ICOT Technical Report: TR-462

TR-462

ストリームとオブジェクト

吉田かおる、近山　隆

March, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ストリームとオブジェクト

吉田かおる　　近山隆

新世代コンピュータ技術開発機構
港区三田 1-4-28 三田国際ビル 21 階
email: {yoshida,chikayama}@icot.junet

概要

$A'UM$ は並列計算とオブジェクト抽象化をストリーム計算 (あるいは半順序集合演算系) の微視的な視点から捉えた, 並列オブジェクト指向プログラミング言語である. 本稿は, $A'UM$ の計算モデルおよび言語を紹介しながら, ストリームとオブジェクトの融合は並列言語として高い表現力と高い処理効率の両者を追及する上で重要な鍵となることを明らかにする.

1　はじめに

並列計算機システム, 並列プログラミング言語に対する関心が最近高まりつつも, 議論の多くがこれまでの長い逐次型計算機システムの歴史を上台としてなされているように思う. また, オブジェクトによる計算の抽象化はそのあまりの自然さからすんなり受け入れ, オブジェクトとは何かを問う間もなく, ソフトウエア作りへの貢献度をまず云々される傾向にある. 我々は並列をゼロから探りたい. そもそも並列とは? 並列システムとは? 並列プログラミングとは? 並列における抽象化とは何か"

(並列) システムを可能な限り分解していくと何が残るか? このような非常に微視的な視点から並列システムを眺めていく, その結果として, (並列) 計算機システムを何から構成すべきかを知りたいからである.

$A'UM$ はストリームを計算の基礎として, 並列計算とオブジェクトの抽象化をストリームの視点から捉えた, 並列オブジェクト指向計算モデル / 言語である[Yoshida88b, Yoshida89b].

本論文は, $A'UM$ の計算モデルおよび言語を図解と例を交えて説明する. 2 章に計算モデル, 3 章と 4 章に言語を述べる. 5 章でストリームとオブジェクトの関係を明らかにする.

2　計算モデル

2.1　並列システムと半順序集合

並列システムとは, 一つ以上の事象が独立に発生しうる系である. ここで "独立" とはその間に順序関係を規定できないことを意味する.

例えば, 三つの事象 a,b,c が発生する系があり,

- a は b より後に起こらなければならない.
- c は a や b とは独立である.

なる事象間の順序関係があるとしよう.

一般に, 任意の要素間に順序関係が規定できる集合は全順序集合あるいは鎖 (chain) と呼ばれ, また順序関係を規定できないような要素を含む集合は半順序集合 (poset) と呼ばれる.

a,b,c それぞれ一つの事象からなる三つの集合 $A = \{a\}, B = \{b\}, C = \{c\}$ を考えると, 上記の系 S は半順序集合演算として次のように表現できる.

$$S = (A \dotplus B) + C$$

ここで, $\dotplus$ は順序和 (ordinal sum), $+$ は基本和 (cardinal sum) と呼ばれ, それぞれ半順序集合に対する異なる種類の加算演算子を示し, その演算結果は半順序集合である[Birkhoff79]. すなわち, 並列システムは事象の半順序集合であり, 並列システムは半順序集合の演算系として定義することができる.

鎖は順序和だけから構成されるものであり, 半順序集合は鎖の基本和に分解できる. この様子は, 鎖を具体化したストリームを用いて自然に表現される.

まず, 水の流れ (川), 電子の流れ (電流), 車の流れ (交通) など, どく身近な例から, ストリームの具体的イメージを, 図 1に示すような川でよい.

川は水から, さらには水の粒すなわち水滴から成っている. 一粒の水滴がもう一粒の水滴にくっついて, 最後に連続した流れを作る. 上流から下流へ見渡すと, 二本の支流ごとに合流して, 最後に本流は海へ流れ込んでいる. ここで上流と下流という言葉は流れの方向を示している. ある支流からの一粒と別の支流からの一粒はどちらが先に海に辿り着くかわからない.

それぞれの水滴が海に辿り着く度に海をその様相を変えていく.

$A'UM$ では川の代わりにメッセージの流れを扱う. 海をオブジェクトに, 水滴をメッセージとみなせばよい.

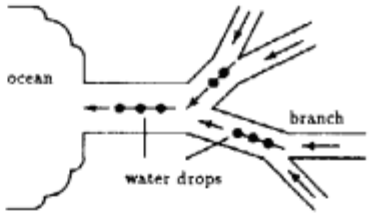

図 1: 川と海

2.2　ストリーム

ストリーム (stream) はメッセージの鎖である.

メッセージの順序関係を示すストリームの方向は, 二つの端末である入力端 (inlet) と 出力端 (outlet) によって表現する (図 2). ここで, 入力端は "^" 付き変数 (例えば, "X) で, 出力端は ただの変数 (例えば, X) で指定する.

ストリーム両端は, ストリームからメッセージの受信しストリームへメッセージを送信する オブジェクトから見て入ってくる端であるか出ていく端であるかという視点から, 頭部を入力端, 末尾を出力端と呼んでいる.

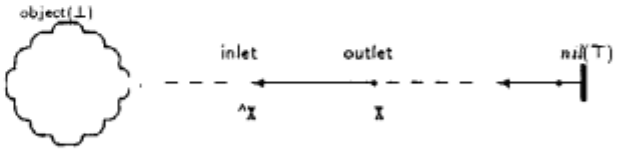

図 2. ストリーム表現

2.3 ストリーム演算

ストリーム計算はストリームの生産と消費からなる (図 3).

送信 (send): 出力端 X へメッセージ m を送信し, これに続くストリームの入力端を ^Y とする.

閉鎖 (close): 出力端 X を閉鎖する. この時, 出力端を閉鎖する際に残る状態を 空 (nil) と呼び, このストリームがもはやアクセスできないことを示す.

接続 (connect): 入力端 ^Y を出力端 X へ接続する. この時, 入力端 ^Y に続くメッセージは出力端 X より前に位置するメッセージに連結される.

受信 (receive): 入力端 ^X からメッセージを受信し, それに続くストリームの出力端を Y とする.

閉鎖検出 (is closed): 入力端 ^X が閉鎖されていることを検出する.

このように, ストリーム計算は入力端, 出力端および空の二種類の用語を使って説明される.

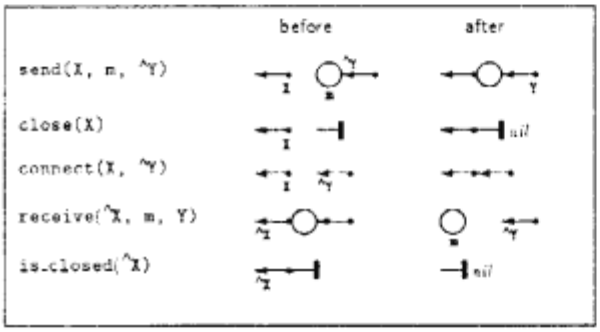

図 3: 基本演算

2.4 ジョイント

ストリームの木を構成するためにストリームの二項演算を二種類定義し, これらをジョイント (joint) と呼ぶ (図 4).

併合 (merge): 入力端 ^X と入力端 ^Y それぞれからの, メッセージを非決定的順序で併合し, 順次, 出力端 Z に送信する. ただし, ストリーム ^X およびストリーム ^Y におけるメッセージの順序関係は保たれる.

連結 (append): 入力端 ^Y からのメッセージが入力端 ^X からのメッセージのどれよりも後になるように, 順次, 出力端 Z に送信する. ただし, ストリーム X および ストリーム Y におけるメッセージの順序関係は保たれる.

これらの二項演算 (とりわけ, 併合演算) は

$$f : Chain \times Chain \rightarrow Poset, g : Poset \rightarrow Chain$$

の二種類の関数の合成 $g \circ f$ で表される. 鎖を半順序集合に一般化すると, 併合演算は基本和 (cardinal sum) に, 連結演算は順序和 (ordinal sum) に対応する.

ストリームが鎖であるのに対して, ジョイントにより構成される半順序集合を チャネル (channel) と呼ぶ.

2.5 メッセージ

メッセージ (message) はメッセージ名とストリーム端末 (入力端あるいは出力端) の列をその引数として持つ.

一個も引数を持たないもの, すなわち, メッセージ名だけから成るメッセージを 原子メッセージ (atomic message) と呼び, 引数を有するメッセージを 合成メッセージ (compound message) と呼ぶ.

各メッセージはメッセージ名と引数の数および 受信者から見た各引数の方向により識別される. メッセージ名が一致しても, 引数の数あるいは方向が異なれば別のメッセージと解釈する.

各メッセージは, 送信者が実引数として渡すストリーム端と受信者に仮引数として渡されるストリーム端を接続するストリーム接続器として働く. 二本のストリームの接続は一方の入力端と他方の出力端が与えられて可能となるので, メッセージの送信者と受信者はそれぞれの引数に対して, 逆方向を指定する必要がある.

例えば, 図 5 の m(X, ^Y, Z) はメッセージ名 m を持ち, 二個の出力端 X と Z, 一個の入力端 ^Y の合わせて三個の引数を保持する. したがって, m(-+-) のように識別名が与えられる. メッセージの送信者が m(^U, V, ^W) を指定すれば, ^U が X に, V が ^Y に, そして ^W が Z にそれぞれ接続される.

