

データ構築子 Action の第 1 引数は，系内で一意なチャネルの名前を文字列で与える．第 2 引数は，チャネルに対する出力動作に対応する Haskell アクションを与える．アクションチャネルに対し出力動作を行うとき，この Haskell アクションを関数 Control. Concurrent. forkIO により並行実行するよう，3.5 節で導入した評価器を拡張する．

例 5.1 標準出力への非同期書き込み Haskell アクションの putStrLn を用いて，標準出力へのチャネル cout を作る：

```
cout :: O String
cout = Action "cout" (\x -> putStrLn x)
```

cout を用いて，標準出力へ文字列 "Hello, world!" の書き込みを行うプロセスを以下のように定義できる：

```
hello :: PiMonad ()
hello = send cout "Hello, World!"
```

5.2 同期と双方向通信

Haskell の I/O システムからの値の返却 ここま

```
fopenW :: O (String, O (O (String,O ()), O (O ())))
fopenW = Action "fopenW" (\(filename,retCh) -> do
    handle <- openFile filename WriteMode
    let write :: O (String, O ())
        write  = Action ("write"++show handle) (\(str,ch) -> do
            hPutStrLn handle str
            putBuf ch ()
          )
        close :: O (O ())
        close  = Action ("close"++show handle) (\ch -> do
            hClose handle >> putBuf ch ()
            putBuf ch ()
          )
    putBuf retCh (write, close)
   )
  where
    putBuf :: O a -> a -> IO ()
    putBuf (Action _ f) v = f v
```

図 13 ファイル書き込みチャネル `fopenW`
Fig. 13 File writer channel `fopenW`.

での実装では，非同期出力に対応する Haskell アクションが発火しても，Haskell アクションから評価器への値の返却の手段が存在しないため，Haskell アクションの完了を待機するプロセスや，戻り値を持つような Haskell アクションの呼び出しを記述することができない．

Haskell の I/O システムが返却値を格納し，評価器が取り出すバッファ `RecvBuf` を，FIFO バッファである `Control. Concurrent. Chan` を用いて準備する：

```
type RecvBuf = Chan RecvData
data RecvData =
  forall x. (ValueType x) =>
    RecvData (O x) x
```

`RecvData` は受信に用いたローカルチャネルと受け取った値を引数に持つ．バッファに投入された `RecvData` 型のデータは，評価器により取り出され，出力プロセスとして PiMonad のプロセスと並行合成される．こうして値の受け渡しが実現する．たとえばアクションチャネル `x` を介して値 `v` が送られた場合，バッファにはデータ `RecvData x v` が投入され，評価器によりプロセス `Send x v` が並行合成される．

次に，Haskell アクションが値を `RecvBuf` に格納するための方法を述べる．

Haskell アクション内でのローカルチャネルの扱い PiMonad の評価器から Haskell の I/O システムへローカルチャネルを渡す際，渡すローカルチャネルを，`RecvBuf` へ値を投入するアクションチャネルへ変換する．Haskell アクション側は，これを実行することで，値の返却を行う．このため，前章で導入した `ValueType` に，以下のように変換メソッド `trans` を追加する：

```
class (Typeable x) => ValueType x where
  ...
  trans :: RecvBuf -> x -> IO x
  trans _ x = return x
```

`trans` は変換を行わないデフォルトメソッドを持つ．出力チャネルについてのみこれをオーバライドする：

```
instance (ValueType x) =>
           ValueType (O x) where
  ...
  trans buf ch@(O i) =
    return (Action (show i) (\x ->
      writeChan buf (RecvData ch x)))
  trans _ ch = return ch
```

`trans` は，ローカルチャネルを，`RecvBuf` へ値を投入（`writeChan`）するアクションチャネルへ変換する．アクションチャネルの文字列表現には元のローカルチャネルの整数表現が用いられる（`show i`）．

