ＫＬ１[*1]による探索問題のプログラミング

１研　大　木

１．　は　じ　め　に

人工知能や知識工学の問題をプログラミングする際、探索を必要とする場合がある。従来の言語のPrologでは言語自体にバックトラックという機構を持っているため、Prologのプログラマは、バックトラックを使うことによって探索を必要とする問題でも容易にプログラミングすることができた。一方、ＩＣＯＴで開発しているＫＬ１(Kernel Language 1 [ICOT 85]) はPrologと同様に知識工学や人工知能の問題を対象としているが、ＫＬ１はPrologと異なり、並列型言語で言語自体にバックトラックに相当する機構を持っていない。そのため、ＫＬ１を使って探索を必要とする問題を解決するにはプログラム・レベルで陽に探索を記述する必要がある。ここでは、ＫＬ１による探索問題のプログラミング方法を確立するために次の項目について検討した。

(1) 探索問題に対するプログラミングの方針、
(2) 探索問題の並列実行方式、
(3) ＯＲノードの並列実行の実現方式、
(4) ＡＮＤノードの並列実行の実現方式、
(5) 解の候補の生成方式。

そして、ふく面演算を例にして（ふく面演算を例にした並列型言語によるプログラミングには[Kornfeld 81] がある。）、これらの検討に基づいていくつかの方式でプログラムを作成した。

*1 「GHC による……」としなかった理由は、プログラミングの際に高遷なＧＨＣ [Ueda 85] の精神に従っていないところがあるからである。しかし、ＫＬ１の詳細な言語仕様は現在のところ決まっていないので、ここで使用したシンタックスはＧＨＣと同じものである。

２．　ＫＬ１による探索問題のプログラミング

ここでは、ＫＬ１による探索問題のプログラミング方式をいくつかの面から検討する。

２．１　探索問題に対するプログラミングの方針

探索を行なう場合の基本方針は、言語によらずgenerate and test 法である。generate and test 法とは、解の候補を生成(generate)し、それが条件を満足するか検査(test)を行ない解を求める方法である。ここでのＫＬ１による探索問題のプログラミング方式はすべてgenerate and test 法に基づいている。

２．２　探索問題の並列方式

探索の計算過程は一般に探索木で表わされる。一般の探索木はＡＮＤ／ＯＲ木である[Nilsson 80]。ＡＮＤ／ＯＲ木の探索を並列に実行する方式として、ＯＲ並列実行方式とＡＮＤ並列実行方式の２つがある。ＯＲ並列実行方式はＯＲノードに関する探索を並列に実行する方式のことで、ＡＮＤ並列実行方式はＡＮＤノードに関する探索を並列に実行する方式のことである。探索木は計算過程を表わしているので、同じ問題でも解き方によって探索木が異なることがある。

ふく面演算では図２．１のような探索木が考えられる。(a) は、英文字に関して解の候補を１つずつ割り当てていった場合の探索木であり、この探索木は純粋なＯＲ木である。一方、(b) は桁ごとの英文字に対して解の候補を１つづつ割り当てていった場合の探索木

