並列論理型言語による探索問題のプログラミング†

——層状ストリーム法の拡張——

松本裕治†† 奥村晃†††

Committed Choice 言語と呼ばれる並列論理型言語による探索問題のプログラミング法について述べる．探索問題が持っているAND並列性やOR並列性を自然に抽出するために，層状ストリーム(layered stream)という入れ子構造を持つストリームを基本的なデータ構造として用いるプログラミング手法を提案する．問題の再帰的表現による記述から出発し，問題のタイプに応じた変形によって，並列度の高いプログラムが機械的に得られることを示す．

1. はじめに

Committed Choice 言語（以後 CCL）と呼ばれる並列論理型言語は，GHC[1] や PARLOG[2] に代表されるように，制限された OR 並列性を持つことにより多重環境を不要にした AND 並列論理型言語である．ところが，一般に，探索問題では多くの異なる可能性を探る必要があり，これを並列に解くためには同時に相容れない環境を取り扱う OR 並列型の探索が必要である．したがって，CCL によって探索問題を解くプログラムを直接的に記述するのは容易ではない．

このような問題を CCL を用いて解く方法として，上田[3]や玉木[4]は，Prolog で記述されたプログラムを CCL のプログラムに変換することにより探索時の OR 並列性を CCL の AND 並列によって実現できることを示した．また，著者らは以前に層状ストリーム (layered stream) というデータ構造を利用した並列プログラミング方式を提案し[5),6]，再帰的な構造を持つ探索問題に対して AND 並列と OR 並列を効果的に引き出すことが可能であることを示した．本論文は，以前の方法の一般化を行う．具体的には，問題の再帰的な記述を行い，そこから出発して，層状ストリームを用いた CCL プログラムを得るための方法を提案する．

次章で層状ストリームについて述べ，第3章で探索問題の考え方について説明する．我々がここで考えるのは，問題の解の記述がその部分問題の解により構成されるというように，再帰的に記述することができる探索問題である．多くの探索問題は再帰的な記述が可能である．つまり，より大きな問題の解は，その部分問題の解および問題を構成する他の要素を用いたある演算または組合せによって再帰的に記述することができる．

第4章では，層状ストリーム法の一般形について述べる．探索問題の部分解または解を構成するその他の要素が並列に動くプロセスとして表現され，それらが通信し合うことによって問題が解かれる．通信に用いられるデータはプロセスを結ぶストリーム（ただし，ここでは層状ストリーム）を介して送受される．層状ストリームはストリームを入れ子構造的に含むストリームである．この構造がストリームの動的な結合と部分解の共有に役立つ．

問題が持つ OR 並列性は，相異なる部分解に関する情報をストリームに流すことによって表現される．また，問題が持つ AND 並列性は，1つの解の異なる部分を別のプロセスに処理させ，それらの解をストリームを通じて他のプロセスへ伝えることによって達成される．特に，動的計画法によって解かれる問題のように，部分解を蓄積して他の部分解を計算する時にそれらを再利用することによって効率を達成する解法は，部分解をプロセスとして表現することにより層状ストリーム法で自然に実現することが可能である．

2. 層状ストリームとその特徴

本論文では，CCL についての基本的な知識を仮定し，言語としては GHC を用いることにする．GHC では，節の本体部のゴールの実行順序に逐次性がな

† Parallel Programming of Searching Problems in Committed-Choice Parallel Logic Programming Languages —An Extension of Layered Stream Programming Method— by YUJI MATSUMOTO (Department of Electrical Engineering II, Faculty of Engineering, Kyoto University) and AKIRA OKUMURA (Knowledge Information Processing Systems Department, Systems Laboratories, Oki Electric Industry Co., Ltd.).

†† 京都大学工学部電気工学第二学科
††† 沖電気工業(株)総合システム研究所知識情報処理研究部

く，並列に実行することができる．ただし，GHC では，述語の呼び出し時に，呼び出し側の述語に含まれる変数に値の割当てが起こる場合にはその節の実行を一時的に中断する．そのような節は，他の述語の実行によって呼び出し側の変数に十分な値が得られた後，実行を再開する．

CCL では，述語が並列に動くプロセスとして扱われ，通信チャネルとしてストリーム（構造上は，リスト表現）が用いられる．我々が提案する層状ストリームは，構文的には単に自分自身と相似な構造を要素としてもつストリームであるが，後に述べる問題の再帰的な記述からの並列プログラムの導出からわかるように次のような特徴を持つ CCL の並列プログラムの実現に寄与する．

