TR-268

レイヤードストリーム を用いた並列プログラミング

奥村　晃，松本裕治

June, 1987

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

レイヤードストリームを用いた並列プログラミング

奥村 晃、松本 裕治
(財)新世代コンピュータ技術開発機構

概要

GHCやConcurrent Prolog等の言語で探索問題を解くためのプログラミング手法について論じる。まず、並列実行に適した、階層化された再帰的構造「レイヤードストリーム(layered stream)」を導入し、これによって並列度の高い処理が可能になることを示す。つぎに、レイヤードストリームを用いたプログラミングについて例を用いて説明する。探索問題を解く手続きを大きく三つに分け、各々に応じて容易にプログラムが作成できることを示す。

1. はじめに

committed-choice language (以下、CCL)と呼ばれる言語群は論理型言語Prologを並列化したものとして考案された。Prologはユニフィケーションとバックトラッキングを基本機構として持ち、探索問題に適した言語であったが、CCLはバックトラック機構を持たないため探索問題の扱いには一考を要する。例えばPrologでは解の構造の一部を共有し、別の一部をバックトラックで置き換えていって解を求めることが出来る。CCLの場合、バックトラックするかわりに最初から選択枝の数だけ探索手続きを起動することが考えられるが、一部を共有する構造を複数の環境で保持するのでは効率の良い実現は望めない。

本稿ではCCLで探索問題を並列に解くために新しく考案したデータ構造「レイヤードストリーム」を紹介し、それを用いたプログラミング手法について論じる。

2. 探索問題とCCL

探索問題では探索木のORノードにおいて各枝の可能性を等しく勘案する必要がある。Prologでは、とりあえず一つの枝について探索を継続し、選択されなかった枝については選択点を記録することによってバックトラック時の再選択を可能にしている。

CCLで同様にプログラムを行っても、選択されなかった節の可能性はコミット時に棄てられてしまうので、一般に正しい答えが発見されるとは限らない。つまり問題の宣言的記述をそのままCCLの述語定義に映しても正しいプログラムとはならない。

CCLで探索問題を解く方法としては、OR並列言語のインタプリタをCCLで記述する方法や、制限付のPrologプログラムからコンパイルする方法[上田 85][Tamaki 87]などが提案されているが、どれも間接的なものである。これらの方法で効率の良い実行を望むなら、インタプリタやコンパイラの処理の内容を熟知していて、場合によってはコードを書き替えることも必要となる。

効率的なプログラムを得るためには、探索手続きをプログラマがそのままCCLのコードで記述するのが早道と考える。探索問題を直接CCLで簡単に記述して、しかも効率の良いプログラミング技法は考えられないだろうか。

3. 探索問題の一般的構造

探索問題を解く手続きを、探索木をルート節点からリーフ節点まで枝を伸ばして行くものとして捉える。途中の各節点では、ルートからその節点まで辿ってきた履歴をもとに新しい節点を決定する。この手続きは次のように再帰的に定義できる。

1. 現在までの履歴を基に新しい節点を一つ求める
2. 新しい節点に対し同じ手続きを適用する

実際のプログラミングにおいても上の構造が多く現われる。また、探索問題の解はルートからリーフまでの一連の節点の並びで表現できる。これは再帰的データ構造で表すのが適当でありここではリスト構造を想定する。この場合プログラムの各ステップは前段までのリストと競合しない新しい要素を一つ求めて、それを結果のリストに付け加えて次段に渡すということを行う。新しい要素となる可能性のある候補はあらかじめ分かっているのが普通で、そのなかから競合しないものを選択することになる。つまり、先の手続きは、

1. 候補集合Xsを求め、そこから新しい要素Xを一つ決める
2. 現在までの履歴LとXが競合しないことを確かめる

3.[X|L]について同じ手続きを適用する

1では、一般に複数存在する候補の中からなんらかの基準で一つを選択する。Prologの場合、残りの候補についてはバックトラック機構によって順次選択される。CCLではそれができないので、常にすべての可能性を求めて手続きの各段を進んで行くのが一つの自然な解法であろう。つまり各段では、可能なすべての[X|L]が生成される。この場合、各段の入力はその段までの可能なすべてのLの集りである。これをLSというリストで表現すると各段の手続きは次のようになる。

a-1　候補集合Xsを求める
a-2　Xsの各要素Xにbを適用する
a-3　bの解を集めてそれにaを適用する

b-1　LSのうち、Xと競合しないものの集りをLS'とする
b-2　Xを頭部としLS'の要素の一つを尾部とするリストの集りを返す

これは玉木のストリーム方式[Tamaki 87]の基本的考え方である。b-2でLS'の各要素にXを追加することを行っているが、このリストは次段で再び分解されて次の要素と競合しないかどうかの検査が行われる。検査はほとんどの場合リストを頭から分解して行われるので、b-2でわざわざXをLS'の要素の各々に追加するのは無駄な処理と思われる。