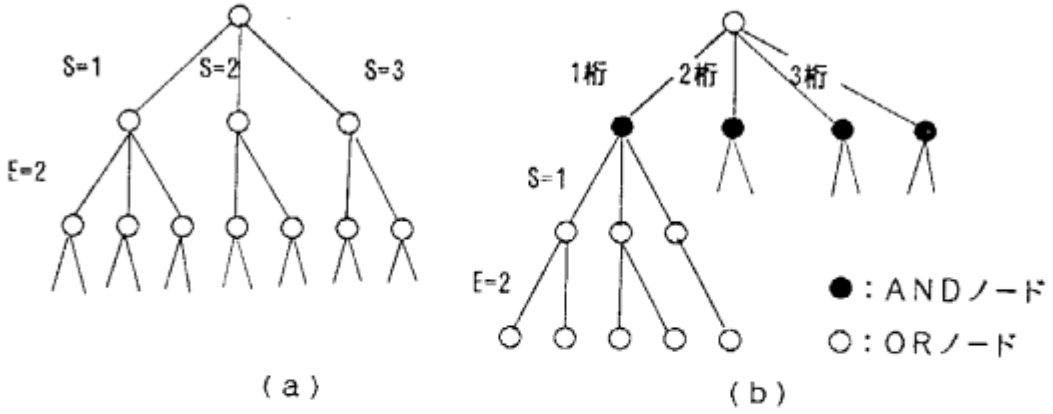


図２．１　ふく覆面演算の探索木

で、桁ごとの探索がＡＮＤになっている。

探索の並列性にはＯＲノードの並列に関するＯＲ並列とＡＮＤノードの並列に関するＡＮＤ並列の２種類があるため探索の並列実行方式としては、次の４種類が考えられる。

(1) ＯＲ並列　ＡＮＤ並列　実行方式

探索木に関してＯＲノードもＡＮＤノードも並列に実行する方式で、最も並列性が高いと考えられる。

(2) ＯＲ並列　ＡＮＤ逐次　実行方式

ＯＲノードは並列に実行するが、ＡＮＤノードは逐次に実行する方式で、pure Prolog などの並列実行方式は一般にこの方式である。

(3) ＯＲ逐次　ＡＮＤ並列　実行方式

探索は探索木のＯＲに本質的なところがあるので、この方式は特別な場合（ＯＲになっている枝が極めて少ない場合など）を除いて余り有効ではない。

(4) ＯＲ逐次　ＡＮＤ逐次　実行方式

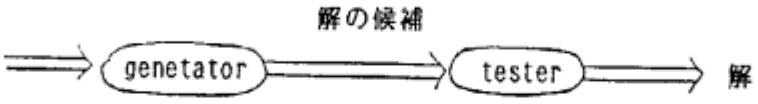
Prologの実行方式で、ＫＬ１で推奨できる方式ではない。

２．３　ＯＲノードの並列実行の実現方式

ＯＲノードをＫＬ１で並列に実行するための方式には次の３つの方式が考えられる。

(1) ストリーム方式

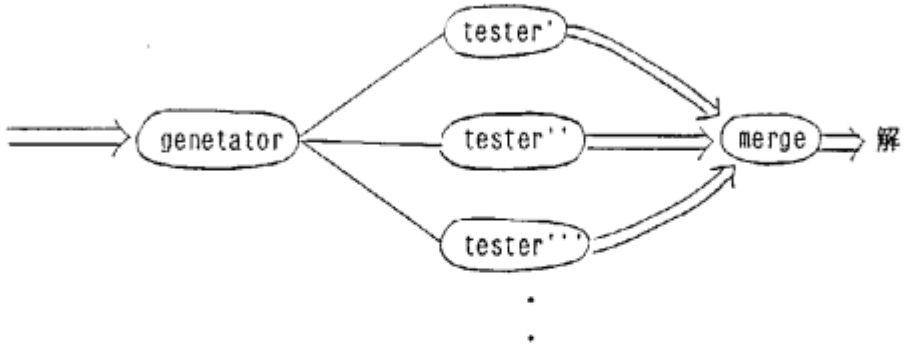
generator とtesterの間をストリームを使って解の候補を流す。generator は次々