1. 探索問題における部分問題の解の共有
2. 問題の AND 並列性および OR 並列性の抽出
3. ストリーム（通信チャネル）自身をストリームの要素として通信することによる動的な通信チャネルの結合

層状ストリームは，一般に次のような構造をしている．

$[a_1 * LS_1, a_2 * LS_2, \cdots]$

ここに，'a_i' は項，'*' は結合記号，'LS_i' は内部に含まれる層状ストリームである．層状ストリームは，一般的には，問題の部分解の集合を表すと考えられる．項は，問題の部分解の一部を成すデータであり，問題固有のデータである．結合記号は，そのようなデータと，より規模の小さい部分解の集合を表す層状ストリームを結合するための記号であり，ここでは記号 '*' を用いることにする．上の例の先頭の要素，

$a_1 * LS_1 = a_1 * [b_1 * LS_1', b_2 * LS_2', \cdots]$

は，

$[a_1 * b_1 * LS_1', a_1 * b_2 * LS_2', \cdots]$

という集合を表している．これを再帰的に適用すると，元の層状ストリームは，それを結合記号 '*' についてすべて展開して得られるデータの集合を表現していると考えることができる．

このような部分解の再帰的な表現によって，後に述べる探索問題の再帰的な記述との対応が明確になる．しかも，部分解を構成する項の部分の生成および処理と，内部の層状ストリームの生成および処理を独立に，すなわち，並列に行うことが可能になる．層状ストリームの構造の深さ方向の並列処理が AND 並列に，幅方向の並列処理が OR 並列に対応する．また，層状ストリームは部分解の共通な部分を共有したデータ構造であり，個別の部分解に対する処理を重複して行うことを避けることができるので，逐次的に実行しても効率のよいアルゴリズムを得ることが可能になる．

また，ストリームのデータ内部に含まれているストリームが新しく生まれたプロセスに渡されることによって通信チャネルの動的な結合が行うことができ，動的なプロセス間通信が実現できる．

3. 探索問題とその再帰的記述

3.1 問題記述の基本形

ここでは，探索問題を，ある指定された構造を持つデータのうち特定の条件を満足するものを選び出す問題であると考える．そして，そのような問題の再帰的な記述を考えることにする．ここでは，次のような2通りの再帰的な記述を対象にする．

1. 部分問題$_k$＝部分問題$_{k-1}$ ⊕ 解要素
   - **整合性**：部分問題$_{k-1}$ と解要素の間に成立する条件（AND 条件）
   - **曖昧性**：部分問題$_{k-1}$ および解要素の候補となり得るための条件（OR 条件）
   - **目標関数**：部分問題$_k$ の多様な解の中から最適なものを選ぶための条件
2. 部分問題$_k$＝部分問題$_{i_1}$ ⊕ 部分問題$_{i_2}$ ⊕ … ⊕ 部分問題$_{i_n}$ $\left(n \geq 2,\ k = \sum_{j=1}^{n} i_j\right)$
   - **整合性**：右辺の部分問題の間で成立しなければならない条件（AND 条件）
   - **曖昧性**：右辺の部分問題の候補となり得るための条件（OR 条件）
   - **目標関数**：部分問題$_k$ の多様な解の中から最適なものを選ぶための条件

この記述は非形式的な記述であり，プログラマがどのような指針で問題の構成を明らかにすればよいかということを示しているだけである．ここに，'部分問題$_k$' とは規模が k の部分問題ということであり，問題によってその意味は異なる．上の再帰的記述の2つ目の記述例においては，i_j の合計の規模が k となる．また，⊕は，部分問題の解同士や解要素をもとにより大きな部分解を構成するために必要な演算を表しており，これも問題に応じて異なる．

問題の再帰的な記述と共に様々な条件が記述されている．ここでは条件として，整合性条件，曖昧性条件，目標関数の三種類を考えている．整合性とは，再帰的記述の右辺に現れる部分問題の解や解要素の間で

なりたたなければならない条件である．曖昧性は，問題が多重の解をもっているときの曖昧性の許容範囲を定義するための条件であり，どのような部分解や解要素が右辺に現れ得るかを規定する．目標関数は，左辺の部分問題の解となり得るものが複数個存在する場合，どのような基準で適当な解を選択すべきかを定義する．