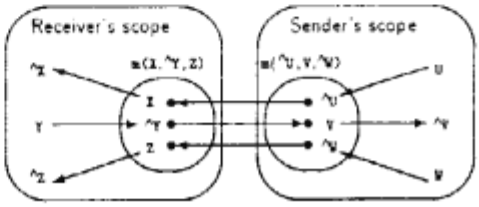

図 5: ストリーム接続器としてのメッセージ

2.6 オブジェクト

オブジェクトは繰返し計算を抽象化したものである. オブジェクトは, その生成時に, 外部とのインタフェースとして働くインタフェース・ストリームと呼ばれる, 一本の入力端を与えられる. 内部にはオブジェクトの状態を表すスロット群を保持することができる.

送信者の立場から, あるオブジェクトへのストリーム (の出力端) はオブジェクトそのものに見える. あるオブジェクトと知り合いになるということはそのオブジェクトへのストリーム (の出力端) を獲得することである. あるオブジェクト (A) を別のオブジェクト (B) に紹介することは, A へのストリームを分岐し, その片方を B に渡すことである.

2.6.1 オブジェクトの生成

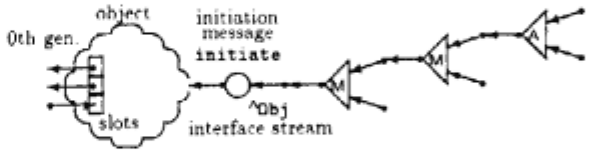

図 6: オブジェクトの外観

クラスにインスタンス生成メッセージ new(^Obj) を送信することにより, そのクラスのインスタンス・オブジェクトが生成される.

この時, 生成されたばかりのオブジェクト (これを 第 0 世代 と呼ぶ) には暗黙裏に起動メッセージ initiate が送られ, それに続くストリームの入力端が ^Obj とされる. 起動メッセージはオブジェクトの内部状態の初期化のために送信される.

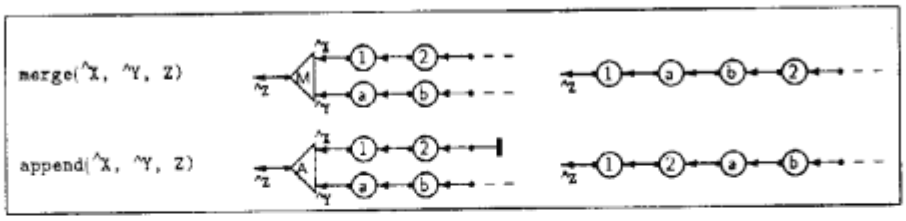

図 4: ジョイント

2.7 世代

オブジェクトは以下に述べる二段階を一周期とする周期の繰返しである．ここで周期のことをオブジェクトの世代 (generation) と呼ぶ．

1. 受動期　インタフェース・ストリームに関する事象，すなわちメッセージの受信あるいは閉鎖検出を待つ．

2. 能動期:

   世代はインタフェース・ストリームに関する事象を観測した後，その観測事象に応じた一連の動作を実行する．一連の動作とは，

   - 任意個の送信，閉鎖，接続，併合，連結および基底オブジェクトの生成動作
   - 一個以下の世代降下

   から成る．各動作は任意の順序あるい並列に実行される．

   [注] 一般オブジェクトの生成は，クラス・オブジェクトの生成とそれへのメッセージ送信に分解される

世代は，インタフェース・ストリームの入力端，オブジェクトの内部状態を表すストリーム端から成るストリーム端の集合体である．

2.7.1 メソッド

一世代の振舞いを定義したものを **メソッド** (method) と呼ぶ．メソッドは観測すべき事象を定義する受動部の記述とその観測事象に応じて執るべき動作を定義する能動部の記述から成る．

2.7.2 世代降下

ある世代が次の世代を生成する動作を **世代降下** (generation descending) と呼ぶ．世代降下は他の動作と任意の順序あるいは並列に実行される．

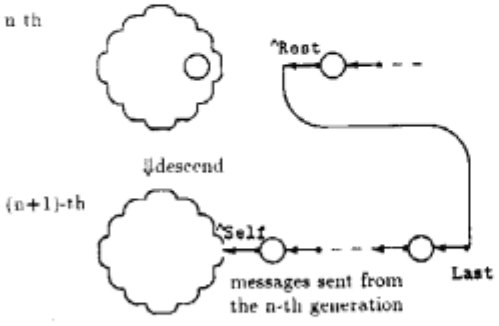

図 7: 世代降下

2.7.3 自分自身 (Self)

ある世代のオブジェクトにとって，**自分自身** (Self) とは次世代を意味する．自分自身にメッセージを送るということは次世代へ繋がるストリームの出力端にメッセージを送ることに他ならない．

自分自身の分岐:　自分自身を分岐して，その片方の出力端を受信メッセージあるいは送信メッセージの引数に接続すれば，その送信者あるいは受信者は，将来この自分自身にメッセージを送信してくることが可能となる．

2.7.4 継続と終了

オブジェクトは，オブジェクトが終了メッセージを受信するまで世代降下する．

オブジェクトの終了:　オブジェクトは終了メッセージ ($terminate) を受信すると，それが保持するストリーム端をすべて充満化する．すなわち，出力端は閉鎖し，入力端は回収する．インタフェース・ストリームも同様に回収する．

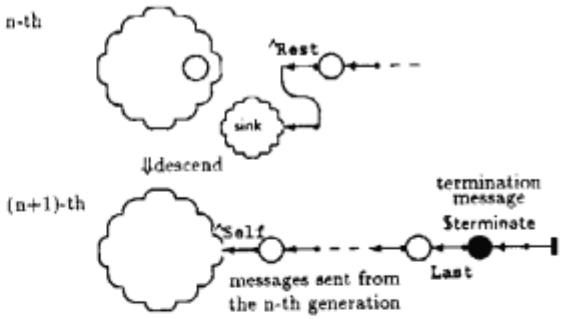

図 8: オブジェクト終了の起動

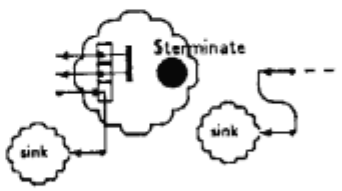

図 9: オブジェクトの終了

したがって，オブジェクトの一生は次のような世代連鎖として示すことができる

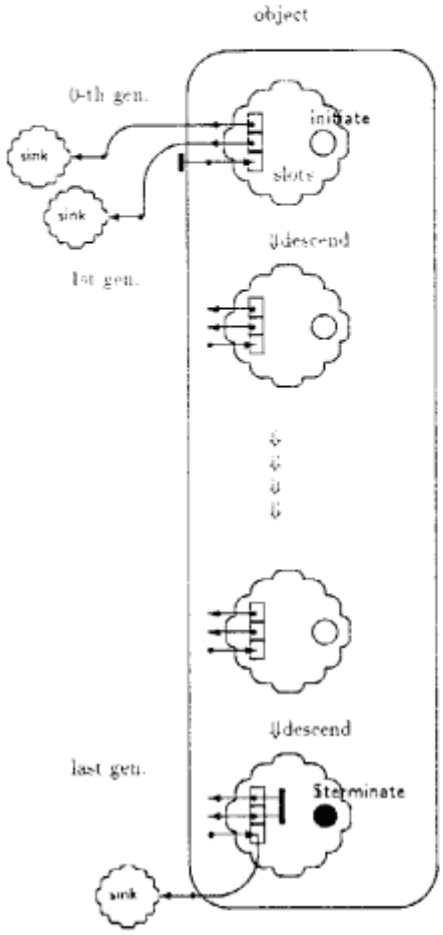

図 10: 世代連鎖としてのオブジェクト

2.8 スロット

オブジェクトはその状態を表すスロット群を内部に保持できる．各スロットはストリーム端を名前で連想させて保持する．入力端と出力端のどちらを保持するかにより **入力端スロット** あるいは **出力端スロット** と呼ぶ．

スロット名はスロットが定義されるクラス内で一意に決まる．すなわち，各スロットはクラス名とスロット名により識別される．

スロットへのアクセスは，スロット・アクセスのためのメッセージをオブジェクト自身すなわち次世代へ送信することにより実現される．

2.8.1 入力端スロット

初期化: オブジェクト生成時，各入力端スロットは閉鎖されている．

参照: 入力端スロット参照メッセージ `get_inlet(Name, Slot)` がオブジェクトに送信されると，現在の入力端をアクセサが手渡す出力端 (メッセージの第一引数) に接続する．新しいスロットは閉鎖されている．

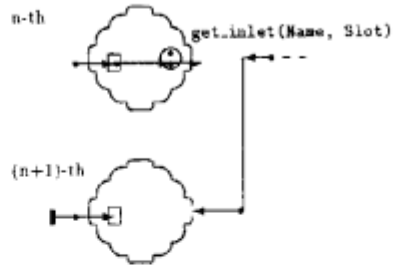

図 11: 入力端スロットの参照

更新: 入力端スロット更新メッセージ `set_inlet(Name,^Slot)` がオブジェクトに送信されると，現在の入力端をシンク・オブジェクト (後述) へ接続する．アクセサが手渡す入力端 (メッセージの第二引数) を新しいスロットへ接続する

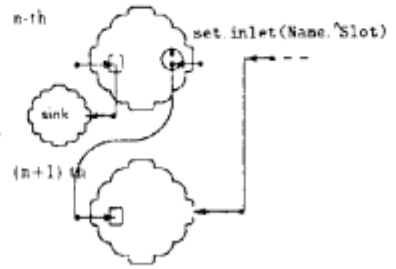

図 12: 入力端スロットの更新

後始末: オブジェクト終了時，各入力端スロットはシンク・オブジェクトへ接続される．

2.8.2 出力端スロット

初期化: オブジェクト生成時，各出力端スロットはシンク・オブジェクトへ接続されている

参照: 出力端スロット参照メッセージ `get_outlet(Name,^Slot)` がオブジェクトに送信されると，現在の出力端のストリームは分岐し，アクセサが手渡す入力端 (メッセージの第二引数) を片方へ接続する．新しいスロットをもう片方へ接続する．