4.レイヤードストリーム

あるXについてbを行った結果は、Xをその頭部として共有するリストの集りである。各リストは次段で再び頭から分解されることが分かっているので、実際そういうリストの集りを生成するかわりに「Xを頭部として共有する」ことを示す構造を結果として返すことにする。二項演算子*を導入し、Aを頭部としリストDsの各要素を尾部とするリストの集りをA*Dsと表現することにすれば、先の手続きのbの答えはX*LS'となる。Xs＝[X1,X2,...]とし、LSのうちXn(n＝1,2,..)と競合しないものの集りをLSnとすると、a-3で次段に渡すリストは、[X1*LS1,X2*LS2,...]となる。各段でこの形のリストを次段に渡すので、LSやLSnも同じ形をしている。このようなデータ構造をレイヤードストリームと呼ぶ。つまりレイヤードストリームは*による構造とリスト構造からなる再帰的複合構造である。各層は*による構造のリストであり、外からただちに参照可能な最上位の層を表層、または第一層と呼び、*の後ろ部分を第二層、さらに内部に入り込む毎に第三層、第四層という風に呼ぶことにする。

レイヤードストリームを用いて探索問題を解くことを考える。各段の手続きは3章のものとほとんど同じであるが、並列実行のモデルとして考えると大きな違いがある。まず、b-2は単にXとLS'を*で結合するだけなのでb-1の結果を待たずに実行できる。そうするとa-3もbが終了する前に実行可能であり、この段の結果の一部を次段で逸早く参照できることになる。つまりレイヤードストリームの各要素について*の後ろの部分(可能な尾部)は未決定の段階で、*の前の部分(共有される頭部)は次段で競合検査の対象となるわけである。これが各段で起きるため各段の競合検査が並列に実行できる。

つぎに競合検査の手続きを考える。競合検査の手続きはXとLSを受け取ってLSの要素でXと競合しない要素の集りLS'を返す。LSはレイヤードストリームであり、*による構造の各々が頭から調べられる。この手続きは次のように定義できる。

c-1　LSの表層の要素の各々についてdを行う
c-2　dの答えをまとめて返す

d-1　LSの表層の要素をXX*LLとするXとXXが競合していたら何も返さない
d-2　XとXXが競合していないならば、XとLLについてcを行い結果をLL'とする
d-3　XX*LL'を返す

dはまず*の前を調べて競合しないようなら*後方のレイヤードストリームについて競合検査を継続する訳である。ここでもd-2とd-3は並列実行可能である。つまりXXはLL'が決定されるのを待たずに出力可能であり、次段の競合検査の対象とできる。

総じて言えば、探索手続きの各段は次段に可能なXを送るのと同時に前段からのLSの競合検査を開始する。LSははじめ表層しか参照できないが、前段の競合検査が進行するにつれてしだいに内部の層が参照できるようになり、検査がすみ次第、結果は次段に送られ次段の検査の対象となる。これが各層で同時に並列して行われるため並列度の高い処理が期待できる。

5.レイヤードストリーム・プログラミング

前章の手続きに基づくプログラミング・スタイルを提案する。基本的には上のa、bとc、dの手続きに対応してプログラムを構築するが、その前にaについて若干の見直しを行う。

aはそれ自身再帰的な手続きである。ところが、その中のa-2はXsの要素の各々に対してbを行うの

でa-2の部分も再帰的なプログラムとなる。よってaは二重再帰になるのでaを二つの手続きに分けて次のように再定義する。

top-1　入力LSにaを適用する
top-2　aの結果にtopを適用する

a-1　候補集合Xsを求める
a-2　Xsの各要素(X)にbを適用する
a-3　bの解を集めて返す

こうすることによってa、bは一つの段の手続きとなる。また、topは各段の手続きの結果をレイヤードストリームで次段の手続きに渡すことに相当する。上のtopの手続きは再帰的に定義されているが問題によってはあらかじめ必要な段数が判明している場合もある。ここではより一般的にtopを再定義する。

top　必要なだけaの手続きを起動し、それらの間をレイヤードストリームで結合する

bは単に競合検査の手続き(c、d)を起動するだけなので、以降はaとひとまとめにしてabとして取り扱う。

競合検査を行うcとdのうち、cはリスト構造を分解しdは*の構造を分解する。しかしcとdの手続きを融合して、リストを分解しながら*の前の部分を検査したほうが効率的である。よってcとdもひとまとめにしてcdとして扱う。

結局手続きはtop、ab、cdの三つになった。ここで再び各々を定義し直すことはしないが、全体の中での各部の働きをまとめておく。

top　問題の構成に応じてabの手続きを起動する
ab　その段での解の要素の候補を発生し、その各々についてcdを起動する
cd　前段までの結果を検査しその段での要素と競合しないものを返す

topは起動したabのプロセスをレイヤードストリームによって結合する。各abは前段から入力を受け取るが最前段のabには入力としてbeginというアトムを与えることにする。これはレイヤードストリームの最低層を示し、いかなる要素とも競合しない。つまりcd手続きがbeginを発見したら検査の終了を意味する。

レイヤードストリーム・プログラミングは、個々の問題の持つ性質を上の三つの手続きとして捉えることによって実現される。つまり、三つの手続きは問題の解構造、解を構成する要素の候補、解の要素間の制約を反映するもので、探索問題の宣言的記述からの自然なプログラミングが可能である。