これらの2つのタイプの記述のうち，前者は部分問題の解に何らかの要素を追加していくことによってより大きな部分解が得られる問題であり，次に示すクイーン問題やグラフの最短パスの問題などがこの型に属す．また，後者は相似な部分解をいくつか結合することによってより大きな部分解を得る問題であり，次に述べる行列の積のコストの最適化問題や文脈自由文法の構文解析の問題がこの型に相当する．

いくつかの探索問題の再帰的記述例を示す．

(例1) N クイーン問題

$N\times N$ のチェス盤上に N 個のクイーンをお互いに相手を取ることができないように配置する問題．

$\mathrm{queens}_k = \mathrm{queens}_{k-1} \oplus \mathrm{queen}$

queens_k：最初の k 個のクイーンを盤上の最初の k 列において得られる部分解

queen：1個のクイーンの盤上の1つの列内の位置

整合性：解要素の queen は，queen_{k-1} に含まれるどの queen とも同一行，同一対角線上にあってはならない

曖昧性：すべての可能な queen_{k-1} の解とすべての解要素 queen の可能な位置を考慮の対象とする

目標関数：なし

(例2) グラフの最短経路問題

グラフ内のある決められた始節点と終節点を結ぶ最小コストの経路を発見する問題．

$f_k(v) = \min_u (f_{k-1}(u) + d(u,v) \mid (u,v) \in E)$

$f_k(v)$：始節点から k 本以内の枝を通って節点 v へ至る最短経路長

$d(u,v)$：u と v を結ぶ枝 (u,v) の長さ

E：グラフの枝の集合

整合性：$f_{k-1}(u)$ と $d(u,v)$ において u が同一節点でなければならない

曖昧性：すべての可能な節点 u を対象とする

目標関数：$f_k(v)$ の最小値

(例3) N 個の行列の積のコストの最小値問題

与えられた N 個の行列の積を最小回数の乗算によって求めるための計算順序を決定する問題．

$matrix_{ij} = \min_k (matrix_{ik} + matrix_{kj})$

$matrix_{ij}$：第 i 行列から第 j 行列までのすべての行列の積を計算するためのコスト

整合性：右辺の行列が隣接していなければならない

曖昧性：すべての可能な $k(i \le k \le j)$ を対象とする

目標関数：$matrix_{ij}$ の最小値

(例4) 文脈自由文法の構文解析

与えられた文脈自由文法に基づいて，N 個の単語よりなる入力が文と認められるかどうかを判定する問題．

$NT_{ij} = \underset{k_1,k_2,\cdots,k_n}{\mathrm{exist}} (NT_{ik_1} \oplus NT_{k_1k_2} \oplus \cdots \oplus NT_{k_nj})$

NT_{ij}：入力の第 i 単語から第 j 単語によって構成される非終端記号

整合性：$NT_{ij} \leftarrow NT_{ik_1}NT_{k_1k_2}\cdots NT_{k_nj}$ という文法規則が存在しなければならない

曖昧性：すべての可能な文法規則を対象とする

目標関数：NT_{ij} が少なくとも1つ存在する

3.2 条件を実現するためのメカニズム

次の章で示すように，再帰的に記述された問題の解や解の集合を表現するために前章で導入した層状ストリームを利用する．本章では，問題の再帰的記述において定義された種々の条件をプログラム上で実現するための基本的なメカニズムを説明する．

層状ストリームプログラミングでは，部分問題の解や解を構成する要素が層状ストリームを用いて表現され，規模の小さな部分問題の解や解の構成要素からより大きな問題の解を構成するようにプログラムされる．その中で，曖昧性条件は，次章で説明するように，利用可能な部分解や解要素に対応する情報を層状ストリームとして生成する操作として実現される．

一方，整合性条件と目標関数は，層状ストリームの中から問題の条件に合致しない部分を取り除くフィルタプロセスとして実現される．整合性条件に対応するフィルタは**整合フィルタ** (consistency filter) と呼ばれる．また，目標関数を実現するフィルタを**優先フィルタ** (preference filter) と呼ぶ．

目標関数の特殊な例として，構文解析の例のように各部分解の存在のみを要求する場合がある．これは，

ストリームが同一の解を通さないようにすることによって実現可能であり，プロセス間に重複解を取り除くためのフィルタをおけばよい．そのようなフィルタは**重複フィルタ** (dupulication filter) と呼ばれる．したがって，重複フィルタは，優先フィルタの特別な場合である．