genetator ： 解の候補を生成するもの

tester ： 解の検査を生成するもの

図２．２　ストリーム方式によるＯＲノードの並列実行方式



解の候補を引数としてtesterを生成する。

図２．３　プロセス生成方式によるＯＲノードの並列実行方式

に解の候補を生成するプロセスであり、testerはストリームで流される解の候補を次々に検査するプロセスである。図にすると図２．２のようになる。

(2) プロセス生成方式

図２．３に示すように、generator が個々の解の候補を検査するtesterを次々に生成し、testerを通った解はmerge で集められる。

(3) ガードＯＲ方式

ＫＬ１のガード部はＯＲ並列に実行されるため、この機構を使ってＯＲノードのゴールの探索を行なう。ただし、ガード機構を使っているため１つの解しか得ることができない。

ガードＯＲ並列実行　：　－
　　解の１つの生成　，　検査　｜　true
ガードＯＲ並列実行　：　－
　　残りの解に関するガードＯＲ並列実行　｜　true

この方式を図で表わすと図２．４のようになる。

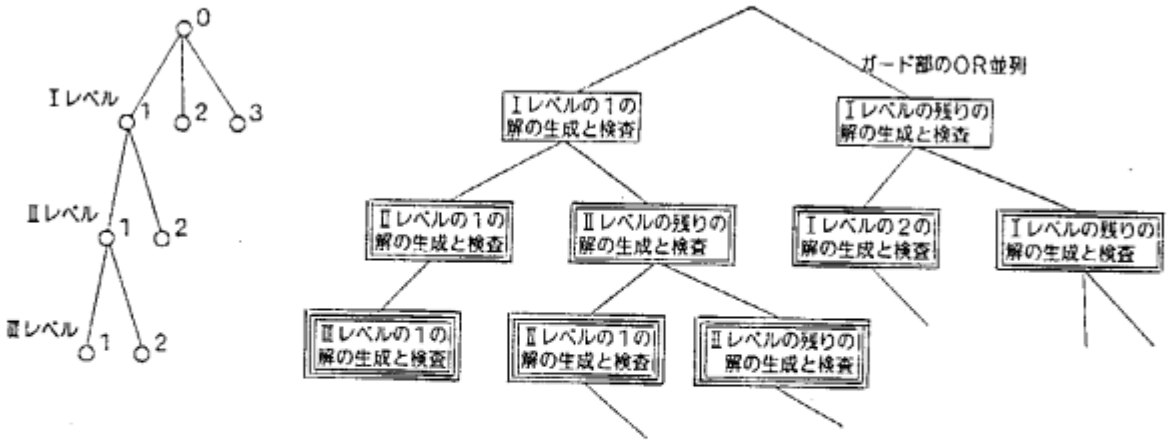


図２．４　ガードＯＲ方式によるＯＲノードの並列実行方式

２．４　ＡＮＤノードの並列実行の実現方式

探索においてＡＮＤノードを並列に実行することは、単にゴールをＡＮＤ並列に実行する以外に次の機能を実現する必要がある。

(1) ＡＮＤノードのゴールの実行が失敗に終わっても全体が失敗してはならない。

(2) ＡＮＤノードのゴールの１つが失敗したならば、残りのゴールの実行を中断する。

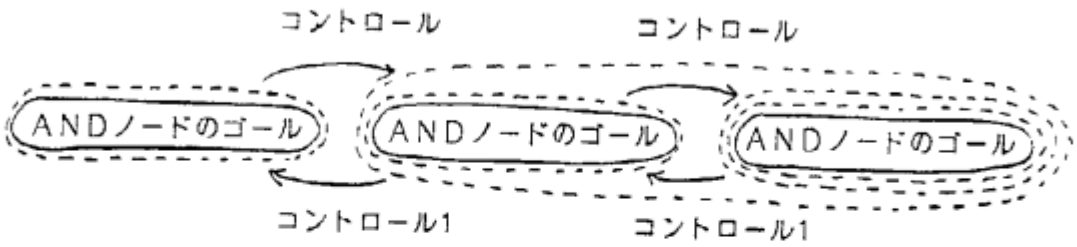
(3) ＡＮＤノードのすべてのゴールの実行が成功裏に終らない限り、ＡＮＤノードの上位のゴールの実行は失敗となる。

以上の機能を満たすＡＮＤノードの並列実行方式として次の方式が考えられる。

(1) メタコール方式

図２．５のようにＡＮＤノードのゴールをメタコール(meta call) でネストしてＡＮＤノードを並列に実行させる方式である。

ＡＮＤノードの複数ゴールを最初のゴールと残りのゴールとに分け、それぞれのゴールをメタコールでＡＮＤ並列に実行させる。メタコールの間では次のようなコントロールを行う。もし、最初のゴールが失敗すると、残りのゴールを解くメタコールを中断し、余分な計算を行なうことを防ぐ。同様に、後のゴールが失敗したならば、前のゴールのメタコールを中断する。また、自分自身も含めてすべてのＡＮＤゴールが