6. プログラム例

ここでは前章の手法に基づくプログラムの例を示す。プログラムはすべてGHC[Ueda 85]で書かれているが、Concurrent Prolog[Shapiro 86]、PARLOG[Clark 86]等の他のCCLでも同様にプログラミング出来る。

6.1　nクイーン問題

問題は、n行n列のチェッカボードにn個のクイーンを縦横斜めのどの方向でも重なることのないように配置する問題である。各列にちょうど一つずつクイーンが配置されるのは自明であるので、問題を、「横も斜めも重なることのないように各列のクイーンの縦位置を決定する」と言ってもよい。縦位置としては1からnまでが考えられる。すると、この問題における三つの手続きは次のようになる。

top　ボードの列に対応してn個のabを順次レイヤードストリームで結合する
ab　ボードの行に対応してn個のます目の要素を発生し、各々についてcdを起動する
cd　前列までの結果のうち、その列の要素と重ならないものを返す

nをパラメータとして与えてnクイーンの問題を解くプログラムは上の三つに対応して単純に作成できるが、説明をさらに容易にするためここでは4クイーン問題について扱う。プログラムを図1に示す。fourQueen、q、filterがそれぞれtop、ab、cdに相当する。三つの各々について簡単に説明する。

fourQueen：4クイーンのボードは4行4列である。各列についてqを起動する。最初のqにはbeginを入力する。最後のqの出力が4クイーンの答えである。

q：一つの列には1から4までの四つの位置が考えられる。四つの各々についてfilterを起動する。1から4までの数字と各filterの出力を*で結合したものをリストにして返す。

filter：第一引数に渡される前列までの結果を、第二引数であるその列の縦位置と重ならないかど

```
fourQueen(Q4) :- true |
    q(begin,Q1),
    q(Q1,Q2),
    q(Q2,Q3),
    q(Q3,Q4).

q(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([I*_|Ins],I,D,Out) :- true |
    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, filter(In1,I,D1,Out1),
    filter(Ins,I,D,Outs),
    Out = [J*Out1|Outs].
```

図1. 4クイーン問題のプログラム

うかを検査し、大丈夫なものだけを第四引数に返す。

第一節はbeginを検出するものである。レイヤードストリームの最下層であり検査を終了する。

第二節は空リストを検出する。検査すべきレイヤードストリームの要素がもうないので処理を終了する。

第三節は横に重なる場合である。検査中のレイヤードストリームの先頭とその列の縦位置が同じなので、横に重なる。その要素は読み飛ばされ、残りの要素について処理が継続される。

第四、第五節は斜めに重なる場合である。filterの第三引数は斜めの重なりを検査するためのパラメータで、入力レイヤードストリームの現在検査中の層に対応する列と、親手続きのqの対応している列との間の隔たりを示す。縦位置の差がこの隔たりと同じなら斜めに重なっていることになるので、その要素は読み飛ばされる。

最後の節は先頭要素の最上層が検査に合格した場合である。その要素の*の後ろの部分(内層)と、残りの要素について処理を継続すると同時に、先頭要素の*の前の部分を出力する。

qの最後の行に注目していただきたい。filterの起動と出力パターンの生成を並行して行っている。つまり、filterの結果を待つことなく表層を次段に出力出来る。次段のfilterは表層を受け取っただけで処理を開始できるので、各列のfilterは同時に並列して処理を行える。filterの定義についても同じ事が言える。最後の節によって検査の終わった要素はつぎつぎ出力される。これは次段のfilterによって順次処理される。こうして各列のfilterは同時並列処理を継続できる。

6.2 グッドパス問題

図2のようなグラフ上で任意の二点間を結ぶループを含まない(同じ節点を二度以上通らない)経路を見付けたい。

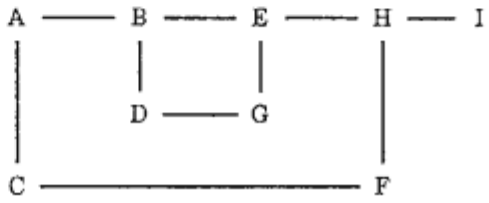


図2. 簡単なグラフの例

この問題の解法としては、始節点から出発して隣接する節点の中から未到節点であることを確認しながら一歩ずつ進んで行くというのが一般的であろう。レイヤードストリームを用いた処理では各節点からの経路の探索が同時に起動され全体が並列に動作する。

nクイーンではn個のab手続きが協調して解を生成するが、グッドパスでは経路はすべての節点を通る必要はなく、解は経路を構成する一部のab手続きから生成される。それゆえレイヤードストリームによる連結は直線的ではなく、解の要素の並びうる順序、つまりグラフ上での結合に従って連結を行う。図3にプログラムを示し、以下で各部の説明を行う。