これらについては次章で例を用いて説明する．

4. 層状ストリームプログラミング

我々が提案する層状ストリームプログラミングでは，問題の再帰的な記述から出発し，層状ストリームを通信チャネルとして用いる並列プロセスの集まりとしてプログラムを作成し問題を解く．基本的には，再帰的な記述の右辺に現れる部分問題や解要素を並列プロセスまたは層状ストリームによって表現することによってプログラミングが行われる．ただし，問題の記述の中のどの部分を層状ストリームを用いたデータとして表現し，どの部分を並列プロセスとして表現するかということは一意的に決まるのではなく，問題に応じて最もふさわしい選択を行う必要がある．

例として，N クイーン問題を取り上げる．前章で示した問題の記述の右辺には部分解 queens_{k-1} と解要素 queen が存在する．問題を解くためにはこれらが相互作用を行わなければならないが，これらのうちいずれを並列プロセスとし，いずれをデータとして表現するかの選択肢がある．次に示すようにいずれに対するプログラミングも可能である．

4.1 queens_{k-1} を層状ストリームとし，queen をプロセスとする場合

4クイーンについてのプログラムを次に示す．

```
queens(Ans) :-true|
    queen(begin, Q1), queen(Q1, Q2),
    queen(Q2, Q3), queen(Q3, Ans).
queen(In, Out) :-true|
    filter(In, 1, 1, Out1),
    filter(In, 2, 1, Out2),
    filter(In, 3, 1, Out3),
    filter(In, 4, 1, Out4),
    Out=[1*Out1, 2*Out2, 3*Out3, 4*Out4].
filter(begin, _, _, Out) :-true|Out=begin.
filter([], _, _, Out) :-true|Out=[].
filter([J*_|Ins], I, D, Out) :-I=J|
    filter(Ins, I, D, Out).
filter([J*_|Ins], I, D, Out) :-D=:=I-J|
    filter(Ins, I, D, Out).
filter([J*_|Ins], I, D, Out) :-D=:=J-I|
    filter(Ins, I, D, Out).
filter([J*In1|Ins], I, D, Out) :-
    J\=I, D=\=I-J, D=\=J-I|
    D1:=D+1, Out=[J*Out1|Outs],
    filter(In1, I, D1, Out1),
    filter(Ins, I, D, Outs).
```

'queens' は全体の解を求めるプロセスであり，'queen' が個々のクイーンに相当する．'queens' に呼び出された4つの 'queen' プロセスは直前の部分解に相当する層状ストリームを受け取り，自分の位置の情報を追加した解を新たな層状ストリームとして次のクイーンへ渡す．'queen' は新たな層状ストリームを作ると同時にそれぞれの位置に関するフィルタを呼び出している．このフィルタは整合フィルタである．これは，全解を求めるクイーン問題であり，優先フィルタは用いられていない．'filter' の第一引数と第四引数は，入力および出力ストリーム，第二引数は当該クイーンの置かれた場所，第三引数は当該クイーンとこの 'filter' がチェックしているクイーンとの距離（始めは隣のクイーンとのチェックを行うので初期値は1である）を表している．

4.2 queens_{k-1} をプロセスとし，queen を層状ストリームとする場合

同様に4クイーンのプログラムを示す．

```
queens(Ans) :-true|
    q(LS), qs(LS, [], Out-[]).
q(LS) :-true|
    qX(begin, Q1), qX(Q1, Q2),
    qX(Q2, Q3), qX(Q3, LS).
qX(In, Out) :-true|
    Out=[1*In, 2*In, 3*In, 4*In].
qs(begin, Board, From-To) :-true|
    From=[Board|To].
qs([I*LSi|LSs], Board, From-To) :-true|
    checkQs(Board, I, 1, LSi, Board, From-M),
    qs(LSs, Board, M-To).
qs([], _, From-To) :-true|From=To.

checkQs([Q|Qs], Q, D, In, Board, Out) :-
    true|Out=End-End.
checkQs([Q|Qs], I, D, In, Board, Out) :-
    Q=:=I+D|Out=End-End.
```

```
checkQs([Q|Qs], I, D, In, Board, Out) :-
    Q= :=I-D|Out=End-End.
checkQs([Q|Qs], I, D, In, Board, Out) :-
    Q=\=I+D, Q=\=I-D, Q\=I|D1 :-D+1,
    checkQs(Qs, I, D1, In, Board, Out).
checkQs([], I, D, In, Board, Out) :-
    true|qs(In, [I|Board], Out).
```