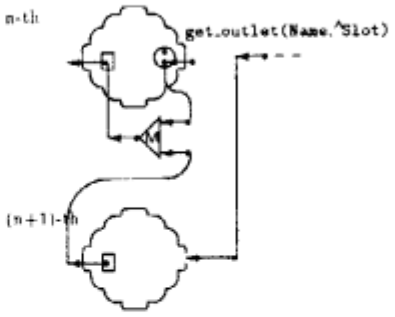

図 13: 出力端スロットの参照

更新: 出力端スロット更新メッセージ `set_outlet(Name, Slot)` がオブジェクトに送信されると，現在の出力端を nil で閉鎖する．アクセサが手渡す出力端 (メッセージの第二引数) は新しいスロットへ接続する．

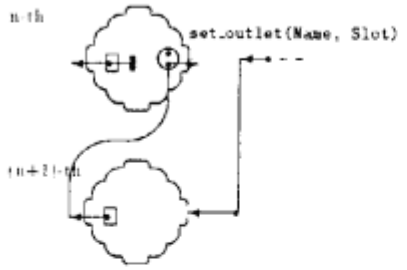

図 11: 出力端スロットの更新

後始末: オブジェクト終了時，各出力端スロットは閉鎖される．

2.9 原始オブジェクト

Ailuli ではストリームが基底であり，あらゆるものがストリームをインターフェースとしてメッセージ交信しあうオブジェクトである．

一般に原始データとして扱われる，整数 (例えば，5)，アトム (例えば，a)，ブール値 (例えば，'true)，ストリング (例えば，"hi") などは **原始オブジェクト** と呼ばれ，他の抽象オブジェクトと全く同様にストリームをインタフェースとするオブジェクトである．クラスもまた原始オブジェクトである．

例えば，整数 5 をプログラムに指定すると，それは 整数オブジェクト 5 を生成し，それへの出力端を示す．その出力端に加算メッセージadd(1, "Sum) を送信すると，オブジェクト 5 はその加算メッセージをいつか受信し，加算結果 6 の整数オブジェクトを生成し，それへの出力端が "Sum となる．被加算数 1 そして加算結果 6 に対しても同様である．

原始オブジェクトに関する操作のために簡易表現が提供されており，ストリームをほとんど意識せずに操作が行なえるようになっている．

リストおよびベクタは原始オブジェクトから合成される抽象オブジェクトであるが，同様に簡易表現が提供されている．

2.10 ストリームの完備化と計算の終了

半順序集合は鎖に分解できる．集合全体として順序関係の下で最小元が存在し，それぞれの鎖に最小上界 (鎖中のすべて要素より順序関係で大きい集合の最小元) が与えられるとき，これは完備化された半順序集合 (CPO) と呼ばれる．また，集合全体の最小元とその各鎖の最小上界を与えることは半順序集合の完備化と呼ばれる

最小元は計算の開始，集合自体は計算過程，最小上界は計算の終了とみなすと，最小元が決まらないとは計算が開始しない，また最小上界が決まらないとは計算が終了しないと解釈できる

入力端と出力端のいずれか一方しか現れないストリームが存在する場合，次のような弊害を引き起こす．

出力端の放置 —
その入力端からのメッセージ受信あるいは閉鎖検出をあるオブジェクトが期待し続けるかもしれないので，デッドロックを招く可能性がある．

入力端の放置 —
流れ込んで来るメッセージ群は誰にも受信されず永久的ゴミ (一括 GC の対象) となる．

そして，これらのメッセージ中に引数として含まれるストリームの入力端および出力端も放置されることになる．メッセージの送信者は，受信者がこれら引数として運ばれる入力端を何かのオブジェクトへ接続したり出力端を閉鎖することを期待している．したがって，やはりデッドロックを招く可能性がある．

ストリーム計算において，それぞれのストリームの入力端と出力端の接続先を決定することを **ストリームの完備化** と呼ぶ．入力端あるいは出力端のない不完備なストリームの最上界および最下界は次のように定める

最上界は空: 放置された出力端は閉鎖する．

最下界はシンク・オブジェクト: 放置された入力端はシンク・オブジェクトに接続する．

2.10.1 シンク・オブジェクト

シンク・オブジェクト (sink object) はメッセージを解釈能力を有しメッセージを回収する役割を果たすオブジェクトである．シンク・オブジェクトの各世代は，

- メッセージを受信した場合，それが引数として保持するストリーム端のそれぞれに対して次のような動作を行なうとともに世代降下する．
  - 入力端であれば，新たにシンク・オブジェクトを生成し，それへ接続する．これを **入力端 (あるいはストリーム) の回収** と呼ぶ．
  - 出力端であれば，閉鎖する．
- 閉鎖を検出した場合，終了する．

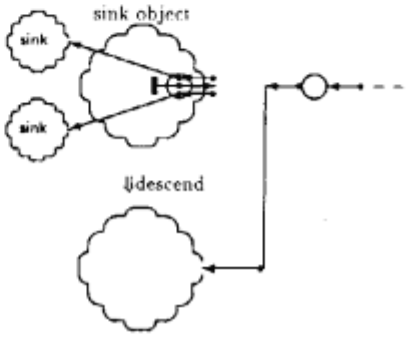

図 15: シンク・オブジェクト

2.11 クラス

クラスはインスタンスの型を定義するもので，
- 継承すべきクラス群の定義
- 保持すべき入力端スロットおよび出力端スロット群の定義
- オブジェクトの世代を定義するメソッド群の定義

から成る．

クラスはストリームをインタフェースとする原始オブジェクトである．インスタンス生成メッセージを受信すると，その型定義に基づいてインスタンスを生成する．

クラスのクラスであるメタクラスの概念はない．

2.12 継承

Ailuli はプログラムコードを最小化するという目的から，多重クラス継承を支援している．

一つのクラスは，任意個のクラスからメソッド群を継承できる．クラス継承により，オブジェクトに適用可能なメソッド群およびアクセス可能なスロット群の空間は広がるが，継承する上位クラスに対応してそれぞれインスタンスを生成するわけではない．クラス継承はストリーム計算によって直接影響を受けるものではない．

2.12.1 継承木

クラスで定義されるクラスの継承関係は，一般に，そのクラスを根とする木構造をなす．これを **クラス継承木** と呼ぶ．クラス継承木は根を起点として深さ優先かつ左優先で継承順序が与えられ，継承列が生成される．

例えば，

1. クラス a がクラス b c を継承
2. クラス b がクラス p, q を継承
3. クラス c がクラス r, s を継承

なる継承関係に対して，

a — b — p — q — c — r — s —

の継承列となる．このようにユーザにより明示的定義に生成される継承列を **ユーザ定義継承列** と呼ぶ．このユーザ定義継承列の両端に，

下界クラス — ユーザ定義継承列 — 上界クラス

のように，二つのクラスを暗黙裏に付加したものが各クラスにとっての継承列全体となる．これを **完全継承列** と呼ぶ．たとえ明示的に継承しないクラスでも，

下界クラス — ユーザ定義クラス — 上界クラス

なる継承列が生成されることになる

2.12.2 メソッド探索

オブジェクトは受動期において観測した事象に対して，この完全継承列から適用可能なメソッドを探索し決定した後，能動期に入る．最初にどのクラスから探索を始めるかを指定できるが，指定のないときは下界クラスから探索を開始する

上界クラスおよび下界クラスは "すべてのクラスのオブジェクトに適用可能な共通メソッド群" を定義しているクラスであり，それらがユーザ定義クラスによって再定義可能か否かによって，両クラスは異なる．

2.12.3 上界クラス

上界クラスはユーザ定義継承列の最後尾に付加されるクラスである．共通メソッド群で，ユーザ定義クラスにより再定義可能なものを定義している．以下のメソッドが含まれる．

- 初期化メッセージ (initiate) に対するメソッド
- 閉鎖検出によるオブジェクトの終了を定義するメソッド
- メソッド未定義のエラー処理を定義するメソッド

2.12.4 下界クラス

下界クラスはユーザ定義継承列の最前列に付加されるクラスである．共通メソッド群で，ユーザ定義クラスでは 再定義不可能な ものを定義している．以下のメソッドが含まれる

- スロット・アクセス・メソッド
- 終了メッセージ ($terminate) に対するメソッド

3 基本文法

並列システムは事象の半順序集合であり，並列プログラミングとは事象のグラフを記述することである．したがって，言語にとって，"いかに容易にグラフを描けるようにするか" がその果たすべき第一の課題である．図表現も一種の言語であろうが，ここでは文字表現による言語を定義する．

また，実時間での最大限のゴミの回収とデッドロック回避の問題も合わせ，ストリーム・プログラミングにとってストリームの完備化が必須である．*A'U*ハイ は容易，安全かつ効率のよいプログラムが書けるように，関数的文法を定義するとともに各種支援を行なっている．

3.1 クラス定義

クラス定義は，クラス名の定義，継承クラスの定義，入力端スロットおよび出力端スロットの定義，そして一世代の振舞いを記述するメソッドの定義から成る．

```
<ClassDefinition> ::=
    class <ClassName> ";"
        [ <SuperclassDefitvon> ";" ]
        [ <InletSlotDefinition> ";" ]
        [ <OutletSlotDefinition> ";" ]
        { <Method> ";" }
    end ";"

<SuperclassDefinition> ::=
    super <SuperClassName> { ","<SuperClassName> }
<InletSlotDefinition> ::=
    in <SlotName> { ","<SlotName> }
<OutletSlotDefinition> ::=
    out <SlotName> { ","<SlotName> }
```