goodPath : 各節点毎にnodeの手続きを起動する。グラフの枝に従ってレイヤードストリームでnodeを連結する。nodeの第四引数が入力、第五引数が出力である。例えば、節点eで入力が[B,G,H]とあるのは節点b、g、hの出力をリストでまとめて受け取ることを表している。goodPathは、始節点名と終節点名をnodeに引き渡す。第一引数のStartが始節点名、第二引数のGoalが終節点名を受け取るための引数である。

```
goodPath(Start,Goal,Path) :- true |
     node(Start,Goal,a,[B,C],A,Path),
     node(Start,Goal,b,[A,E,D],B,Path),
     node(Start,Goal,c,[A,F],C,Path),
     node(Start,Goal,d,[B,G],D,Path),
     node(Start,Goal,e,[B,G,H],E,Path),
     node(Start,Goal,f,[C,J],F,Path),
     node(Start,Goal,g,[D,E],G,Path),
     node(Start,Goal,h,[E,I,J],H,Path),
     node(Start,Goal,i,[H],I,Path),
     node(Start,Goal,j,[F,H],J,Path).

node(S,_,S,_,Out,_) :- true |
     Out = S*begin.
node(_,G,G,In,Out,Path) :- true |
     Out = nil, Path = G*In.
node(S,G,N,In,Out,_) :- S \= N, G \= N |
     Out = N*In1, filter(In,N,In1).

filter(begin,_,Out) :- true | Out =
begin.
filter([],_,Out) :- true | Out = [].
filter([nil|In],Node,Out) :- true |
     filter(In,Node,Out).
filter([Node*_|In],Node,Out) :- true |
     filter(In,Node,Out).
filter([N*Ns|In],Node,Out) :- Node \= N |
     Out = [N*Ns1|Out1],
     filter(Ns,Node,Ns1),
     filter(In,Node,Out1).
```

図3. グッドパス問題のプログラム

nodeはこれらと第三引数に与えられた自分の節点名を見比べて、自分が始節点や終節点であるかどうかを知ることが出来る。

node : 第三節が一般の節点についての手続きである。nodeが求めるべき解の要素の候補は唯一その節点名だけであるので、単に自分の節点名を*の前の部分として出力する。それ以外ににnodeは自分が始節点または終節点である場合、それに応じた処理を行う。

第一節は自節点名と始節点名が同じなので始節点の処理を行う。この場合派、自分の節点名とbeginを*で結合して出力する。

第二節は自節点名が終節点名と同じなので終節点用である。終節点を通過するような経路はグッドパスになり得ないので、節点名を出力する事はせず、かわりにアトムnilを出力する。また、終節点に至る経路そのものが解であるから、解出力用の第六引数に自分の節点名と入力引数を*で結合したものを返す。他の節点への出力に自分の節点名を与えていないので、入力の中に自分の節点名があるかどうかを検査する必要はない。

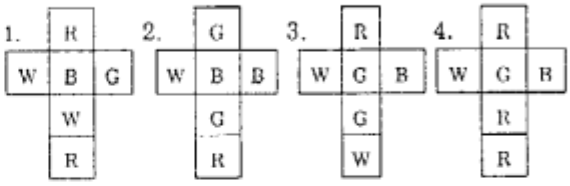
filter : 競合の検査は単に自分の節点名を含む要素を入力から除去するだけなので、nクイーンに比べて単純である。第一、第二節はnクイーンのものと同じである。第三節は終節点の出力を読み飛ばすためのものである。第四節は同じ節点名を発見した場合で、やはり読み飛ばす。第五節は先頭要素が検査に合格した場合のもので、nクイーンの第六節と同様、内層と残りの要素の検査を継続すると同時に出力の生成を行う。

このプログラムでは各節点で一斉に競合検査の処理を行う。解となる経路に全然関与しない節点についても他と同様に処理を行う。また、各節点で独立して経路を生成するため、始節点に向かって大きく逆行する経路も生成される。これらは、始節点から一歩ずつ進む方法では起こりえない現象なので、本方式のほうが全体の処理量としては大きくなることがあり得るが、並列に実行したときのステップ数では本方式が数倍すぐれている。つまり、充分な数の処理要素をもつ並列計算機上では、レイヤードストリーム方式は処理時間の大幅な低減が期待できる。

6.3 四色キューブ問題

次の問題を考える。「各面を四色の何れかで塗られている四つの立方体がある。これを一列に並べると高さが4の正四角柱ができるが、その四つの側面の何れにも一つの色が二度以上出現しないように立方体を配置せよ」