点線内はメタコールのもとで実行れているゴールである

図２．５　メタコール方式によるＡＮＤノードの並列実行方式

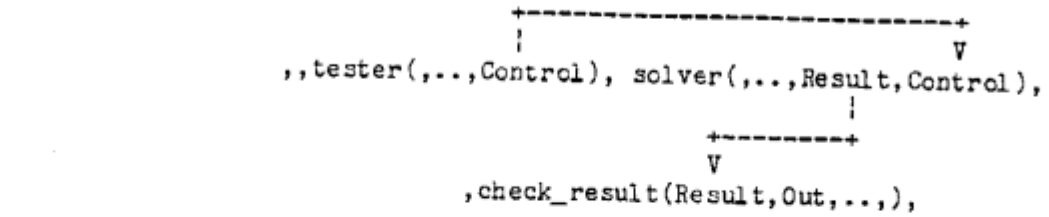
成功しない限り、上位のＡＮＤゴールに解を報告しない。

(2) ガード方式

メタコールの代わりにガードを使う。ＫＬ１では図２．６に示すようなプログラムで実現できる。図２．６のtesterの実行とsolverの実行は並列に行なわれるが、solverの実行の完了はtesterの実行の完了を持つ。これは変数 "Control"によって実現されている。testerは解の候補を検査し、成功するとControl はcontinueにinstantiate され、検査が失敗するとfailにinstantiate される。solverの実質的な処理は③のガード内で実行される。ただし、そのガードのすべてのゴールが完了するためには①での検査が終了しControl にcontinueがinstantiate される必要がある。もし、①での検査が失敗すれば②の節によりControl はstopにinstantiate され、solverの実行としては④の節が選ばれる。もし、③のガード内でsolverの実質的な処理が行なわれていてもControl がstopにinstantiate され、④の節のガードがコミットされることにより、③でのsolverの処理は途中で中断される。check-result述語は、solverの実行結果に従って解を報告する。

２．５　解の候補の生成方式

解の候補を生成する方式として、解の候補を生成する際に解の候補が持つすべての次元に対して候補を決める方式（一括的解の生成方式）とすべての次元ではなく、いくつ




```
①   tester(,..,Control) :-
            tester1(,..,) |
            Control = continue.
②   tester(,..,Control) :-
            otherwise |
            Control = stop.

③   solver(,..,Result ,Control) :-
            Control = continue, solver1(,..,Result1) |
            Result = Result1.
④   solver(,..,Result1,Control) :-
            Control = stop |
            Result1 = fail.
⑤   solver1(,..,Result) :-
            solver2(,..,) |
            Result = success.
⑥   solver1(,..,Result) :-
            otherwise |
            Result = fail.

⑦   check_result(success,Out,..,) :-
            Out = ,..,
⑧   check_result(fail,Out,..,) :-
            Out = [].
```

図２．６　ガード方式によるＡＮＤノードの並列実行方式

かの次元に対する候補を決める方式（段階的解の生成方式）との２つの方式が考えられる。

(1) 一括的解の生成方式

解空間のすべての次元に対する１つの候補を生成する。図２．７に示すようにgenerator は１つしか存在しない。

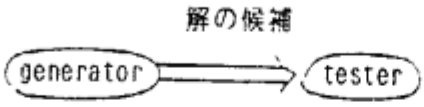


図２．７　一括的解の生成方式

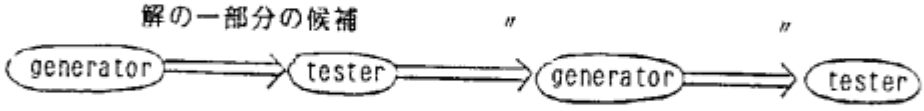


図２．８　段階的解の生成方式

(2) 段階的解の生成方式

解空間のいくつかの次元ごとに段階的に解の候補を生成し、検査する。図２．８に示すように複数のgenerator が段階的に解を生成する。段階的解の生成方式はすべての探索問題に使用できる訳ではなく、問題に依存している。

２．６　ＫＬ１による探索問題のプログラミング方式

ここでは、ＯＲノードの実現方式と並列実行方式との関係について示す。この関係をまとめると次の表のようになる。

表２．１　ＯＲノードの実現方式と並列実行方式

実現方式 / 並列性	ストリーム方式		プロセス生成方式		ガードＯＲ方式	
ＯＲ並列 ＡＮＤ並列	△	ＡＮＤ並列は は難しい。	○	メタコールによる プロセスの停止。	×	ＡＮＤ並列は できない。
ＯＲ並列 ＡＮＤ逐次	○		○		○	
ＯＲ逐次 ＡＮＤ並列	△	ＡＮＤ並列は は難しい。	○	ＯＲ並列の部分を シーケンシャルＡＮＤ で実行する。	×	ガードはＯＲ 並列で実行さ れる。
ＯＲ逐次 ＡＮＤ逐次	△	bounded bufferによる 逐次制御。	○	ＯＲ並列もＡＮＤ並列 もシーケンシャル ＡＮＤで実行する。	×	同　上