このプログラムでは 'qX' が1つのクイーンに相当し，これによって4クイーン問題の解になり得る組合せが層状ストリームの形で構成される．'qs' が部分解に対応するプロセスであり，これが層状ストリームを消費しながら新しい部分解へと分岐していく．

後者のプログラムは，クイーンのあらゆる可能な位置をしらみつぶしに調べるのに対し，前者では，層状ストリームの構成と共に配置されたフィルタが同時に枝狩りを開始し，しかも密にデータの共有を行っている．実験による比較では，4クイーンの場合の総リダクション数（頭部単一化の数）が前者と後者とでは，160対1,615，6クイーンでは，1,382対236,185であった．クイーンの数が増すとこの差は急激に拡がる．これは，層状ストリームを効果的に使った場合の解の共有の効果を顕著に示す例である．

4.3 文脈自由文法の構文解析のプログラミング

文脈自由文法の構文解析の問題を用いて，もう1つのタイプの再帰的記述による問題のプログラミングを説明する．この問題の場合は，記述の右辺も左辺も部分問題しか含まないので，部分問題の解（すなわち，非終端記号）を並列プロセスと考える．右辺の部分問題同士は情報のやりとりを行う必要があるので，各プロセスは隣のプロセスへ自分の存在を示すデータ（非終端記号名）を送る．相互にデータを送り合う必要はないので，必ず左のプロセスが右のプロセスへデータを送ると仮定する．具体的には，各プロセスは左のプロセスから受け取ったデータ（層状ストリーム）の先頭に自分の持つ情報を付け加えた新しい層状ストリームを右のプロセスへ送れば良い．このようにして，構文解析の並列プログラムを実現することができる．初期プロセスは，解析しようとしている入力を構成する品詞の列に対応するプロセス列である．

右辺の部分解をどのようにして結合するかということにいくつかの選択肢がある．構文解析システムPAX[7]は，層状ストリームに基づくシステムであるが*このシステムでは，例えば，a←bcd という文法規則に対して，下の(1)に対応するプログラムを生成する．cについて見ると，cは左隣にbが存在することを通信によって知ると，非終端記号bとcが連続して現れたという情報をデータ b_c で表現し，それを右隣へ送っている．これがプロセスdによって受け取られると，この文法規則の右辺が完成したことになり，左辺に対応するaというプロセスが呼ばれる．

一方，下の(2)に示すように別の考え方で同じ文法規則を扱うことも可能である．cは自分の名前を右へ送っているのであるから，それを受け取ったプロセスdが，cとdが連続して現れたことを示すプロセス c_d を生成してもよい．このような考え方が佐藤[8]と山崎[9]によって提案されている．つまり，前者ではプロセスの並びをデータで表現しているのに対し，後者ではそれをもプロセスで表現しているのである．(2)では，c_d というプロセスはその右隣にプロセスbが存在するかどうかを観察し，bを示すデータを受け取ると文法規則の右辺が完成したとしてプロセスaを生成している．

(1)
```
b(In, Out) :-true|Out=[b*In].
c([b*In|R], Out) :-true|
    Out=[b_c*In|Out1],
    c(R, Out1).
d([b_c*In|R], Out) :-true|
    a(In, Out1), d(R, Out2),
    merge(Out1, Out2, Out).
```

(2)
```
b(In, Out) :-true|Out=[b*In].
c(In, Out) :-true|Out=[c*In].
d([c*In|R], Out) :-true|
    c_d(In, Out1), d(R, Out2),
    merge(Out1, Out2, Out).
c_d([b*In|R], Out) :-true|
    a(In, Out1), c_d(R, Out2),
    merge(Out1, Out2, Out).
```

プログラムの実現にもう1つの選択肢がある．例えば，(1)のdプロセスは，b_c というデータを先頭に持った層状ストリームを受け取らない限り絶対にプロセスaを呼び出さない．(2)に関しても同様である．整合性条件はこれらのプログラムではこのように暗に実現されている．一方，これらのプログラムにおけるプロセスの生成を，N クイーンのプログラムでのデータの生成のように積極的に行って，フィルタによって整合性条件を実現する方法も考えられる[10]．ただし，

* 構文解析におけるより強い整合性条件として下降型の予測を用いることが可能である．詳細については，文献7) の top-down filtering の記述を参照．