例 5.2 ファイル書き込みチャネル ファイルへの書き込みに用いるチャネル `fopenW` を**図 13** に定義する．

fopenW はファイル名を表す文字列 filename とチャネル retCh の対を受け取り，ファイルを開き，書き込み用チャネル write とファイルを閉じるためのチャネル close の 2 つを，渡されたチャネル retCh を介して返す．書き込み用チャネルは書き込む文字列 str とチャネル ch を受け取る．文字列がファイルに書き込まれた後，ch を介し値 () を送信する．これにより利用者は書き込みの完了を知ることができる．close も同期のためのチャネルを受け取る．

fopenW を使い，ファイル書き込みは以下のように書くことができる：

```
helloWriter :: String -> PiMonad ()
helloWriter filename = do
  (ret::O (O (String, O ()), O (O ())))
    <- new
  send fopenW (filename, ret)
  (write,close) <- recv ret
  (sync::O ()) <- new
  send write ("Hello world!",sync)
  _ <- recv sync
  send_sync close
```

6. ネットワーク実装

本章では，Haskell で提供されるネットワーク API と，前章で与えた Haskell の I/O システムへの拡張を組み合わせて，PiMonad のネットワーク実装を与える．

表面上は 2 つのチャネルを加えることで実現できる．1 つは，リモートホストからの接続を待ち受けるチャネルであり，もう 1 つはリモートホストへと接続するためのチャネルである．これらを初期チャネルと呼ぶことにする．通信するホストどうしは，まずこれら 2 つのチャネルを介し相互にローカルチャネルを渡しあう．交換したチャネルを介して，通信の続きを行う．

さらに，前章で定義した出力チャネルに，ネットワーク上の通信ホストの位置情報を含めるよう修正する．他に，Haskell のネットワーク通信に必要なスレッドを準備し，値の直列化を実装する．

6.1 基本メカニズム

直列化 Haskell のネットワーク通信は文字列の通信として記述されるため，値型として定義した型クラス ValueType に文字列への直列化と，値の復元の関数を追加する．通信は RecvBuf を直列化し行う．すなわち，各通信ごとに，ピア間で送信チャネルと値の文字列表現の対がやりとりされる．

アクションチャネルの修正と接続ホスト管理 アクションチャネルを，ネットワーク接続情報を追加するために以下のように修正する：

```
data O a =
    O Integer
  | Action
     String
     Location
     (PiPeer -> a -> IO ())
```

追加された第 2 引数 Location は，ホスト名等のネットワーク上の通信ホストの位置情報を格納するために用いる．Haskell アクションの追加引数 PiPeer は，(1) 当該ホストのホスト名，(2) 受信のために用いられるローカルチャネルと，その文字列表現を対応付けるテーブル，(3) 利用できる TCP コネクションのリストからなるデータである．

着信モニタ 評価器に TCP コネクションの待ち受けスレッド（モニタ）を追加する．コネクション要求を受け付けたとき，モニタは受信データを待ち受けるスレッド，送信のためのスレッドを 1 つずつ生成する．2 つのスレッドは，それぞれ入力動作と出力動作に対応した通信を行う．

初期入力チャネル 初期チャネルのうちの 1 つが，初期入力チャネルである．初期入力チャネルは TCP コネクション確立時に入力動作が発火される．すべてのホストは，後で導入する名前解決チャネルを介してお互いの初期入力チャネルを得，このチャネルに向けて送信を行い，コネクションを確立する．初期入力チャネルはトップレベルの PiMonad プログラムの引数としてのみ与えられる．初期入力チャネルは型 I a を持ち，型変数 a はアプリケーション固有の通信の内容を表すように使用者側で具体化する．

例 6.1 エコーバック・サーバ 初期入力チャネルで着信を待ち，文字列とチャネルの対を受け取り，チャネルを介して文字列をそのまま返すエコーバック・サーバの例を示す：

```
echo :: I (String, O String) -> PiMonad ()
echo init =
  rep init (\(str, reply) -> send reply str)
```

ここで引数 init が初期入力チャネルである．多相的な型付けにより，初期入力チャネルで受け取る型は任意の型をとることができる．

名前解決チャネル もう 1 つの初期チャネルが，名前解決チャネル resolve である．名前解決チャネルはホスト名とポート番号の対を受け取り，対応するホストで待ち受けているプロセスの初期入力チャネルのコピーを返す．このチャネルを介した出力動作が，TCP