注）　○は可能，△は可能だが難しい，×は不可能

この表での実現方式はＯＲ並列の実現方式であるため、すべての実現方式でＯＲ並列ＡＮＤ逐次方式が可能である。

3．　ＫＬ１によるふく面演算のプログラミング

次に、ふく面演算を例にしてＫＬ１による探索問題のプログラミングを行う。

3．1　ふく面演算とは

ふく面演算とは下記のような式の英文字に０から９までの異なる数字を割り当てて式を成立させるパズルである。

```
   SEND
+  MORE
-------
  MONEY
```

ふく面演算を人間が解く場合は、どれか１つの英文字に対して、既に使用した数字を取り除いた０から９までの数字の１つを割り当てることを繰り返して行う。どこかで式が成立しなくなれば、適当なところに戻って数字の割り当てをやり直す。このように、ふく面演算では、英文字に数字を割り当てる際に試行錯誤は必須である。

3．2　ふく面演算のheuristics

ふく面演算にはふく面演算特有のheuristicsがいくつかある。

(1) 桁ごとに注目して不確定性の少ない順に探索すると探索空間が小さくなる［古川85］。例えば、先の例では探索の途中で次のように数字が割り当てられていると、

```
   9END
+  10RE
-------
  10NEY
carrier  10C2C3C4
```

下から３桁目の英文字に数字を割り当てる方が探索空間が小さくなる。それは、残りの桁は３つの英文字を含んでいるが、３桁目は２つの英文字しか含んでいないので、不確定さが少なくなるためである。

(2) 問題によっては、ある英文字に対してコンストレイントを割り当てることができる。例えば、先の例ではMは簡単に１と決めることができ、同時にSは9か8のどちらかとわかり、Oは0か1のどちらかとわかる。

(3) 英文字に割り当てた数字がすべて異なるという問題の条件は、数字を割り当てる際にチェックする方が割り当てた最後でチェックするより探索空間が小さい。

(4) 英文字への数字の割り当ては桁ごとに割り当てる方が桁を考慮しない割り当てに比べて探索空間が小さい。

３．３　ふく面演算のプログラミング

２章で述べたＫＬ１による探索問題のプログラミング方式のいくつかの方式でふく面演算のプログラムを作成する。ここでは、表３．１に示すような方式のプログラムを作成した。

表３．１　　ふく面演算のプログラム

項番	プログラム名	並列実行方式				実現方式				解の生成		備考
		OR並列AND並列	OR並列AND逐次	OR逐次AND並列	OR逐次AND逐次	ストリーム方式	プロセス生成方式	ガードOR方式		一括的解の生成	段階的解の生成	
1	MASK1		○			○					○	
2	MASK2		○				○				○	
3	MASK3	○					○				○	
4	MASK4	○					○				○	permutation 内にgeneratorを組み込む。
5	MASK5	*1 ○					○			○		
6	MASK6		○					○			○	
7	MASK7		○			○					○	copy述語を使用しない。
8												

*1)　ANDノードがないためORに関してのみ並列に実行する。

次に、表３．２にプログラムとプログラム・リストとの対応関係を示す。

表３．２　プログラムとプログラム・リストとの対応表

項番	プログラム名	呼び出し	メ　イ　ン	共通ルーチン
1	mask1	図３．７	図３．９～図３．１０	図３．２１
2	mask2	図３．８	図３．１１～図３．１２	〃
3	mask3	図３．８	mask2のtesterを図３．１３で置きかえる。	〃
4	mask4	図３．８	mask3のgenerator とpermutationを図３．１４で置きかえる。	〃
5	mask5	図３．８	図３．１５～図３．１６	〃
6	mask6	図３．８	図３．１７	なし
7	mask7	図３．８	図３．１８～図３．２０	図３．２１