この方法も無闇にプロセスを発生させることは，たとえ並列度を上げることはできても無駄な処理の増加を招く場合もある．(3)および(4)に2つのタイプの例を示す．対象とする文法規則は上と同様，a← b c d である．

```
(3)  d(In, Out) :-true|
       Out=[d*In|Out1],
       filter([c, b], In, In1),
       a(In1, Out1).
(4)  d([c*In|R], Out) :-true|
       filter(b, In, In1),
       a(In1, Out1), d(R, Out2),
       merge(Out1, Out2, Out).
```

(3)では，文法規則内の右辺の右端に現れるすべての非終端記号をこのように書き直すことによって並列構文解析が実現できる．filter の第一引数が直前に現れなければならない非終端記号列を表している．filter は入力ストリーム In の中からcおよびbが連続して存在することを示すデータだけを取り出して In1 へ出力する．このプログラムでは，c，bいずれの非終端記号の存在の確認も待たずにプロセスaを呼び出しているため，並列度は極めて高いが，無駄な処理も多い．

(4)は，プロセスの生成について(3)より慎重なプログラムであり，filter の第一引数がただ1つで済む場合のみプロセスの生成を積極的に行う．このプログラムでは，プロセスdは左隣にcが存在する場合にのみプロセス a を生成し，同時に b の存在を確認する filter を起動する．ただし，その結果を待たずにプロセスaを呼び出している．

解析すべき入力記号列が $a_1a_2\cdots a_n$ であるとき，以上のプログラムでは，次のような初期ゴールによって構文解析の実行が始まる．

a1(begin, S1), dfilter(S1, SF1), a2(SF1, S2), …,
dfilter(Sn−1, SFn−1), an(SFn−1, Sn).

ここで，dfilter は，重複した解を除去するための重複フィルタであり，第一引数に現れた値を記録しておき，それ以後まったく等価な値が発見されたときにそれを第二引数へ流さないようにする．これによりまったく同一の解析を行わずに済む．

4.4 行列積のプログラムと優先フィルタ

最後に，行列の積の問題を例として優先フィルタについて説明する．次に示すプログラムは，3つの行列(R1 行 C1 列，R2 行 C2 列，R3 行 C3 列) の積のコストの最小値を求める層状ストリームプログラムの例である．

```
go([R1, C1, R2, C2, R3, C3], Out) :-true|
  matrix(begin, 1, R1, C1, 1, 0, S1),
  pfilter(S1, LS1),
  matrix(LS1, 2, R2, C2, 2, 0, S2),
  pfilter(S2, LS2),
  matrix(LS2, 3, R3, C3, 3, 0, S3),
  pfilter(S3, Out).
matrix(In, N, R, C, Tr, Cost, Out) :-true|
  Out=[matrix(N, R, Tr, Cost)*In|Outs],
  matrix1(In, N, R, C, Tr, Cost, Outs).
matrix1([matrix(N, RN, TrN, CostN)*InN|Ins],
        I, R, C, Tr, Cost, Out) :-true|
  Cost1 :=RN*R*C+CostN+Cost,
  matrix(InN, N, RN, C, (TrN−Tr), Cost1, OutN),
  matrix1(Ins, I, R, C, Tr, Cost, Out1),
  merge(OutN, Out1, Out).
matrix1([], _, _, _, _, _, Out) :-true|Out=[].
matrix1(begin, _, _, _, _, _, Out) :-true|Out=[].
```

基本的な考え方は，文脈自由文法の構文解析と同じである．行列の積のコスト計算は，常に連続する2つの行列によって決定できるので，プログラムの中心部分 (matrix1 の定義) は，構文解析に比べて遙かに簡単である．pfilter が優先フィルタである．pfilter のプログラムは省略するが，pfilter は自分の場所を右端とするそれぞれの行列の組合せ (pfilter の場所を第 i 番目の行列の後とすると，第1行列から第 i 行列，第2行列から第 i 行列，等々) について，今までに得られた最小値を保持し，新しく計算したコストがそれを下回らない限り出力として流さないようにする．

4.5 優先フィルタとプロセスの厳密，非厳密モード

優先フィルタは問題において与えられた目標関数を実現するものであるが，目標関数の中には「最小値」のように，すべての部分解を知らなければ求められないものがある．上の行列積の pfilter は，プログラムの途中の段階で最もコストが小さいと考えられるものを出力として流すことを考えており，必ずしも最小コストの解のみが得られる保証はない．

これを修正するには，優先フィルタが入力ストリームのすべてのデータを読み終り，真の目標値を計算するまでデータを出力しないようにすればよい．このような優先フィルタを厳密モード (strict mode) と呼び，途中における最適値を積極的に流す場合を非厳密

モード (non-strict mode) と呼ぶ.