これを、四色キューブ問題と呼ぶ。四つの立方体の展開図を図4に示す。



R = Red　　G = Green
W = White　　B = Blue

図4. 四色キューブの例

この問題では底面の色を問わないので、キューブの順序は関係ない。プログラムでは上の1から4の順で配置することにする。各位置のキューブをどの向きに置くかが問題である。図5にプログラムを示す。ab手続きの部分(part 2)が少々複雑になっているが、これも簡単なプログラムである。プログラムを三つの部分に分けて説明する。

```
% part 1 %
cubes(S4) :- true |
     cube(1,Q1), set(Q1,begin,S1),
     cube(2,Q2), set(Q2,S1,S2),
     cube(3,Q3), set(Q3,S2,S3),
     cube(4,Q4), set(Q4,S3,S4).
cube(1,Out) :- true |
     Out = q(p(w,g),p(r,w),p(b,r)).
cube(2,Out) :- true |
     Out = q(p(w,b),p(g,g),p(b,r)).
cube(3,Out) :- true |
     Out = q(p(w,b),p(r,g),p(g,w)).
cube(4,Out) :- true |
     Out = q(p(w,b),p(r,r),p(g,r)).
% part 2 %
set(q(P1,P2,P3),In,Out) :- true |
     rotate1(P1,P2,In,Out,Out1),
     rotate1(P1,P3,In,Out1,Out2),
     rotate1(P2,P3,In,Out2,[]).
rotate1(P1,P2,In,S,T) :- true |
     rotate2(P1,P2,In,S,SS),
     rotate2(P2,P1,In,SS,T).
rotate2(p(C1,C2),P2,In,S,T) :- true |
     rotate3(C1,C2,P2,In,S,SS),
     rotate3(C2,C1,P2,In,SS,T).
rotate3(C1,C2,p(C3,C4),In,Out,T) :-
          true |
     filter(C1,C2,C3,C4,In,Out1),
     filter(C1,C2,C4,C3,In,Out2),
     Out=[q(C1,C2,C3,C4)*Out1,
          q(C1,C2,C4,C3)*Out2|T].
% part 3 %
filter(_,_,_,_,[],O) :- true | O = [].
filter(_,_,_,_,begin,O) :- true |
     O = begin.
filter(C1,C2,C3,C4,[q(X,_,_,_)*_|I],O) :-
     C1 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,X,_,_)*_|I],O) :-
     C2 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,_,X,_)*_|I],O) :-
     C3 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(_,_,_,X)*_|I],O) :-
     C4 = X | filter(C1,C2,C3,C4,I,O).
filter(C1,C2,C3,C4,[q(P,Q,R,S)*J|I],O) :-
          C1\=P, C2\=Q, C3\=R, C4\=S |
     filter(C1,C2,C3,C4,J,O1),
     filter(C1,C2,C3,C4,I,Os),
     O=[Q*O1|Os].
```

図5. 四色キューブ問題のプログラム

part1(top手続き)

cubesは四つのキューブの定義を取り出し、setで向きを決める。ここでは、キューブの定義を二層になった構造体で表している。cubeのボディ部に示されているように、一つのキューブは、互いに反対になる面同士の3つの組み合わせとして定義した。

part2(ab手続き)

ab手続きは、解の要素となりうる候補を生成する。nクイーンでは1からnまでの整数を発生し、グッドパスでは節点名を発生するだけで、ごく単純なものであったが、四色キューブ問題ではキューブの可能な置き方をすべて見付けなくてはいけないので多少複雑である。

キューブの置き方は、側面となる四つの面の色で表現する。四つの面を、上、下、左、右と呼ぶ。まずsetでは三つある面の組の内からどの二つを側面に用いるかで三つの可能性を発生している。rotate1は採用された二つの組の内、どちらを上下としどちらを左右とするかで二つの可能性を発生する。rotate2の第一引数は上下であり、第二引数は左右である。rotate2は上下となった組について上の面と下の面を決定する。ここでも二つの可能性がある。最後にrotate3は左右となった組について左と右を決定する。上下と同様二つの可能性を持つ。結局、3×2×2×2=24の可能性が生じ、その各々についてfilterが発生する。

プログラムが四段構えになってしまったが、24の可能性を一気に並べ立ててももちろん構わない。

part3(cd手続き)

この問題の持つ制約は、「一つの側面に同じ色が二度以上現われてはいけない」というものである。cd手続きはこの制約を満たさない要素を入力から除去する。

filterの第一、第二節は前二例と同じであるから、説明は不要であろう。第三から第六までの節はそれぞれ、上、下、左、右に同じ色を発見した場合で、何れの場合もその要素は読み飛ばされる。最後の節はやはり前二例と同じで内層の検査と残りの要素の検査を継続すると同時に出力を行うものである。

この問題は探索木の深さが四段しかないが、各段で24もの可能性があるので探索空間は横に大きく広がる。そのため並列処理を行う効果は大きい。

7. レイヤードストリームによる全解の表現

前章で紹介した4クイーン・プログラムの出力結果を図6に示す。

```
| ?- ghc fourQueen(Q).
52 msec.

Q = [    1*[3*[],4*[2*[]]],
         2*[4*[1*[3*begin]]],
         3*[1*[4*[2*begin]]],
         4*[1*[3*[],2*[]]]

yes
```

図6. 4クイーン・プログラム出力結果

表層の要素から内層へ構造を辿ってbeginに至る経路の一つ一つが解である。いくつかある空リスト([])は競合検査時に全要素が読み飛ばされた場合に生じる。空リストを*の後ろに持つ要素は解とならない。つまり、上の構造は、[2,4,1,3]と[3,1,4,2]の二つの解を表すものである。