3.2 メソッド定義

メソッドは，観測すべき事象を定義する受動部とそれに応じて執るべき動作を定義する能動部の二つの部分から成る．

```
<Method> ::=<Event> "|" <Actions> { ","<Actions> }

<Actions> ::=<NilExpression>
```

3.3 ストリーム変数

ストリームの方向は，次のようなストリーム変数を用いて指定される．

入力端変数: "^" が頭に付いた変数名 (例えば，^X) の出現は入力端を表す．

出力端変数: 変数名だけ (例えば，X) の出現は出力端を表す．

3.4 関数的表現式

メソッドの受動部および能動部はそれぞれ関数的表現式によって定義する (表 1)．

表 1: 基本表現式

relation	*expression*	*result*
`receive(^X,m,Y)`	`':' <Message> '=' <Out>` `:m = Y`	*<Nil>*
`is_closed(^X)`	`::`	*<Nil>*
`send(X,m,^Y)`	*<Out>* `':' <Message>` `X : m`	*<Out>* `Y`
`close(X)`	*<Out>* `'::'` `X ::`	*<Nil>*
`merge(^X,^Y,Z)`	*<Out>* `'=' <In>` `Z = ^X`	*<Out>* `Y`
`append(^X,^Y,Z)`	*<Out>* `'\' <In>` `Z \ ^X`	*<Out>* `^Y`
`descend(^X,S)`	`'<==' <In>` `<== ^X`	*<Nil>*

各表現式は，その評価値として入力端，出力端あるいは空のいずれかを示し，それに応じて，**入力端表現式**, **出力端表現式** あるいは **空表現式** と

呼ばれる．複雑なグラフもこれらの表現式を組み立てることにより簡単に描くことができる．

例えば，C:up:up:up:show(^U) はメッセージ送信式を組み立てたものである．最初の C:up は出力端 C へメッセージ up を送信した後，それに続くストリームの出力端（これを C1 とする）をその評価値とする．したがって，この表現式は C1:up:up:show(^U) と書き換えられる．これを繰り返すと，表現式全体はメッセージ show(^U) を送信した後のそれに続くストリームの出力端を表すことになる．

3.4.1 "右から左" 原則

これまで示した図はいずれも，メッセージが右から左に流れて 最後に最も左にあるオブジェクトに流れつくように，すなわち各メッセージはそれより右に位置するメッセージより早くオブジェクトに受信されるように描かれている．オブジェクトの受信順序から見れば，時間が左から右に進むようにと言ってもよい．

表現式はこの原則を守るように設計されている．これにより，図を描くようにプログラムを書くことができる．

3.4.2 空の導出

基本文法は，入力端あるいは出力端が放置された不完備なストリームによる弊害を防ぎ，計算の終了を促すように，次のような規則を与えている．

> 規則 1（ストリーム変数の対の出現）: それぞれのストリーム変数は，入力端変数と出力端変数が対で出現しなければならない．
>
> 規則 2（空表現式のみの出現）: メソッドの能動部の トップ・レベルである動作列（< *Actions* > フィールド）には，"基本的に" 空表現式しか指定できない．

後に述べる，自動閉鎖および自動回収の支援を含む文法の拡張によりこれらの規則は排除される．

3.4.3 原始オブジェクト

原始オブジェクトに対応する字句の出現は，原始オブジェクトを生成しそのオブジェクトへの出力端を意味する．

3.4.4 共通メッセージ

原始オブジェクトを含めすべてのオブジェクトが受信すべきメッセージがいくつかあり，一般オブジェクトの場合，下界クラスあるいは下界クラスに対応するメソッドが定義されている．ここでは，中でも特徴的な誰何メッセージを紹介する．

誰何メッセージ: who_are_you(Who) はオブジェクトにその "表示イメージ" を尋ねるメッセージである．

例えば，整数 1 へ 1:who_are_you(Who) のようにこのメッセージを送信すると，整数 1 はこの出力端に対して 自分自身が 1 であることを伝えるメッセージ 1 を送信した後，閉鎖する．

"表示イメージ" とは，オブジェクトをそれとして認識できるものなら基本的に何でもよい．

一般オブジェクトが誰何メッセージを受信した場合，自分自身へのストリームを引数とするメッセージを返す．すなわち，"君は誰?" の問いかけに "私は私" と答えることになる．

```
:who_are_you(^Who) = Rest | <== ^Self,
                            Who:i_am(Self2)::,
                            Self = ^Self2 = ^Rest:: .
```

ここで最も簡単な例として，カウンタのプログラムを基本文法で記述してみよう．

[応用例-1 (カウンタ)]

```
class   counter.
     out n.
  :up = Rest       | <== ^Self,
                     Self:get_outlet(n, ^N)
                         :set_outlet(n, N1) = ^Rest ::,
                     N:add(1, ^N1):: .
  :down = Rest     | <== ^Self,
                     Self:get_outlet(n, ^N)
                         :set_outlet(n, N1) = ^Rest ::,
                     N:sub(1, ^N1):: .
  :set(^N) = Rest  | <== ^Self,
                     Self:set_outlet(n, N) = ^Rest :: .
  :show(N) = Rest  | <== ^Self,
                     Self:get_outlet(n, ^N) = ^Rest :: .
  ::               | <== ^Self,
                     Self:'$terminte':: .
end.

class   test.
  :test = Rest          | <== ^Self,
                          Self:testM(^Um,^Dm):testA(^Ua,^Da)
                              :nop(^Result) = ^Rest ::,
                          Result\ ^WUm\ ^WDm\ ^WUa\ ^WDa ::,
                          Um:who_are_you(WUm)::,
                          Dm:who_are_you(WDm)::,
                          Ua:who_are_you(WUa)::,
                          Da:who_are_you(WDa):: .
  :testM(U, D) = Rest | <== ^Rest,
                          ##counter:new(^Counter)::,
                          Counter:set(5) = ^C ::,
                          C1:up:up:show(^U)::,
                          C2:up:up:show(^D)::,
                          C = ^C1 = ^C2 :: .
  :testA(U, D) = Rest | <== ^Rest,
                          ##counter:new(^Counter)::,
                          Counter:set(5) = ^C ::,
                          C1:up:up:show(^U)::,
                          C2:up:up:show(^D)::,
                          C \ ^C1 \ ^C2 :: .
  :nop(Result) = Rest | <== ^Rest,
                          ##sink:new(^Result)::.
end.
```

図 16: 基本文法によるカウンタのプログラム

カウンタは，メッセージ up あるは down により，そのカウンタ値を 1 上げるか下げるかするオブジェクトである．また，メッセージ set あるいは show によりカウンタ値が設定あるいは参照される．このカウンタに対して二種類のテストをする．クラス test のメソッド testM と testA がそれである．これらのメソッドは共通して，

1. カウンタ・オブジェクトを生成し，
2. まず set(5) メッセージを送信してカウンタ値を 5 に設定した後，その出力端を分岐する．
3. 片方に，up メッセージ二個の後にメッセージ show(^U) を送信し，カウンタ値を参照してみる．

1. そちら方に，down メッセージ二個とメッセージ show(^D)を送信し，カウンタ値を参照してみる．

両メソッドの異なる点は，

- メソッド testM では，二本の分岐ストリーム (C1 と C2) は併合され，メッセージ up と down はどちらが先に幾つ到着するかは非決定的である．したがって，参照結果 ^U は {5, 6, 7} のいずれかの値を，^D は {3, 4, 5} のいずれかの値を示す
- メソッド testA では，二本の分岐ストリーム (C1 と C2) は連結され，C1 からの二個の up メッセージは C2 からの二個の down メッセージより必ず先に到着する．したがって，参照結果 ^U は 7 を，^D は 5 を示す

カウンタ・オブジェクトの第一世代がメッセージ set(5) を受信したばかりの能動態の様子を図 17 に示す．

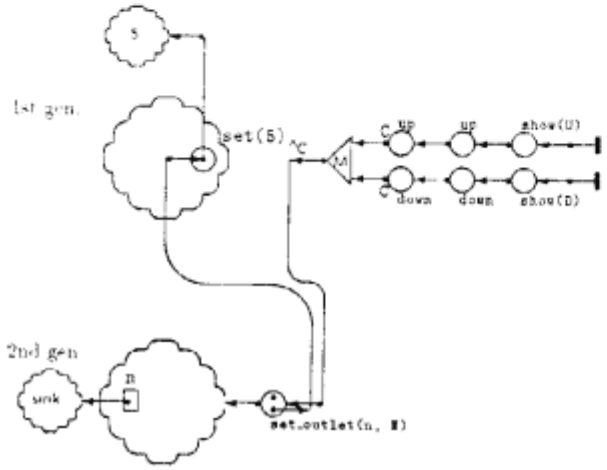

図 17: カウンタの第一世代

本プログラムは次節で述べる文法の拡張により図 18のように書き直される．

```
class   counter.
    out  n
  :up   -> !n + 1 = !n.     % X+Y => S ; X:add(Y,^S)
  :down -> !n - 1 = !n.     % X-Y => D ; X:sub(Y,^D)
  :set(^N) -> N = !n.       % !n => $self:set_outlet(n,N)
  :show(N) -> !n = ^N.      % !n => $self:get_outlet(n,^N)
end.

class  test.
  :test -> :testM(^Um, ^Dm):testA(^Ua, ^Da):nop(^Result),
           Result$1 = (Um?), Result$2 = (Dm?),
           Result$3 = (Ua?), Result$4 = (Da?).

  :testM(U, D) ->
           $counter:set(5) = ^C, C:up:up:show(^U),
                                 C:down:down:show(^D).
  :testA(U, D) ->
           $counter:set(5) = ^C, C$1:up:up:show(^U),
                                 C$2:down:down:show(^D).
  :nop(Result) -> .
end.
```