コネクションの確立に対応する．resolve は次の型シグネチャを持つ：

```
resolve :: (ValueType a) =>
          O ((Hostname,PortNumber), O (O a))
```

型変数 a は，初期入力チャネル同様，使用者側で具体化する．

ネットワーク通信における各チャネル型の役割 初期入力チャネルはリモートホストからのコネクションを待つためだけに用いるため，入力能力型 I a を持つ．一方，通信において多く用いるのは出力能力型 O a とチャネル型 L a である．ALπ の局所性のため，他プロセスから入力プレフィクスで受信するチャネルはすべて出力能力型である．出力能力型はネットワークの通信ホストの位置情報を持つよう本章で拡張する．一方，new で生成するチャネルはチャネル型を持ち，send により出力能力のみをリモートホストに渡しそのチャネルで入力することで，継続したネットワーク通信が可能になる．

6.2 多相型付けされた初期チャネルと通信の安全性

アプリケーション固有の型を送受信する余地を残すため，2 つの初期チャネルは多相的に型付けされている．それゆえ，それぞれのホストで期待する値型が違う可能性があり，通信が失敗するおそれがある．

そこで，初期チャネルでの通信のみ，相互の型の比較を型名のレベルで行う．初期チャネルの型が整合していれば，継続する通信の型はつねに整合していると推測される．

6.3 問 題 点

名前の枯渇 π 計算は加算無限個の名前の集合を仮定している．本フレームワークにおいて名前は Haskell 処理系の実装に依存した有限の整数で表現されるため，サーバプロセス等長時間動作するプロセスでは名前の数が足りなくなることがある．

振舞い等価性等の諸規則により使われない名前を特定してごみ集めすることはある程度可能である．しかしながら，I/O システムやリモートホストに送信された名前についての参照カウント等を用いて管理しなければ，名前の枯渇を防ぐことはできない．

プロセスの溢れ 同様に，それ以上動作しないプロセスがメモリ領域を占有することがある．そうしたプロセスは π 計算の振舞い等価性により削除できる．たとえば，$(\boldsymbol{\nu} x : L)(!x(y).P) \cong_c \mathbf{0}$ より，通信プロセス ([NewChan x], ReprRecv x (\y->p)) はこれ以上相互作用しないことが保証されるため，系から取り除くことができる．

7. プログラム例

本章では，構築したフレームワークを用いてインスタント・メッセンジャの例を示す．プログラムはクライアントがサーバに接続することでメッセージをやりとりする．キーボードや画面表示は単一チャネルで表されているものとし，ネットワーク通信に関係のある部分のみ取り上げる．

7.1 メッセージ交換プロセス

図 14 で与えられるプロセス imMain はサーバとクライアントの両方で実行される．このプロセスはメッセージ交換に共通する部分を与える．最初の 2 つの引数はそれぞれユーザへの入力と出力を表すものとする．残りの 2 つの引数は，リモートのユーザ入力と出力である．imMain は並列に起動される receiver と sender によって成り立つ．

receiver はループ用チャネル rloop で待った後，リモートホストからチャネル rin を介し入力を待つ．メッセージが来ると，ただちにユーザに送られ，メッセージが表示されるまで待つ．表示が完了したら，rloop に値 () を送り次のサイクルを起動する．

sender はループ用チャネル sloop で待った後，ユーザ入力へチャネルを送り，待つ．ユーザ入力がなされた時点でリモートホストへ入力文字列を送る．この際 sloop をリモートホストに送ることで，リモートホストからの応答があった時点でただちに次のサイクルが起動される．