この構造を、別のレイヤードストリーム・プログラムへの入力とすることは当然可能であるが、通常の全解リスト[[2,4,1,3],[3,1,4,2]]による解出力を要求されることも考えられる。その場合、図6の構造から変換することも可能であるが、変換プログラムは一種の探索プログラムになってしまい、好ましくない。従ってレイヤードストリームを用いた探索の最終段階で通常のリスト出力に切り換えることを考える。これは容易で、最終段の競合検査手続きが、構造の内層に向けて処理を継続するのと同時に、構造の各層の要素(*の前の部分)をスタックに記録しておけば、beginを発見したときにスタックの内容が解の一つとなっている。これを差分リストで収集すれば全解リストが出来上がる。この考えに基づいて特殊化したlastFilterとこれを起動するlastQを図7に示す。図1のプログラムのfourQueenの節の最後のqの呼び出しをlastQで置き換えることにより、目的を達成できる。新しいプログラムは元のプログラムと出力形式が違うだけなので、リダクション数等(次章を参照)には影響しない。

```
lastQ(In,Out) :- true |
    lastFilter([1],In,1,1,Out,Out1),
    lastFilter([2],In,2,1,Out1,Out2),
    lastFilter([3],In,3,1,Out2,Out3),
    lastFilter([4],In,4,1,Out3,[]).
lastFilter(Stack,begin,_,_,S,T) :- true |
    S = [Stack|T].
lastFilter(Stack,[],_,_,S,T) :- true |
    S = T.
lastFilter(Stack,[I*_|Ins],I,D,S,T) :-
    true |
    lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*_|Ins],I,D,S,T) :-
    D =:= I-J |
    lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*_|Ins],I,D,S,T) :-
    D =:= J-I |
    lastFilter(Stack,Ins,I,D,S,T).
lastFilter(Stack,[J*In|Ins],I,D,S,T) :-
    J \= I, D =\= I-J, D =\= J-I |
    D1 := D+1,
lastFilter([J|Stack],In,I,D1,S,SS),
    lastFilter(Stack,Ins,I,D,SS,T).
```

図7. 特殊化された競合検査手続き

8. 実行効率

レイヤードストリーム・プログラミングは、並列度が高くステップ数の少ない実行を可能にする。この章では本方式の実行効率について他方式との比較を行う。比較の対象として[上田85]の方式と[Tamaki 87]の方式を選ぶ。この二つは制限付のPrologプログラムからCCLに変換するものであり本方式とは立場を異にするが、変換されたプログラムは探索問題を並列に解くことを目的としているので比較を行うのに適当である。なお、ここでは前者を上田方式、後者を玉木方式と呼ぶ。

6章の例題の各方式によるプログラムの実行結果を表1に示す。Red.はリダクション数、Sus.はサスペンション数、Cyc.はサイクル数、そして、Para.は並列度を表す。尚、上田方式、玉木方式の

	Red.	Sus.	Cyc.	Para.
=<6クイーン>=				
上田方式	2932	0	80	36.65
玉木方式	3161	1566	69	45.81
本方式	1340	124	25	53.60
=<8クイーン>=				
上田方式	48543	0	133	364.98
玉木方式	53824	25033	112	476.63
本方式	19418	2470	38	511.00
=<グッドパス>=				
上田方式	205	0	48	4.27
玉木方式	312	92	62	5.03
本方式	181	26	13	13.92
=<四色キューブ>=				
上田方式	47383	0	33	1435.85
玉木方式	62601	15810	121	517.36
本方式	8085	2033	44	183.75

表1. 例題プログラムの実行結果

プログラムは、[上田85]、[Tamaki 87]に基づいてPrologプログラムから変換したものを一部最適化したものである。最適化は機械的に出来るものに留め、全体のアルゴリズムを変えるような変更は行っていない。参考のため、4クイーン問題の両方式のGHCプログラムを付録1、2に示しておく。

リダクション数とは、実行が終わるまでにcommit operationが何回行われたかを示す。処理は一サイクルを単位として進行する。一サイクルとは、前のサイクルが終わった時点で残っているゴールのうち、commitできるものすべてをcommitするものである。サイクル数は実行終了までにこれが何回繰り返されたかを示す。commitできなかったものはサスペンションと呼びマークを付ける。マーク付けされたゴールの総数がサスペンション数である。commitで生じた新しいゴールとサスペンションとなったゴールが次のサイクルでcommitの対象となる。並列度はリダクション数をサイクル数で割ったもので、各サイクル平均いくつのリダクションが起こったかの目安である。

この結果を基に各方式の比較を行う。定性的な比較がまだできていないので、ここでは各方式の基本的性質や、プログラムコードから容易に推察されることだけを述べる。

まずリダクション数で比較すると、本方式は全ての例について他方式より少ない数字を示している。その比は大きい場合で6倍から7倍にもなっている。これは、本方式によるプログラムの全体の処理量が少ないことを示している。4例の中でグッドパスについては比較的差が少なくなっているが、これは6.2節の最後で述べた無駄な計算によるものである。

上田方式では、一つのゴールが完了してその結果を次のゴールに渡すときに、継続情報から解を構成する処理が行われる。これはかなり頻繁に行われるため、リダクション数に相当の影響を与える。しかし、これはリストの反転と同等の処理であり、処理系に与える負荷は軽微なものである。実際、逐次処理系上での時間比較では、本方式と同等の結果を出している。