図 18: 拡張文法によるカウンタのプログラム

4 拡張文法

基本文法は，ストリーム計算モデルをそのまま表すように定義された．大規模な並列問題の記述には，より簡潔あるいは抽象化された表現が必要になる．表現力を高め，簡潔かつ安全なプログラムを書くために，基本文法に次の拡張が施されている．

- 簡易 (マクロ) 表現の導入
- ストリーム変数からチャネル変数への変数の意味の拡張
- ストリーム完備化の支援
- 擬発オブジェクトの導入

4.1 簡易表現 (マクロ)

容易にプログラムが書けるように，種々の簡易表現が提供されている．基本文法が関数的表現式から成っているように，これらの簡易表現はいずれも，ある表現式 (の列) の展開とともにその評価値として入力端，出力端あるいは空を表す．

4.1.1 算術 / 論理演算マクロ

主に，原始オブジェクトの演算機作 (例えば整数の四則演算) などのために以下に示すマクロ群が定義されている．それらのほとんどは，評価値として，演算結果オブジェクトへのストリームの出力端を示す．

前述の基本表現式と同様に，これらのマクロを組み立てて複雑な演算式を記述することができる．

例えば，3 + 5 == 8 という演算式に対して，

```
3:add(5, ^Sum)::, Sum:eq(8, ^True)::,
```

が展開され，真オブジェクト 'true へストリームの出力端 True がその評価値となる．

4.1.2 擬似変数 $self

オブジェクト自身すなわち次世代へのストリーム端は，擬似変数 $self の出現により指定され，出現箇所によりその意味は異なる．

- 出力端では参照．出力端フィールドに出現した場合，最新の自分を参照することを意味する

  メッセージ送信式の出力端を省略した場合，擬似変数 $self が指定されたことを意味する．

- 入力端では更新．入力端フィールドに出現した場合，それがこれからの自分となることを意味する．

[自己アクセス順序に関する規則]　自分自身へのアクセス順序は，文脈内での擬似変数 $self，送信先の省略あるいはスロット・アクセスなど自己アクセスに関わるマクロの出現を含め，文脈内での出現順序に応じて逐次的になるように決定される．具体的には，

- メソッドは，**事象から動作列へ**
- メソッドの動作列 (<*Actions*> フィールド列) は **左から右へ**
- メッセージの引数列は **左から右へ**
- マクロ展開は，**中から外へ**

という．メソッドを構成する表現式の評価順序に従って，自己アクセス順序が与えられる．

例えば，

```
:foo(!a, ^X) -> :fool(!b, !c, ^Y) = $self,
                $self:foo2(Y, ^Z) = $self,
                X + Z = !d.
```

なるメソッドは，次のメソッドと等価である．ただし，後で説明するが，"!"<*SlotName*> はスロット・アクセス・マクロ，"->" は，継続マクロを表す．

```
:foo(A, ^X) -> $self:get_outlet(a, ^A)
                      :get_outlet(b, ^B)
                      :get_outlet(c, ^C)
                      :foo1(B, C, ^Y)
                      :foo2(Y, ^Z)
                      :set_outlet(d, ^Sum) = $self ::,
               X:add(Z,^Sum)::  .
```

4.1.3 世代降下マクロ

世代降下を容易に書けるように，次のような継続マクロおよび終了マクロを提供している．

$$
\begin{aligned}
&< Method > ::= \\
&\quad < EventOnly > < DescendingMacro > < Action > \{ \text{","} < Action > \} \\
&< EventOnly > ::= < ReceivedMessage > \mid < DetectingClosed > \\
&< ReceivedMessage > ::= \text{":"} < MessagePattern > \\
&< DetectingClosed > ::= \text{"."} \\
&< DescendingMacro > ::= < Succession > \mid < Termination > \\
&< Succession > ::= \text{"=>"} \\
&< Termination > ::= \text{"=|"}
\end{aligned}
$$

継続 (succession) マクロ: 継続マクロは，図 7 に示すように，

1. 世代降下する
2. 現世代でメッセージを受信した場合，受信メッセージの後に続くインタフェース・ストリームの入力端 ^Rest を，次世代へのストリームの末尾にあたる出力端 Last に連結する．

という動作列を添加するマクロである．
例えば，メソッド :m -> :do . は次のように展開される．

```
:m = Rest | <== ^Self,
            Self:do = ^Rest.
```

終了 (termination) マクロ: 終了マクロは，図 8 に示すように，

1. 世代降下する．
2. 現世代でメッセージを受信した場合，シンク・オブジェクトを生成し受信メッセージの後に続くインタフェース・ストリームの入力端 ^Rest をこれに接続する．
3. 次世代へのストリームの末尾にあたる出力端 Last に終了メッセージ $terminate を送信した後，これを閉鎖する．

という動作列を添加するマクロである．
例えば，メソッド :m -| :do . は次のように展開される．

```
:m = Rest | <== ^Self ,
            ##sink:new(^Rest):: ,
            Self:do:'$terminate':: .
```

4.1.4 インスタンス生成マクロ

インスタンス生成マクロは，指定されるクラスへインスタンス生成メッセージを送信し，その時得られるインスタンスへのストリームの出力端を意味する．
例えば，#counter は

```
##counter:new(^Counter)::,
```

と展開され，クラス counter のインスタンスへの出力端 Counter を表す．

4.1.5 スロット・アクセス・マクロ

スロットへのアクセスを容易にするために，入力端スロット・アクセス・マクロ @SlotName および出力端スロット・アクセス・マクロ !SlotName を提供している．スロット・アクセス・マクロは，擬似変数 $self と同様に，出現箇所によりその意味が変わる．
出力端スロット・アクセス・マクロの場合，

- 出力端フィールドに指定されると，スロット参照を表す．
  例えば，!n = ^N は，

  ```
  $self:get_outlet(n, ^Slot), Slot = ^N
  ```

  と展開される．
- 入力端フィールドに指定されると，スロット更新を表す．
  例えば，N = !n は，

  ```
  $self:set_outlet(n, Slot), N = ^Slot
  ```

  と展開される．

4.1.6 送信先更新マクロ

スロットおよび自己に関する逐次的な動作列を文脈内で分割して記述したいことがしばしばある．例えば，

```
:foo -> !a:m1(^X) = !a ::,
        ... (actions related to X) ...
        !a:m2 = !a ::.
```

のように，スロットにあるメッセージを送信した後，その結果に関連する動作 (条件分岐を含め) の後に，そのスロットに別のメッセージを送信する場合などである．このような時，自己あるいはスロットの更新の記述をとかく忘れがちである．
そこで，

```
:foo -> !a:m1(^X),
        ... (actions related to X) ...
        !a:m2.
```

のように自己アクセスおよびスロット・アクセスに関して，メッセージ送信後を最新の自己あるいはスロットと暗黙裏に見なせば，文脈内での出現順序が直接それらの世代として反映されるようになる．
メッセージ送信式における送信先更新マクロは，送信先として最初に示される出力端の種類に応じて，送信先の更新を自動的に支援するものである．
プログラム上で直接メッセージ送信表現式として指定されていなくても，他のマクロがメッセージ送信表現式に展開される場合もこのマクロが適用される．
例えば，!n + 1 = !n は

```
!n:add(1, ^Sum)::, Sum = !n ::
```

に展開され，ここに送信先更新マクロが適用されて，

```
!n:add(1, ^Sum) = !n ::, Sum = !n ::
```

となる．

4.2 チャネル変数

これまでは，変数はストリームすなわち線を表すために使われた．ストリーム変数を使って，

> "二人の人物 (X, Y) にある問題 (P) をそれぞれの戦略により解かせ，それぞれが挙げてくる任意個の解答 (A) を集める"

という簡単な問合せを定義しよう．両者から集める解答の扱い方として，次の二通りを考える．

併合: 両者からの解答列をどちらを優先することなく任意順に集める場合. 以下のように表せる.

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A = ^A1 = ^A2 ::.
```

解答者 X および Y はそれぞれ将来.

- ストリーム P1 あるいは P2 を介して問題 P へ独立に問い合わせをし
- 解答をメッセージとして, ストリーム A1 あるいは A2 へ送信し, それらはストリーム A に併合される

ここで, 複数個の出力端変数の出現がストリームの併合演算を表すことにすると, 以下のようになる.

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P, A), Y:solve(P, A).
```

連結: X からの解答を Y からの解答より優先して集める場合. 以下のように表せる.

```
:consult(^P, ^A, ^X, ^Y) ->
    X:solve(P1, A1)::, Y:solve(P2, A2)::,
    P = ^P1 = ^P2 ::, A \ ^A1 \ ^A2 ::.
```

解答者に渡されるストリームは, 上の場合と変わらない. 異なる点は.

- 解答が送信されるストリーム A1 と A2 が連結されて, ストリーム A となる

ことである.

ここで, 序数のついた出力端変数がストリームの連結演算を表すことにすると, 以下のようになる.

```
:consult(^P, ^A, ^X, ^Y, ^Z) ->
    X:solve(P, A$1), Y:solve(P, A$2).
```

この例で明らかなように, これまでのストリーム変数を用いたプログラミングでは, 鎖からどのように併合および連結を用いて半順序集合を構成するかがその大きな部分を占めた. 変数の意味をストリーム (鎖) からチャネル (半順序集合) に拡張することにより, "解答者に一つの問題 P を渡す" とか "全解答を一つにまとめる" とかいうように, 半順序集合全体を一つと見なすことができるようになる.