7.2 スタートアッププロセス

図 15 の server，client はそれぞれサーバおよびクライアントのスタートアッププロセスである．それぞれ初期チャネルを用いて通信を開始する部分である．サーバ，クライアントともに仮引数 cin，仮引数 cout はそれぞれローカルのユーザ入力と出力に対応するものとする．

server は，初期入力チャネル incoming よりクライアントへメッセージを送るチャネル rout と，クライアントからのメッセージを受け取るチャネルを送るチャネル orin の対を受け取る．ユーザにコネクションが来た旨を伝えた（"connection comes"）後，クライアントからのメッセージを受け取るチャネル rin を受け取り，imMain へと継続する．

client は，引数 hostname および portnumber で与えられるホストの初期チャネルを，名前解決チャネル resolve1 を介して送ったチャネル res より取得する．resolve1 は多相的に型付けされた resolve チャネルを具体型にしている．初期チャネルが取得できた

```
imMain ::
  (I (String, O ())) -> (O (String, O ())) ->
  (L (String, O ())) -> (O (String, O ())) -> PiMonad ()
imMain cin cout rin rout = do
    (rloop::L ()) <- new
    rep rloop (receiver rloop)
    (sloop::L ()) <- new
    rep sloop (sender sloop)
    send rloop ()
    send sloop ()
  where
    receiver :: L () -> () -> PiMonad ()
    receiver rloop _ = do
      (str, sync) <- recv rin
      (sync::L ()) <- new
      send cout (str, sync)
      _ <- recv sync
      send rloop ()
    sender :: L () -> () -> PiMonad ()
    sender sloop _ = do
      (inp::L String) <- new
      send cin inp
      str <- recv inp
      send rout (str, sloop)
```

図 14 メッセージ交換プロセス `imMain`
Fig. 14 `imMain`: the core of the IM program.

とき，ユーザに文字列 `"connected."` を通知し，チャネルのやりとりによりリモートからの受信チャネル `rin`，リモートへの送信チャネル `rout` を得て `imMain` へと継続する．

7.3 従来の Haskell ネットワークプログラムとの比較

Haskell のネットワーク API を用いた通信では，文字列と値の相互変換を行う必要がある．通信手順の誤りにより，たとえば，整数型の値を受信しようとしているピアに対し，文字列型の値を送信し，値の復元に失敗し実行時エラーとなる場合がある．

本フレームワークでは，初期チャネルの型が一致していれば，データの直列化と復元に失敗せず，後の通信が失敗しないと期待できる．

8. 関 連 研 究

本章では，関連研究として，プロセス計算に基づくプログラミング言語と，Haskell のモナドによる並行プロセスの表現に関する研究をあげる．

8.1 プロセス計算に基づくプログラミング言語

π 計算に基づく既存のプログラミング言語としては，Pict [12]，Join 計算[3]，Nomadic Pict [16] や Nepi [7] が知られている．Nomadic Pict は，Pict の拡張で分散実装を持ち，π 計算を拡張した Nomadic π 計算に基づき，モバイルエージェントの記述が可能である．Pict では足し算や引き算等の基本的な演算をもチャネルとして提供しているため，より細かく π 計算の代数法則や遷移規則による解析が可能であると考えられる．一方，PiMonad は Haskell の豊富な関数とデータ構造を用いることができるという利点がある．

Join 計算も非同期で局所的な性質を持つ計算に基づいている．Join 計算は CHAM の動作に基づいてできるだけシンプルなメカニズムで柔軟な同期機構を統一的に記述することを目的としている．これに対して，PiMonad は，埋め込み言語として，Haskell の拡張として定義し，Haskell とネットワーク通信をシームレスで記述することを目的としている．

Nepi は Lisp 処理系上に実装された，同期 π 計算

```
server ::
  O (O (String, O ())) -> O (String, O ()) ->
  I (O (String, O ()), (O (O (String, O ()))))) -> PiMonad ()
server cin cout incoming = do
  (rout,orin) <- recv incoming
  (sync::L ()) <- new
  send cout ("connection comes.", sync)
  _ <- recv sync
  (rin::L (String, O ())) <- new
  send orin rin
  imMain cin cout rin rout

client :: String -> Int ->
  O (O (String, O ())) -> O (String, O ()) ->
  I () -> PiMonad ()
client hostname portnumber cin cout _ = do
  (res::L (O (O (String, O ()), O (O (String, O()))))))  <- new
  send resolve1 ((hostname,portnumber), res)
  remote <- recv res
  (sync::O ()) <- new
  send cout ("connected.", sync)
  _ <- recv sync
  (rin::L (String, O ())) <- new
  (irout::L (O (String, O()))) <- new
  send remote (rin, irout)
  rout <- recv irout
  imMain cin cout rin rout

resolve1 :: O ((String, Int),O (O (O (String, O ()), O (O (String, O()))))))
resolve1 = resolve
```

図 15 スタートアッププロセス server および client
Fig. 15 server and client: the startup processes of IM program.