以下、上の７つのプログラムについて簡単に説明する。

(1) プログラム　mask 1

図３．１に示すように桁ごとに英文字に対する解の候補を生成し、検査する。検

査では、その桁の式を満足した解の候補のみを上位の桁を解くsolverにストリームで渡す。桁ごとにgenerate and test を行なうものがsolverで、solverはgenerator とtesterからなり、generator は桁の中で未定の英文字とキャリアに関して解の候補として数の順列を次から次に生成し、testerはgenerator で生成された解の候補を桁の式にあてはめ、検査する。さらに、testerは桁の式を満足した解の候補のみをストリームで上位のsolverに渡す。各solverは探索木のＡＮＤノードに相当している。このプログラムはtesterで検査が完了しない限り解の候補を上位のsolverに伝えないためＡＮＤノードはＡＮＤ逐次で実行される。generator で使われているcopy述語とvar 述語の妥当性については後で検討するが、プログラムmusk 7で述べるようにこのプログラムはcopy述語やvar 述語を使わなくても実現できる。

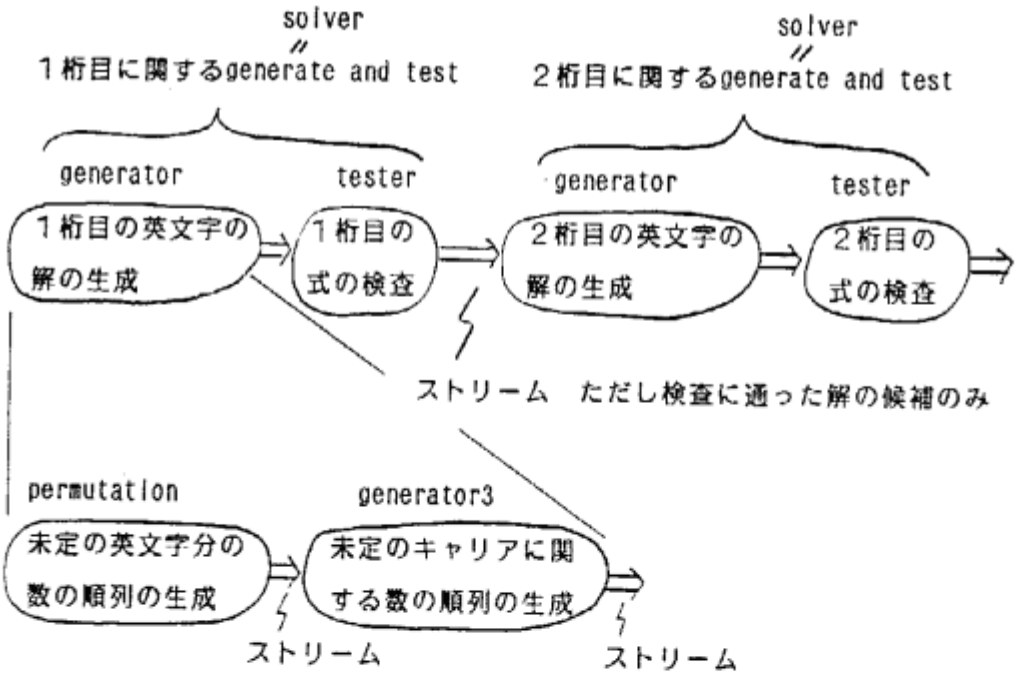


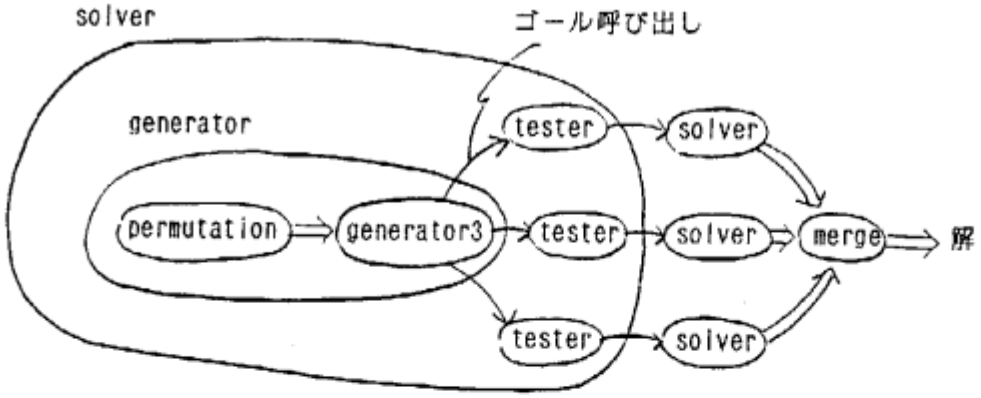
図３．１　プログラム　mask1 の概要