プログラマは, "ストリームをどう繋ぐか" ではなく, "どのオブジェクトに何をするか" の記述に専念できるようになる. つまり, 関心がストリームからオブジェクトに移るわけである.

以上まとめると, チャネルは次のようなチャネル変数の出現により指定される.

1 個あるいはそれ以下の入力端変数: "^" の付いた変数名 (例えば, ^X) の出現は入力端を表わす.

0 個あるいはそれ以上の非順序出力端変数: ただの変数名 (例えば, X) の出現は, 併合すべき併合ジョイントの出力端を表す.

0 個あるいはそれ以上の順序出力端変数: "$" と序数が添加された変数名 (例えば, X$1) の出現は, 序数の小さい方が先に連結されるような連結ジョイントの出力端を表す.

序数は正の整数であれば良く, X$1 のように 1 で始める必要も, また X$1, X$2, X$3 のように連続数にする必要もない.

例えば, 表現式 X = ^P1 = ^P2 \ ^S1 \ ^S2 \ ^S3 :: は, 図 19 で示されるチャネルを表す. これは, 入力端変数 ^X 一個, 非順序出力端変数 X 二個, そして 順序出力端変数 X$1, X$2, X$3 の出現として表すことができる.

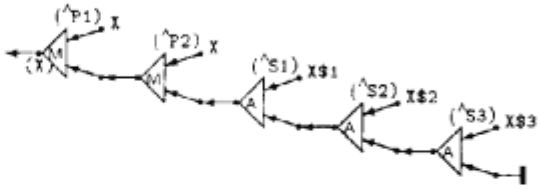

図 19: チャネル変数

4.3 暗黙のストリーム完備化

基本方法では, 不完備なストリームによるデッドロックの発生を防ぎ, 計算の終了を促すように

- ストリーム変数 (入力端変数と出力端変数) は対をなして出現すること
- メソッドの動作列 (< *Actions* > フィールド) には空表現式を指定すること

なる二つの規則が与えられた.

しかし, このように常にストリームを完備化するように, 特に出力端を常に閉鎖するよう心掛けるのは, プログラマにとってかなりの負担となる. さらに, 前節で変数の意味はストリームからチャネルに拡張され, もはや任意個の出力端が出現可能となった.

そこで, 次のように文法を拡張して, 容易に安全なプログラムが書けるようにする.

入力端 / 出力端 (表現式) の放置: メソッドの動作列 (< *Actions* > フィールド) に入力端表現式および出力端表現式を指定してよい. また, 各変数はチャネルを表し, 1 個以下の入力端と任意個の非順序出力端および順序出力端が出現してよい.

自動完備化: 放置された非空表現式は自動的に完備化する. 自動完備化として, 出力端の自動閉鎖, 入力端の自動回収, そしてオブジェクトの自動終了を支援する.

4.3.1 出力端の自動閉鎖

放置された出力端は自動的に閉鎖する. 以下のものにこれを適用する.

- 動作列に放置された出力端表現式の結果
- 出力端が出現しないチャネルの出力端
- スロット更新時の更スロット
- オブジェクト終了時の各出力端スロット
- オブジェクト初期化時の各入力端スロットに対する出力端

4.3.2 入力端の自動回収

放置された入力端は自動的に回収する, すなわち, シンク・オブジェクトを生成し, そこへ接続する. 以下のものにこれを適用する.

- 動作列に放置された入力端表現式の結果
- 入力端が出現しないチャネルの入力端
- オブジェクト終了時のインタフェース・ストリーム
- オブジェクト終了時の各入力端スロット
- オブジェクト初期化時の各出力端スロットに対する入力端

4.3.3 オブジェクトの自動終了

多くのオブジェクトは，そのインタフェース・ストリームの閉鎖を検出した時点で終了するが，とかくこの指定を忘れがちである．そこで，閉鎖検出によりおちに終了するメソッド

```
:: -> ;
```

を上界クラスで定義しており，これは再定義可能である．

4.4 揮発オブジェクト

オブジェクトの各世代は，

1. 観測事象により活性化され，その事象に応じて
2. それと同時に次世代へ降下する．

前者は条件分岐を，後者は継続しを意味する．すなわち，オブジェクトは元来，コンディション・ハンドラとしての機能を備えている．だからと言って，条件判定の度にクラスを一つ定義しているのでは，細かいクラスを数多く定義しなければならないだけでなく，プログラムの文脈がばらばらに分断されてしまう．また，条件判定前の文脈と条件判定後の文脈でストリームの変数環境が独立であると，これを避けるために前者から後者へメッセージを送るのはプログラマには大変負担である．これまで述べたオブジェクトの特徴で，これを解決したいというのが揮発オブジェクト導入の発端である．

揮発クラス: 一つのメソッド内に臨時的に定義されるクラスを **揮発クラス** (volatile class) と呼ぶ．一つのメソッド内に任意個の揮発クラスを定義できる．揮発クラスは，クラス名に関する点を除けば，一般に定義する外部クラスに比べ何の違いもない．その相違点とは，

- 外部クラスはそれを外部から参照するためのクラス名を持ち，これを用いて揮発クラスを含む他のクラスがこれをアクセスあるいは継承することが可能である．
- 揮発クラスはそれを外部から参照するためのクラス名を持たない．

揮発クラスの定義は何段でもネストすることができる．すなわち，ある揮発クラスのメソッド内に別の揮発クラスを定義できる．一般の手続き型言語で条件判定文や繰返し文がネストするのと同様である．

揮発オブジェクト: 揮発クラスのインスタンスを **揮発オブジェクト** (volatile object) と呼ぶ．

生成者オブジェクト: 揮発オブジェクトを生成したオブジェクト，すなわち揮発クラスを定義しているメソッドを実行しているオブジェクトを **生成者オブジェクト** (creator object) と呼ぶ．揮発クラスの定義がネストする場合，内側の揮発オブジェクトにとってそれを定義している外側の揮発オブジェクトはその生成者オブジェクトである．

生成者オブジェクトへのストリームは 擬似変数 `$creator` によって揮発オブジェクトからアクセスできる．

可視領域: スロット名は別々のメソッドに出現して世代を越えて対応するストリームにアクセスできるのに対して，変数名はメソッド内においてのみストリームと一意に対応づけられる．このように名前の有効範囲を名前の可視領域 (name scope) と呼ぶ．

揮発オブジェクトは，生成者オブジェクトとの間の可視領域の関係によって，不変揮発オブジェクトと可変揮発オブジェクトの二種類がある．それぞれの詳細を述べる前に，まず両者に共通なクラス定義とオブジェクトの生成の指定方法について述べる．

4.4.1 揮発オブジェクトの生成

揮発オブジェクト生成表現式は，生成する揮発オブジェクトのインタフェース・ストリームと揮発クラスの定義からなる．揮発オブジェクト生成表現式は空表現式である．

```
< VolatileObjectCreation > ::=
    < ImmutableVolatileObjectCreation > |< MutableVolatileObjectCreation >
< ImmutableVolatileObjectCreation > ::=
    < Interface > "?" < ImmutableVolatileClassDefinition >
< MutableVolatileObjectCreation > ::=
    < Interface > "=>" < MutableVolatileClassDefinition >

< Interface > ::=< InletExpression > |< OutletExpression >

< ImmutableVolatileClassDefinition > ::=
    "(" [ < SuperclassDefinition > ";" ]
        < Method > { ";" < Method > } ")"
< MutableVolatileClassDefinition > ::=
    "(" [ < SuperclassDefinition > ";" ]
        [ < InletSlotDefinition > ";" ]
        [ < OutletSlotDefinition > ";" ]
        < Method > { ";" < Method > } ")"
```

基本: (入力端をインタフェースとして)
基本的に，< *Interface* > フィールドには生成される揮発オブジェクトのインタフェース・ストリームの入力端を指定する．

例えば，

```
^Hunger ? (
    : 'true  -> :eat ;
    : 'false -> :sleep
)
```

と指定すると，生成される揮発オブジェクトは入力端 `^Hunger` から受信するメッセージ `'true` あるいは `'false` により，メッセージ`eat` あるいは `sleep` を自分自身に送信する．

拡張: (出力端には誰何メッセージを送信)
前述の算術マクロを含め多くのマクロは出力端をその評価値としており，その算術結果が何かにより条件分岐したいことがよくある．これを容易に記述できるように，次のような支援をしている．

出力端を指定した場合，暗黙裏にこれに誰何メッセージ `who_are_you(Who)` を送信し，入力端 `^Who` をインタフェースとする．

例えば，奇数か偶数により処理を振り分けたい場合，

```
(X mod 2 == 0) ? ( ... )
```

と指定すると，

```
(X mod 2 == 0):who_are_you(Who)::,
^Who ? ( ... )
```

を意味する．

4.4.2 不変揮発オブジェクト

不変揮発オブジェクト (immutable volatile object) は一代限りのオブジェクトであり，生成者オブジェクトと同一の可視領域を有する (図 20)．すなわち，

変数名の可視領域は同一: 不変揮発クラスに出現する変数名と，その生成者オブジェクトのメソッド内の出現する同一の変数名は，同一のチャネルを意味する．

生成者は自分自身: 不変揮発オブジェクトにとって，生成者オブジェクトは自分自身である．つまり，不変揮発オブジェクトにとって，自分自身およびスロット群は生成者オブジェクトのそれらを意味する．擬似変数 `$creator` と `$self` は同義で使われる