プロセスのデータの出力方法についても同様のモードを考えることができる. N クイーン問題の queen, 構文解析問題の非終端記号, 行列積問題の matrix が, 入力ストリームの内容にかかわらずデータを出力するのが非厳密モード, 入力ストリームに何らかのデータが送られて来るのを確認してから出力を行うのが厳密モードである. 構文解析の例では, 非厳密モードの中にもいく通りかの実現が可能なことを示している. そこで示したプログラムの(3)は, 文法規則の右辺の要素が1つ発見されただけで, 左辺のプロセスを生成しているが, (4)では, 右辺の要素が2つ確認されて始めて左辺の要素が生成されている.

優先フィルタにおいてもプロセスにおいても, 厳密モードの方が無駄な処理が少ないが, ストリームへのデータの出力のタイミングが非厳密モードに比べて遅いため, 並列度は低下する. 優先フィルタやプロセスをどちらのモードで実現するのが最適であるかは, 問題の性質に依存する.

5. 実　　験

本章では, 前章までに示したいくつかのプログラムを用いて, 層状ストリームプログラムの並列環境での評価を行う. 並列プログラムの評価を判定するのは容易ではないが, ここでは GHC のインタプリタを用いて実験を行った. この GHC のインタプリタは Prolog 上で動いており, プロセサ数が無限で, 任意のプロセス間における通信が単位時間で行えるという理想的な並列処理環境を仮定している. したがって, 以下の表に示す数値は, プログラムの理想的な性能を評価したものである. 表の中で, reduction 数とは, GHC 節の頭部を含むガード部の評価の成功の総数を示す. GHC のゴールの実行はすべて同時に並列に行ってもよいので, 現時点で存在するすべての reduction 可能なゴールについてその reduction を行う操作を 1 cycle と考える. cycle 数とは, プログラムの実行が終了するまでにこの操作が何回行われたかを示す. ゴールは常に reduction 可能というわけではなく, 必要なデータが利用可能になるまで実行を中断されることがある. これをゴールの suspension と呼ぶ. suspension 数とは, プログラムの実行過程においてsuspension が起こったゴールののべ総数である.

表1 は 4.1 節で示したクイーン問題の実行結果である. クイーン問題はクイーンの数に関して指数オーダの計算時間が必要である. reduction 数は総計算量を示し, 指数関数のオーダで増加している. 一方, cycle 数は, クイーンの数にほぼ比例しており, 理想的な環境では極めて高い並列度が得られていることがわかる.

表1 Queen 問題の実行結果
Table 1 Execution results of Queen problem.

	Reduction 数	Suspension 数	Cycle 数
4-queen	178	6	14
6-queen	1834	124	25
8-queen	26964	2470	38
10-queen	506690	56767	53

表2 構文解析の実行結果 (重複フィルタなし)
Table 2 Execution results of parsing (without dupulication filters).

入力文長	Reduction 数	Suspension 数	Cycle 数
7	95	0	27
10	196	0	38
13	429	0	57
16	1049	0	108
19	2867	1	253
22	8537	3	690
25	26945	6	2081
28	88396	10	6662
31	297813	15	22111

表3 構文解析の実行結果 (重複フィルタあり)
Table 3 Execution results of parsing (with duplication filters).

入力文長	Reduction 数	Suspension 数	Cycle 数
7	159	32	49
10	286	64	71
13	467	113	97
16	714	184	127
19	1039	281	161
22	1454	410	199
25	1971	574	241
28	2602	779	287
31	3359	1029	337

表2 および**表3** は文脈自由文法の構文解析の実行結果である. 文法規則数が 10 個程度の小さな英語の文法を対象にしているが, 前置詞句の係受けの複雑な極めて曖昧性の高い例を選んだ. 表3は重複フィルタを用いて等価な部分解の伝達を抑えた場合である. 重複フィルタによって reduction 数, cycle 数ともに大幅に減少しているのがわかる. ただし, フィルタの出力待ちのため, suspesion を起こすプロセスの数が増加している.