玉木方式では、ゴール間の結果の受け渡しはすべてストリームで行われるため、ストリームを分解、合成するための処理が必要である。これがゴール毎に起きるためリダクション数が著しく増大する。値は上田方式のものよりさらに大きく、しかもリスト反転などといった簡単な処理ではないので、この結果を見る限り効率は最も悪い。しかし、[Tamaki 87]にあるように、依存関係のないANDゴールを処理するときに上田方式で起こる重複計算を避けることが出来るので、問題によっては上田方式を凌ぐ可能性がある。本方式では、依存関係のあるANDゴールの処理も並列に行われる事に注意されたい。

サスペンション数では上田方式のプログラムは常に0を記録している。上田方式ではAND関係のゴール呼出しは、変換前のプログラムでのゴール順序に従って起動される。元のプログラムでは入力変数には基底項の入力だけを仮定しているので、変換されたプログラムが未決定の入力引数によってサスペンドする可能性は本質的に存在しない。これはこの方式の大きな利点である。

玉木方式ではAND関係にあるゴールは、各ゴールの全解を求める手続きをストリームで結合して実行される。各手続きは同時に起動されるが、それぞれ自分より前のゴールの解を受け取らないと探索を開始できない。そのため、サスペンションは他の方式に比べて多くなる。

本方式では、各手続きが解の構造の決定した部分をつぎつぎ次段に転送するため、玉木方式よりはサスペンションが少ない。ただし、競合検査で切り捨てられた要素は出力されないため、その次のサイクルで次段の手続きがサスペンドするが、リダクション数に対してそう大きい数字とは言えない。

サイクル数では、四色キューブの例を除き、本方式は他の方式の三分の一から四分の一程度である。他の二方式は、元のプログラムのゴールの順序にしたがって部分解の受け渡しが起こるため、そこに逐次性が生じてしまう。それに対し本方式では、各段で一斉に部分解の生成を開始することが可能であり、また生成された部分解は直ちに各次段で参照可能となるので競合の検査も各段で同時進行できる。そのため並列性の高い処理が可能で短いサイクル数で処理が完了する。

以上のことは、四色キューブ問題にも当てはまるが、段数が4なのでこれを並列に行うことの優位性は少ない。また、レイヤードストリームの各層はリスト構造で実現されており、各要素を検査するためにはリストの長さ分だけのサイクル数を要する。リストの長さは各段の候補の数であり、この問題では24と段数の4に較べて大きいため、候補数の影響の無い上田方式より全体のサイクル数が多くなっている。しかし各候補についての検査は独立した処理なので、各々を一斉に並行して行うことが本質的に可能である。それの実現は今後の課題である。

9. まとめと今後の課題

CCLのプログラムを評価する基準はまだ確立されていないが、サイクル数は一つの指標として有効であろう。この指標に従って考えると、レイヤードストリーム・プログラミングは処理効率を獲得するのに有効であることが実験で確認された。

また効率以外の長所として、プログラムの書きやすさを上げることが出来る。レイヤードストリーム・プログラミングを行うには、5章で述べたように、個々の探索問題の持つ性質に応じて三つの手続きを書き表せば良い。まずtopの部分は、問題の構造に従ってab手続きを連結する。abは、考えられる候補を発生し、その一つ一つについてcdを起動する。そしてcdは要素間に存在する制約に基づき競合を検出する。問題の構造、要素の候補、要素間の制約、この三つを抽出できれば、後は同じスタイルで容易にプログラミングできる。

レイヤードストリーム・プログラミングの基本的考え方は、複数の構造の部分を共有化してプロセス間のデータの流れを効率化し、並列度を獲得するものである。これは、並列パーザ、PAX [Matsumoto 86]で解析情報を伝達するのに用いているアイデアを一般化したもので、本方式側から見ると並列パーザは一種の変形となっている。現在、新たにレイヤードストリームによるパーザについて検討しており、PAXとは少し異なる特性を実験で記録している。両者の比較検討については、別の機会に行う。

今後の課題としては、実行効率について定性的評価を行い、レイヤードストリーム・プログラミングが有効であるような問題のクラスを明らかにしたい。そのためには、CCLプログラムを評価する基準が必要であるが、言語の処理方式が決まらないと論じるのは困難であろう。一部の例題については、ある種の処理方式を仮定したシミュレーション結果が既に得られているが、別の機会に紹介したい。

ここでは直接CCLのプログラムを書き下す方法について述べたが、抽出すべき問題の性質を表現できる枠組みがあれば、そこからCCLにコンパイルすることも考えられる。適当な表現言語を用意できれば、探索問題のプログラミングはさらに簡単になるであろう。必要な性質は既に整理されているので、直接書き下すのと同等の効率を示すコードを生成するコンパイラが可能と考える。

謝辞

ICOT研究所の古川康一次長、上田和紀主任研究員には有益なコメントを頂いた。また、宮崎敏彦研究員にはGHCプログラム評価実験用ツールを提供していただいた。各氏に感謝する。

参考文献

[上田 85]上田和紀:全解探索プログラムの決定的論理プログラムへの変換、日本ソフトウエア科学会第2回大会論文集.