不変揮発オブジェクトの生成: 不変揮発オブジェクト生成表現式の < *Interface* > フィールドに指定される入力端 (`^IVobj`) は，生成される不変揮発オブジェクトの第 0 世代に接続される．

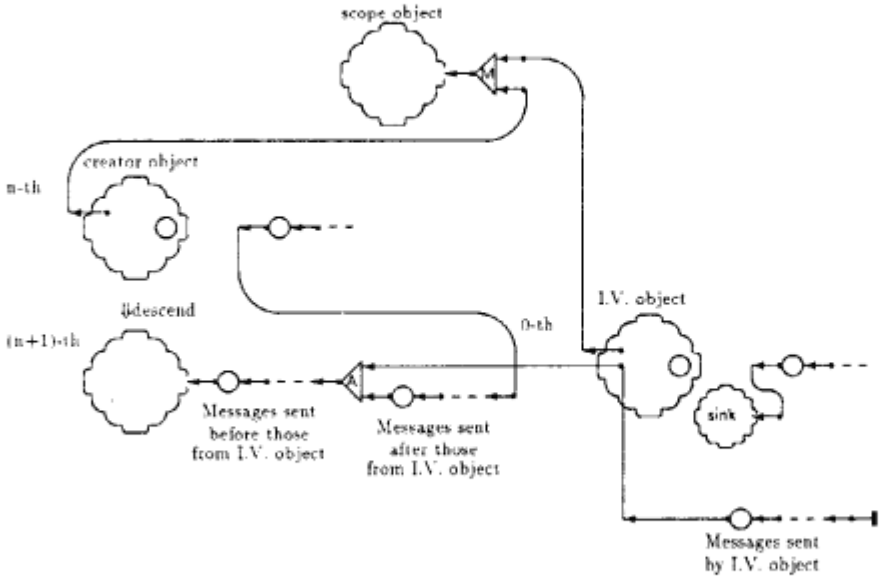

図 20: 不変揮発オブジェクトと生成者オブジェクト

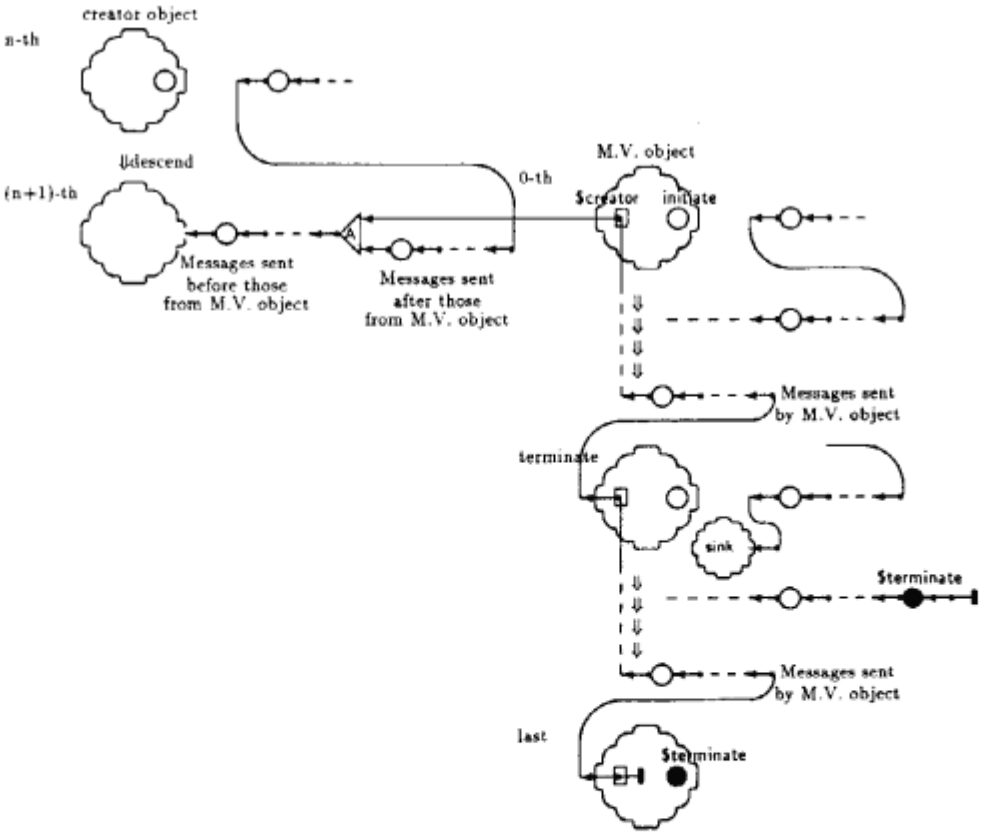

図 21: 可変揮発オブジェクトと生成者オブジェクト

同一の可視領域であるとは，可視領域を共有することに他ならない．不変揮発オブジェクトはその生成時に，生成者オブジェクト自身へのストリームの他に，**可視領域オブジェクト** (scope object) へのストリームが渡される．生成者オブジェクトも同様に，可視領域オブジェクトへのストリームを渡される．両者が共有するチャネル群は，この可視領域オブジェクトによって管理される．

可視領域オブジェクト: 一般のオブジェクトはスロット名でスロット群を管理する．それと全く同様に，**可視領域オブジェクト** (scope object) は，変数名でチャネル群を管理するオブジェクトである．

可視領域の共有: 不変揮発オブジェクトがそのインターフェース・ストリームからどのメッセージを受信し，どのチャネルをアクセスするかは実行時に決まる．

例えば，生成オブジェクトとの間である共通のチャネル変数を参照しており，あるメソッドにはその出力端が3個出現し，別のメソッドにはその出力端が2個出現するとしよう．どちらの場合も関係するチャネル（ストリーム）はすべて完備化されなければならない．

さらに，不変揮発クラスの定義はネスト可能である．これを，可能な限りの場合数だけ別々なコードを静的に用意するのはコード最小化の方針から大きくずれてしまう．

問題をまとめると，物理的なコードは最小化し，かつ，チャネルはすべて完備化しなければならないということである．可視領域オブジェクトの導入をもってこれを解決している．つまり，同一の可視領域を有するオブジェクト間で一つの可視領域オブジェクトを共有することにし，実行時にしか決まらないことは実行時に決めることにした．

不変揮発オブジェクトにとっての自分自身: 文脈中の出現順序に従った，生成者オブジェクトの自己アクセス順序は次のようにして守られる．

- 不変揮発オブジェクトには，生成者オブジェクト自身へのストリームを連結分岐した前方が渡される．

  不変揮発オブジェクトは自分自身へのすべてのメッセージ送信後，これを閉鎖する．

- 後方は生成者オブジェクトが保持し，その不変揮発クラスの定義の後に出現する自分自身へのアクセスに用いる．

不変揮発オブジェクトは，主に条件分岐の目的に使われる．

4.4.3 可変揮発オブジェクト

可変揮発オブジェクト (mutable volatile object) はそれ自身の世代を持つ．生成者オブジェクトとは独立のオブジェクトであり，生成者オブジェクトとは独立の可視領域を有する (図 21)．すなわち，

変数名の可視領域は独立: 可変揮発クラスに出現する変数名と，その生成者オブジェクトのメソッド内の出現する同一の変数名は，別々のチャネルを意味する．

生成者はスロットの一つ: 可変揮発オブジェクトはそれ自身の内部状態としてスロット群を保持することができる．生成者オブジェクトは 擬似変数 `$creator` をスロット名とする一つの出力端スロットにすぎない．一方，擬似変数 `$self` は可変揮発オブジェクトの自分自身を表す．このように，擬似変数 `$creator` と `$self` の意味は異なる．

可変揮発オブジェクトの生成: 可変揮発オブジェクトは，外部オブジェクトと全く同様に，生成された直後，暗黙裏に初期化メッセージが送信される．初期化メッセージに対するメソッドもやはり同様に再定義可能である．

可変揮発オブジェクト生成表現式の ＜*Interface*＞フィールドに指定される入力端 (`^MVobj`) は，この初期化メッセージの後に続くストリームの入力端である．可変揮発オブジェクトもその生成時に，生成者オブジェクト自身へのストリームを渡される．

可変揮発オブジェクトにとっての自分自身: 文脈中の出現順序に従った，生成者オブジェクトの自己アクセス順序は，不変揮発オブジェクトの場合と全く同様に，次のようにして守られる．

- 可変揮発オブジェクトには生成者オブジェクト自身へのストリームを連結分岐した前方が渡され，これを出力端スロット `$creator` に保持する．

  可変揮発オブジェクトの終了時，他の出力端スロットとともに出力端スロット `$creator` は閉鎖される．

- 後方は生成者オブジェクトが保持し，その可変揮発クラスの定義の後に出現する自分自身へのアクセスに用いる．

可変揮発オブジェクトは，主に繰返しの目的に使われる．
以下に，揮発オブジェクトの応用例を示す．

[応用例-2 (素数の生成)]

```
class   prime.                                          % 1
  :primes(^Max, ^Ps) ->                                 % 2
      3 = ^X,                                           % 3
      :generate(X, Max, Ns),                            % 4
      #sift:do(X, ^Ns, Ps:n(2):n(X) ).                  % 5
  :generate(^X, ^Max, ^Ns) ->                           % 6
      ( (X+2 = ^NewX) < Max ) ? (                       % 7
          : 'true  ->                                   % 8
              :generate(NewX, Max, Ns:n(NewX) ) ;       % 9
          : 'false ->         % end %                  %10
      ).                                                %11
end.                                                    %12

class   sift.                                           % 1
  :do(^V, Ns, ^Ps) ->                                   % 2
     S:initialize(V, Ps) = ^Ns,                         % 3
     ^S => (                                            % 4
            out  me, next, to_next, primes ;            % 5
       :initialize(^V, ^Ps) ->                          % 6
          V = !me, 0 = !next, Ps = !primes ;            % 7
       :n(^X) ->                                        % 8
          ( (X mod !me) == 0 ) ? (                      % 9
            : 'true  ->  ;                              %10
            : 'false ->                                 %11
                (!next == 0) ? (                        %12
                   : 'true  ->                          %13
                       X = !next, Ns = !to_next,        %14
                       #sift:do(X, ^Ns, !primes:n(X)); %15
                   : 'false ->                          %16
                     !to_next:n(X)                      %17
            .      )                                    %18
          )                                             %19
     ).                                                 %20
end.                                                    %21

class  test.
    :test ->
        #prime:primes(20, Ps), :nop(^Res).
    :nop(Res) -> .
end.
```

図 22: 素数生成のプログラム

指定される最大値 (max) までの素数列を生成するプログラムである.