に基づくプログラミング言語である．関数型言語上に構築された点において本研究との類似がある．しかしながら Lisp は型システムを提供しないため，Nepi には型システムの概念がない．一方，Nepi は同期通信や非決定的選択プリミティブを提供し，より制御構造を見通し良く記述できるとしている．

8.2 モナドによる並行プロセスの表現

継続モナドにより並行モナドを構成する手法は文献 2) や 14) でも述べられている．これらに対して，我々の手法では並行プリミティブに非同期 π 計算のものを用い，チャネル型の概念による安全な通信が期待できる．さらに，状態モナドを組み合わせることで名前生成を可能にした．

9. まとめと今後の課題

本章では，まとめと今後の課題を述べる．

9.1 ま と め

本論文で，我々は非同期局所化 π 計算（ALπ）に基づくネットワークプログラミングフレームワークとして PiMonad および非同期局所化 π 計算に基づく型付け手法を提案した．PiMonad を定義することで Haskell に対する埋め込み言語として ALπ 計算の演算子を実現した．ALπ では，型付けを行うためにサブタイプ関係が必要であるため，ALπ の型付けにおけるサブタイプ関係の表現には Haskell の標準的な拡張である多引数型クラスを用いた．本手法は単純な coercion

による定義ができない一般のサブタイプ関係については適用できない．本フレームワークでは，データに基づくサブタイプ関係を仮定しないことで Haskell で ALπ の型システムを表現することに成功した．

ALπ の操作意味論に基づく評価器によって，Haskell の `IO` アクションを π 計算のチャネルとしてモデル化した．プログラマは任意のアクションをチャネルとして記述でき，それ自身も他のチャネルを介して通信できる．具体例としてファイル書き込みチャネルの例を示した．さらにネットワーク通信の実装において，2 つの初期チャネルを用いたネットワークプログラミングを実現した．

例としてインスタント・メッセンジャの例を示した．

9.2 今後の課題

型付き ALπ 計算のラベル付き遷移関係と評価器による PiMonad の実行の間には，直観的には双模倣の対応があると予想される．しかしながら，異常終了等の実際の評価器の振舞いが十分抽象的に形式化されていないため，検証は今後の課題とする．

この後さらに，型安全性を保証することが課題としてあげられる．たとえば，PiMonad で記述したプログラムは，`String` 型とチャネル型の取り違え等に起因する実行時エラーが生じない．このようなエラーは一般のネットワーク通信プログラミングでは起こりうるため，静的な解析による検証は有益である．PiMonad で導入した値の Haskell における型判定と，ALπ の型判定の対応関係を定式化しこれを証明する．

一方，現在の実装では 6 章であげたようなメモリ管理上の問題点があるため，チャネルの生存管理を行う必要がある．

他に，π 計算の規則を利用したプログラムのより詳細な解析・検証があげられる．たとえば，文献 4) や 5) の機密度を付加した型システムを導入し，機密性を保証するフレームワークを構築することがあげられる．

謝辞 多くの助言と，様々な相談にのっていただいた阿草研究室の皆様に，深く感謝いたします．また，有益なコメントをくださった査読者の皆様に感謝いたします．本研究の一部は，科学研究費補助金基盤研究（B）17300006，基盤研究（C）16500027，文部科学省リーディングプロジェクト e-Society「高信頼 WebWare 生成技術」および，栢森情報科学振興財団の助成による．