(2) プログラム　mask2

プログラムの構成はプログラムmask1 と同じであるが、解の候補を上位の桁のsolverにストリームで渡すのではなく、解の候補を引数として上位の桁を解くsolverを生成する方式（プログラム生成方式）である。ただし、実行方式は、検査が完了しない限り上位の桁のsolverを生成しないのでＡＮＤ逐次実行方式である。図３．２にこのプログラムの概要を示す。

(3) プログラム　mask3

プログラムmask3 はプログラムmask2 の実行方式をＯＲ並列ＡＮＤ並列実行方式に変えたものである。桁ごとに解を求めるsolverをＡＮＤ並列に実行するために、２．４で述べたＡＮＤノードの並列実行方式の１つであるメタコール方式を使う。プログラムmask2 ではtesterがＡＮＤノードのゴールを逐次的に実行していたが、このプログラムでは、このtesterを並列に実行する。すなわち、testerでの検査と次の桁の解を求めるsolverとをメタコールを使って並列に実行する。図３．３にプログラムの概要を示す。



注）generator, permutation, generator3, tester
solverは(1) の図とほぼ同じである。

図３．２　プログラム　mask2 の概要

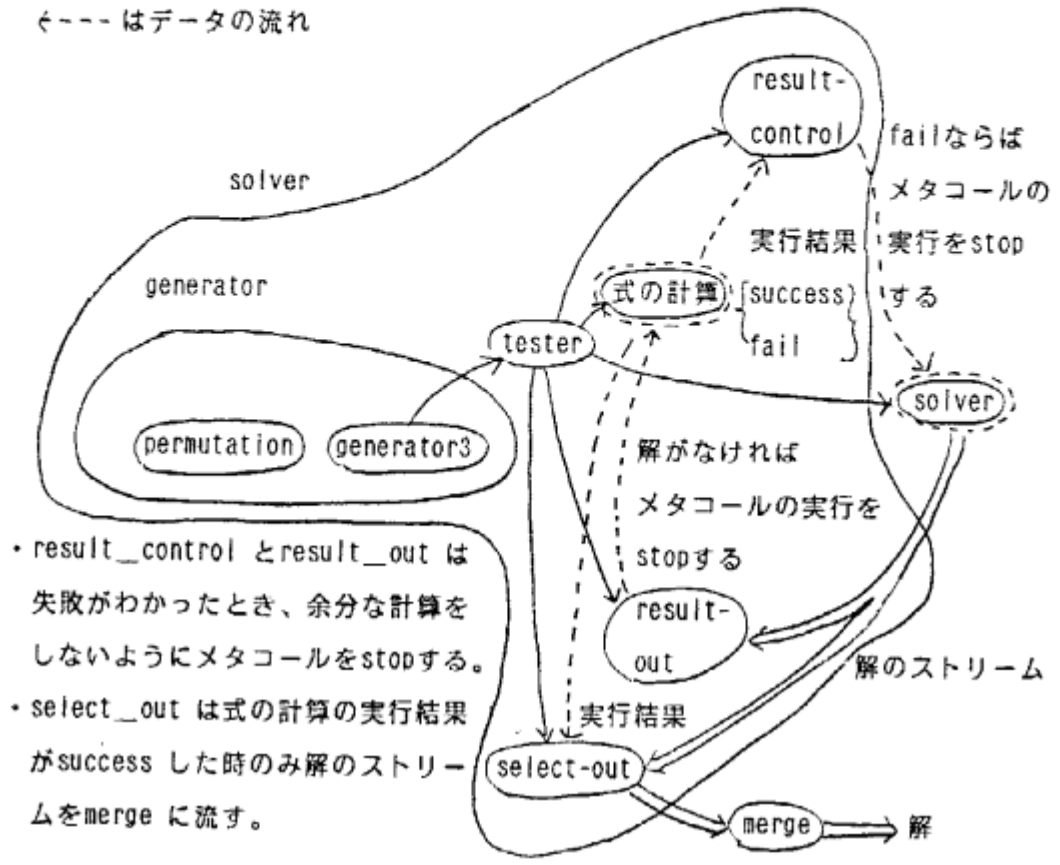
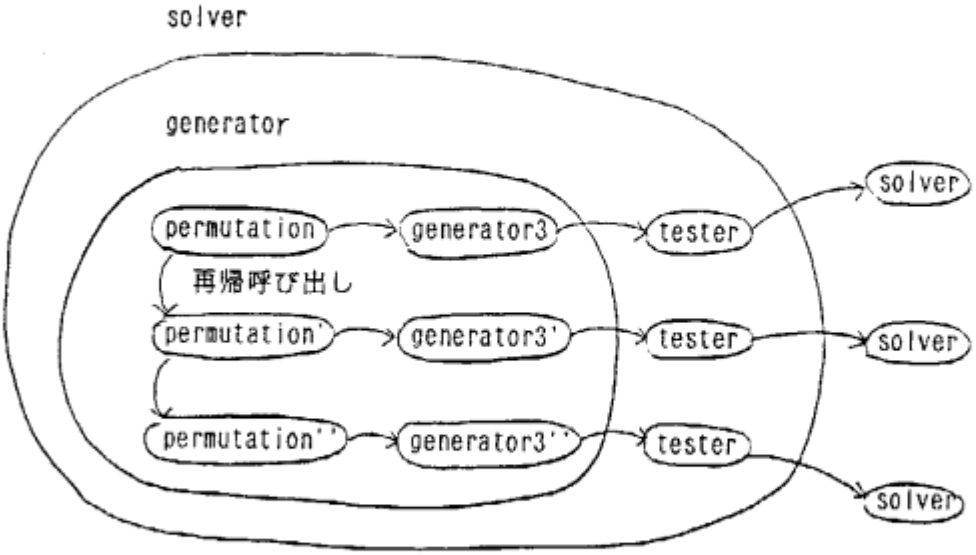