1. まず 2 を送信後に 3 から始まる奇数列を生成する. これをふるいの列に通して得られるのが素数列である.
2. ふるいは新しい素数が見つかる度に一つ作られる. まず 3 のふるいが作られる.
3. ふるいは流れてくる数を自分の素数で割り.
   - 割り切れる (自分の倍数である) 場合, 何もしない.
   - 割り切れない場合, 自分が現時点で最大の素数であるかを調べる
     - 自分が最大の素数である場合, 今流れてきた数は素数である. これに対するふるいを作り, これを "次のふるい" とする.
     - そうでなければ, 今流れてきた数を "次のふるい" に渡す.

   ふるいは流れてくる数がなくなるまで, これを繰り返す.

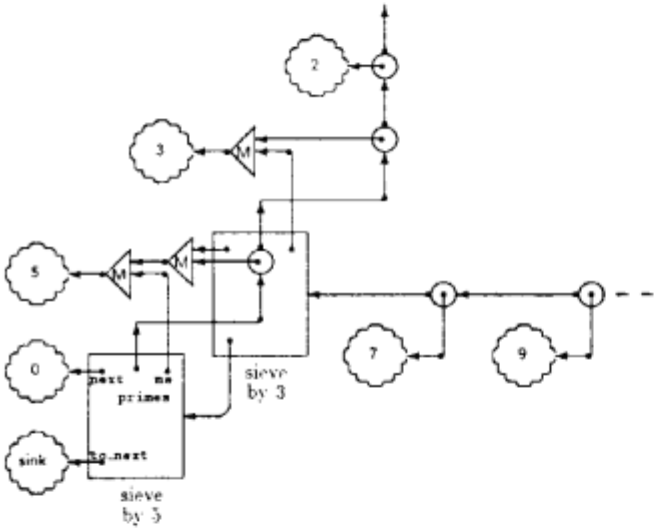

図 23: 素数生成のふるい列

クラス sift の 4 行目から 20 行目に可変揮発オブジェクト (仮に *M1* と呼ぶ) が定義されており, またその中の 9 行目から 19 行目に不変揮発オブジェクトが一つ (仮に *I1* と呼ぶ), さらにその中の 12 行目から 18 行目に別の不変揮発オブジェクト (仮に *I2* と呼ぶ) が定義されている. 可変揮発オブジェクトの外側 (2 行目から 3 行目) に出現する変数 V, Ns, Ps は内側 (4 行目から 20 行目) に出現する変数 V, Ns, Ps とは独立であり, メッセージ initialize を介して内側に伝えられる. また, 7 行目, 9 行目, 14 行目, 15 行目, 17 行目に出現するスロット・アクセスはいずれも, その同一可視領域内で最も外側, すなわち 可変揮発オブジェクト (*M1*) のスロットに関するものである.

5 ストリームとオブジェクトの融合

ストリームとオブジェクトの融合が果たす役割をここにまとめる.

5.1 ストリーム計算と並列計算機システム

ストリーム計算は, より美しい並列計算モデルの構築のためばかりではない. より実用的な並列計算機システムの構築にそれがもたらす恩恵は大きい.

ストリームは入力端と出力端という二つの端点で定義された. 入力端一つと出力端一つとは, 言い換えれば, 読み手一人と書き手一人の単一参照・単一代入を意味する.

単一代入は, 書き手が誰であるか決まれば何時書いても良い, すなわち, どのような順序で実行しようとそのデータ・アクセスに関する意味が一意に定まることを意味する. データと制御を分離でき演算機能の分散を促すので, データフロー言語で広く採用されてきた.

単一参照は, その読み手がメッセージを読んでしまえば, 他に読みたい人はいないので, そのメッセージを捨ててよい, すなわち, 実行時にゴミが回収できるため, 計算機システム全体の連続走行が可能となるのである. 並列計算機構築にとって, ゴミの回収 (GC) は最も重要な課題の一つである. 多重参照において生じるように, 永久的ゴミが堆積した時, 全システムを止めて一括 GC をするのは非実用的である. 計算機の規模が大きくなれば, なおさらである. したがって, この単一参照のもたらす影響は大きい.

5.2 不動点としてのオブジェクト

我々は並列システムを事象の順序付けという極めて微視的な視点から眺めてきた. オブジェクトの一生は世代の連鎖と定義した. オブジェクトの一生を事象の "開いたグラフ" と見ると, 各世代をそのグラフの不動点あるいは循環点 ($x = f(x)$ なる x) とみなすことができる. 一つのグラフを同型をなす部分グラフで畳み込んだものがオブジェクトである.

これを人間になぞらえれば, ごく自然に解釈されよう. 体を構成する細胞は刻一刻再生され, 人は変貌と遂げていく. ある時刻でのその人は, 一秒後のその人とは物理的に違う. 各時刻でその人はある因果で結ばれた物の集合体を表している. 生まれてから死ぬまでその人と認識するのは, 一瞬々のその人を繋ぐ因果である. ストリームによるオブジェクトの表現はこの様子をありのままに映し出している.

同じように微視的な視点から並列システムを定義した計算モデルあるいは言語は他にもいくつかある. 半順序集合モデル [Pratt86], プロセス代数言語群 [Milner80, Milner83, Winskel84, Bergstra84, Winkowski87, Bakker87], 並列論理型言語群, [Shapiro83a, Clark84, Ueda85] 並列関数型言語群 [Broy86], などが含まれる. それぞれの不動点である, 再帰的代数式, 再帰的述語定義, あるいは再帰的関数定義にマクロ表現を導入するなど, やはりオブジェクトによる抽象化がなされつつある [Staples83, Winkowski87, Milner83, Bakker87, Shapiro83b, Broy86].

6 おわりに

最初の問い掛けに対する我々の答えをまとめると, それは極めて簡素なものである. "並列システムは半順序集合であり, 並列プログラミングは事象のグラフを作成することである. グラフを不動点 (循環点) での畳み込みがオブジェクトである."

A'UM は並列計算機システム, 並列プログラミング言語, 並列デバッギング環境を含め並列に関する数多くの興味深い問題を我々に投げ掛けてくる. 今後, これらの解決しながら, 並列とは何かをさらに探っていきたい.

参考文献

[Bakker87] J. W. de Bakker, J. J. Ch. Meyer, and E. R. Orderog: *Infinite Streams and Finite Observations in The Semantics of Uniform Concurrency*, Theoretical Computer Science 49, North-Holland, 1987.

[Bergstra84] J. A. Bergstra and J. W. Klop: *Process Algebra for Synchronous Communication*, Informantion and Control 60, 1984.

[Birkhoff79] Garrett Birkhoff: *Lattice Theory*, American Mathematical Society, Colloquium Publications, Vol. 25, Third edition, 1979

[Broy86] M. Broy: *A Theory for Nondeterminism, Parallelism, Communication, and Concurrency*. Theoretical Computer Science 45, North Holland 1986.

[Clark84] K. Clark and S. Gregory: *PARLOG: Parallel Programming in Logic*. Concurrent Prolog Vol.1, MIT Press, 1987.

[Milner80] R. Milner: *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag 1980.

[Milner83] R. Milner: *Calculi for Synchrony and Asynchrony*. Theoretical Computer Science 25, North-Holland, 1983.

[Pratt86] V. Pratt: *Modeling Concurrency with Partial Orders*. International Journal of Parallel Programming 15(1), 1986.

[Shapiro83a] E. Shapiro: *A Subset of Concurrent Prolog and Its Interpreter*. Concurrent Prolog Vol.1, MIT Press, 1987.

[Shapiro83b] E. Shapiro and A. Takeuchi: *Object Oriented Programming in Concurrent Prolog*. Concurrent Prolog Vol.2, MIT Press, 1987.

[Staples83] J. Staples and V.L. Nguyen: *Computing the Behavior of Asynchronous Processes*. Theoretical Computer Science 26, North-Holland, 1983

[Ueda85] K. Ueda: *Guarded Horn Clauses*. Concurrent Prolog Vol.1, MIT Press, 1987.

[Yoshida88b] Kaoru Yoshida and Takashi Chikayama: *A'UM – A Stream-Based Concurrent Object-Oriented Language –*, Proc. of FGCS'88, Nov. 1988

[Yoshida89b] 吉田かおる, 近山隆: "*A'UM* (V.1) 解説書 (第一版)", ICOT Technical Memorandum TM-690, 1989年3月

[Winskel84] G. Winskel: *Synchronization Trees*. Theoretical Computer Science 34, pp.33-82, North-Holland, 1984.

[Winkowski87] J. Winkowski: *An Algebra of Processes*, Journal of Computer and System Sciences 35, pp.206-228, 1987.