[Clark 86] Clark, K. L. and Gregory, S.: "PARLOG: Parallel Programming in Logic," ACM Trans. Program. Lang. Syst. 8, 1.

[Matsumoto 86] Matsumoto, Y.: "A Parallel Parsing System for Natural Language Analysis," Proc. the 3rd ICLP.

[Shapiro 83] Shapiro, E. Y.: "A Subset of Concurrent Prolog and Its Interpreter," ICOT Tech. Report TR-003.

[Tamaki 87] Tamaki, H.:"Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages", Proc. the 4th ICLP.

[Ueda 85] Ueda, K.: "Guarded Horn Clauses," ICOT Tech. Report TR-103.

付録1. 4クイーン(上田方式)

```
q(B) :- true |
     'sweeper$q1'([1,2,3,4],[],'L1',B,[]).

'sweeper$q1'([H|T],R,Cont,Rs0,Rs1) :-
          true |
     'sweeper$sel'([H|T],'L2'(Cont,R),
          'L2',Rs0,Rs1).
'sweeper$q1'([],R,Cont,Rs0,Rs1) :- true |
     Rs0 = [R|Rs1].

'sweeper$sel'(HT,Cont,Conts,Rs0,Rs2) :-
          true |
     'sel/3#1'(HT,Cont,Conts,Rs0,Rs1),
     'sel/3#2'(HT,Cont,Conts,Rs1,Rs2).

'sel/3#1'([A|L],'L2'(Cont,R),Conts,
          Rs0,Rs1) :- true |
     'sweeper$check1'(R,A,1,
          'L2b'(Cont,R,A,L,Conts),Rs0,Rs1).
'sel/3#1'([],Cont,Conts,Rs0,Rs1) :- true |
     Rs0=Rs1.

'sel/3#2'([H|T],Cont,Conts,Rs0,Rs1) :-
          true |
     'sweeper$sel'(T,Cont,'L5'(Conts,H),
          Rs0,Rs1).
'sel/3#2'([],Cont,Conts,Rs0,Rs1) :- true |
     Rs0=Rs1.

'sweeper$check1'([H|T],U,N,Cont,Rs0,
          Rs1) :- H=\=U+N, H=\=U-N, N1:=N+1 |
```

```
    'sweeper$check1'(T,U,N1,Cont,Rs0,Rs1).
'sweeper$check1'([H|T],U,N,Cont,Rs0,
        Rs1) :- H =:= U+N |
    Rs0=Rs1.
'sweeper$check1'([H|T],U,N,Cont,Rs0,
        Rs1) :- H =:= U-N |
    Rs0=Rs1.
'sweeper$check1'([],U,N,
        'L2b'(Cont,R,A,L,Conts),Rs0,Rs1) :-
          true |
    b(Conts,'L3'(Cont,R,A),L,Rs0,Rs1).

b('L5'(Conts,A),Cont,T,Rs0,Rs1) :- true |
    b(Conts,Cont,[A|T],Rs0,Rs1).
b('L2','L3'(Cont,R,A),L,Rs0,Rs1) :- true |
    'sweeper$q1'(L,[A|R],Cont,Rs0,Rs1).
```

付録2. 4クイーン(玉木方式)

```
queens(Q) :- true |
    'Qq'([1,2,3,4],[],Q,[]).

'Qq'([],Y,Z0,Z1) :- true | Z0 = [Y|Z1].
'Qq'(X,Y,Z0,Z1) :- X \= [] |
    'Qsel'(X,UVs,[]), 'Iq21'(Y,UVs,Z0,Z1).

'Qsel'([],Z0,Z1) :- true | Z0 = Z1.
'Qsel'([X|Y],Z0,Z2) :- true |
    Z0 = [(X,Y)|Z1],
    'Qsel'(Y,UVs,[]),
    'Isel21'(X,UVs,Z1,Z2).

'Iq21'(Y,[(U,V)|UVs],Z0,Z2) :- true |
    'Qcheck'(Y,U,1,YY),
    'Iq22'(V,[U|Y],YY,Z0,Z1),
    'Iq21'(Y,UVs,Z1,Z2).
'Iq21'(_,[],Z0,Z1) :- true | Z0 = Z1.

'Iq22'(V,List,ok,Z0,Z1) :- true |
    'Qq'(V,List,Z0,Z1).
'Iq22'(_,_,ng,Z0,Z1) :- true | Z0 = Z1.

'Isel21'(X,[(U,V)|UVs],Z0,Z2) :- true |
    Z0 = [(U,[X|V])|Z1],
    'Isel21'(X,UVs,Z1,Z2).
'Isel21'(_,[],Z0,Z1) :- true | Z0 = Z1.

'Qcheck'([Q|R],P,N,Res) :-
        Q =\= P+N, Q =\= P-N |
    M := N+1,
    'Qcheck'(R,P,M,Res).
'Qcheck'([Q|R],P,N,Res) :- Q := P+N |
    Res = ng.
'Qcheck'([Q|R],P,N,Res) :- Q := P-N |
    Res = ng.
'Qcheck'([],_,_,Res) :- true | Res = ok.
```