参 考 文 献

1) Boudol, G.: Asynchrony and the pi-calculus, Technical Report 1702, INRIA, Sophia-Antipolis (1992).
2) Claessen, K.: A poor man's concurrency monad, *Journal of Functional Programming*, Vol.9, No.3, pp.313–323 (1999).
3) Fournet, C. and Gonthier, G.: The reflexive CHAM and the join-calculus, *POPL '96: Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.372–385, ACM Press (1996).
4) Hennessy, M.: The security pi-calculus and non-interference, *The Journal of Logic and Algebraic Programming*, Vol.63, No.1, pp.3–34 (2005).
5) Hennessy, M. and Riely, J.: Information flow vs. resource access in the asynchronous pi-calculus, *ACM Trans. Program. Lang. Syst.*, Vol.24, No.5, pp.566–591 (2002).
6) Honda, K. and Tokoro, M.: On asynchronous communication semantics, *ECOOP '91: Proc. Workshop on Object-Based Concurrent Computing*, London, UK, pp.21–51, Springer-Verlag (1992).
7) Kawabe, Y., Mano, K. and Kogure, K.: The Nepi 2 programming system: A π-calculus-based approach to agent-based programming, *FAABS 2000*, LNCS, Vol.1871, pp.90–102 (2001).
8) Merro, M.: Locality in the π-calculus and Applications to Object-Oriented Languages, Ph.D. thesis, Ecole des Mines de Paris (2000).
9) Milner, R.: Functions as processes, *Mathematical Structures in Computer Science*, Vol.2, No.2, pp.119–141 (1992).
10) Milner, R.: *Communicating and mobile systems: The π-calculus*, Cambridge University Press (1999).
11) Pierce, B.C.: *Types and programming languages*, chapter 15, pp.200–206, MIT Press (2002).
12) Pierce, B.C. and Turner, D.N.: Pict: A Programming Language Based on the Pi-calculus, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp.455–494 (2000).
13) Sangiorgi, D. and Walker, D.: *The π-calculus: A Theory of Mobile Processes*, Cambridge University Press (2001).
14) Scholz, E.: A concurrency monad based on constructor primitives, or, being first-class is not enough, Technical report, University of Berlin (1995).
15) Jones, M., Jones, S.P. and Meijer, E.: Type classes: Exploring the design space, *Proc. 2nd Haskell Workshop*, Amsterdam (June 1997). Available on the web from http://www.cse.ogi.edu/~mpj/pubs/multi.html

16) Unyapoth, A. and Sewell, P.: Nomadic Pict: Correct Communication Infrastructure for Mobile Computation, *POPL '01: Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, pp.116–127, ACM Press (2001).
17) Yoshida, N.: Minimality and separation results on asynchronous mobile processes — representability theorems by concurrent combinators, *Theor. Comput. Sci.*, Vol.274, No.1-2, pp.231–276 (2002).

(平成 18 年 1 月 13 日受付)
(平成 18 年 8 月 8 日採録)

今井 敬吾

2004 年名古屋大学工学部物理工学科卒業．2006 年同大学大学院情報科学研究科情報システム学専攻博士前期課程修了．現在，同大学院情報科学研究科情報システム学専攻博士後期課程在学中．プロセス代数，プログラミング言語の基礎理論，関数型言語を用いたシステム開発に興味を持つ．

結縁 祥治（正会員）

1990 年名古屋大学大学院博士課程満了．名古屋大学大学院工学研究科助手，1998 年同情報メディア教育センター助教授を経て，現在，同大学院情報科学研究科助教授．博士（工学）．並行計算モデルのソフトウェアへの応用面の観点で，通信プロセスモデルの研究に従事．形式的な手法によるモデル化に基づくソフトウェア検証に興味を持つ．

阿草 清滋（正会員）

1970 年京都大学工学部電気第二学科卒業．同大学大学院に進学．1974 年より京都大学工学部情報工学科に勤務．同助手，助教授を経て，1989 年より，名古屋大学教授．現在，名古屋大学大学院情報科学研究科情報システム学専攻教授．工学博士．専門はソフトウェア工学．特に要求分析，仕様化技術，再利用技術に関する研究に従事．現在は高信頼性 web ソフトウェアの実現を進めている．電子情報通信学会，ソフトウェア科学会各会員．