6. おわりに

並列論理型言語による探索問題のプログラミング法について述べた．探索問題の再帰的な記述に基づき，層状ストリームというデータ構造を使うことにより，並列度の高いプログラムを得ることができる．問題の記述のどの部分を並列プロセスとして考えるかによって結果としてのプログラムにいくつものヴァリエーションが存在することを示した．また，問題を記述するためのいくつかの条件分けを行い，それぞれをフィルタというメカニズムで実現する方法を示した．さらに，ストリームの要素を受け取るフィルタやプロセスに厳密モードと非厳密モードが考えられることを示した．

現在扱っている問題は，再帰的な記述が可能で，かつ，部分問題の解を用いてより大きな部分解を構成する時に部分解の重複利用が有効なものである．また，部分解からより大きな解への組合せの方法が比較的簡単な問題である．今後，対象とする問題の領域を拡げていきたい．

また，同じ問題の記述によっても，問題の記述のどの部分を並列プロセスと考え，どの部分を層状ストリームと考えるかにはいくつかの見方があることがわかった．問題をどのように見究めることよって最も効果的なプログラムを得ることができるかという議論も今後の課題である．

優先フィルタについては，現在のところ，局所的な情報によってのみ判定できる枠組しか用意していない．ある程度大局的な情報を利用した枝刈りの機能を層状ストリームプログラミムの基本機能として追加することもこれから考えたい．

謝辞 ご討論いただいた京都大学長尾真教授はじめ長尾研究室の諸氏に感謝いたします．後半の例として用いた構文解析のプログラミングの色々な可能性についての議論に対して，ICOT の瀧和男氏，三菱の佐藤裕幸氏，富士通の山崎重一郎氏に感謝いたします．日頃貴重な意見を賜わる ICOT 並行並列プログラムワーキンググループの諸氏に感謝いたします．最後に有益なご意見をいただいた査読者の方々に深謝いたします．

参考文献

1) Ueda, K.: Guarded Horn Clauses, *Logic Programming '85*, Wada, E. (ed.), *Lecture Notes in Computer Science 221*, pp. 168-179, Springer-Verlag (1986).
2) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 1, pp. 1-49 (1986).
3) Ueda, K.: Making Exhaustive Search Programs Deterministic, *Proc. 3rd ICLP, Lecture Notes in Computer Science 225*, pp. 270-282 (1986).
4) Tamaki, H.: Stream-based Compilation of Ground I/O Prolog into Committed-Choice Languages, *Proc. 4th ICLP*, The MIT Press, pp. 376-393 (1987).
5) Okumura, A. and Matsumoto, Y.: Parallel Programming with Layered Streams, *Proc. 1987 International Symposium on Logic Programming*, San Francisco, pp. 224-232 (Sep. 1987).
6) 松本裕治，奥村 晃：Dynamic Layered Stream Programming，日本ソフトウェア科学会第5回大会論文集，pp. 377-380 (1988).
7) Matsumoto, Y.: A Parallel Parsing System for Natural Language Analysis, *Proc. 3rd ICLP, Lecture Notes in Computer Science 225*, pp. 396-409 (1986).
8) 佐藤裕幸：並列自然言語構文解析システム PAX の改良，KL1 Programming Worksop '90, pp. 113-122, ICOT (May 1990).
9) 山崎重一郎：並列自然言語解析システム LaPuta について，KL1 Programming Workshop '90, pp. 100-112, ICOT (May 1990).
10) 奥村 晃，松本裕治：レイヤードストリームを用いた並列構文解析，第35回情報処理学会全国大会論文集，pp. 861-862 (1987).

(平成2年8月24日受付)
(平成3年5月 7 日採録)

松本 裕治（正会員）

昭和30年生．昭和52年京都大学工学部情報工学科卒業．昭和54年同大学院工学研究科修士課程情報工学専攻修了．同年電子技術総合研究所入所．昭和59～60年英国インペリアルカレッジ客員研究員．昭和60～62年(財)新世代コンピュータ技術開発機構に出向．昭和63年京都大学大型計算機センター助教授．平成元年京都大学工学部電気工学第2学科助教授となり，現在に至る．自然言語処理，論理プログラミング等に興味を持つ．

奥村 晃（正会員）

1960年生．1983年筑波大学情報学類卒業．同年沖電気工業(株)入社．1986～90年(財)新世代コンピュータ技術開発機構に出向．現在沖電気工業(株)総合システム研究所勤務．論理プログラミング，知識獲得等に興味を持つ．