図３．３　プログラム　mask3 の概要

(4) プログラム　mask4

プログラムmask4 はプログラムmask3 のpermutation とgenerator3の間をストリーム通信ではなく、ゴール呼び出しにしたものである。プログラムmask3 ではpermutation が生成した解の候補はmerge で集められストリームとしてgenerator3に渡されていたが、プログラムmask4 ではpermutation が解の候補を１つづつ生成する度にgenerator3を呼び出す。プログラムmask3 とプログラムmask4 の違いはpermutation とgenerator3の間にmerge が入っているか入っていないかである。２つのプロセス間のデータ通信をmerge で行なうかゴール呼び出しで行なうかは、サイクル数から見ると基本的に大きな差はない（付録３を参照）。プログラムmask3 とプログラムmask4 とのサイクル数はそれぞれ41と36であり、大きな差はなかった。



今まではpermutation とgenerator3とのストリーム通信は次のように行っていた。

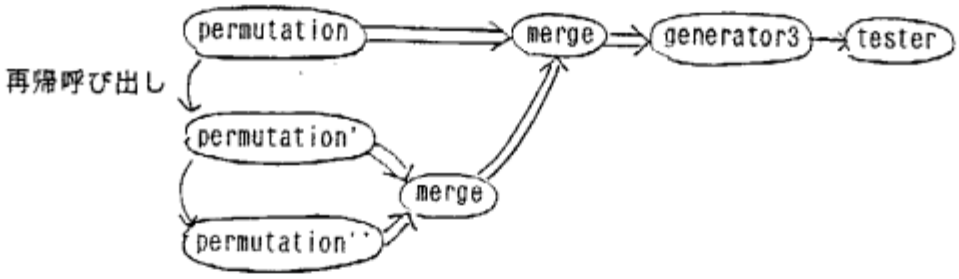


図３．４　プログラム　mask4 の概要

しかし、プログラムを書く上でプログラムが複雑になると、merge かゴール呼び出しかの差は大きな差になると思われる。本プログラムの経験では、merge を使う方がプログラムは解りやすくなると思われる。図３．４にプログラムの概要を示す。

(5) プログラム　mask5

プログラムmask5 は、今までのプログラムが段階的解の生成方式であったのに対して、解の生成方式が一括的解の生成方式である。解を一括的に生成する方式は、

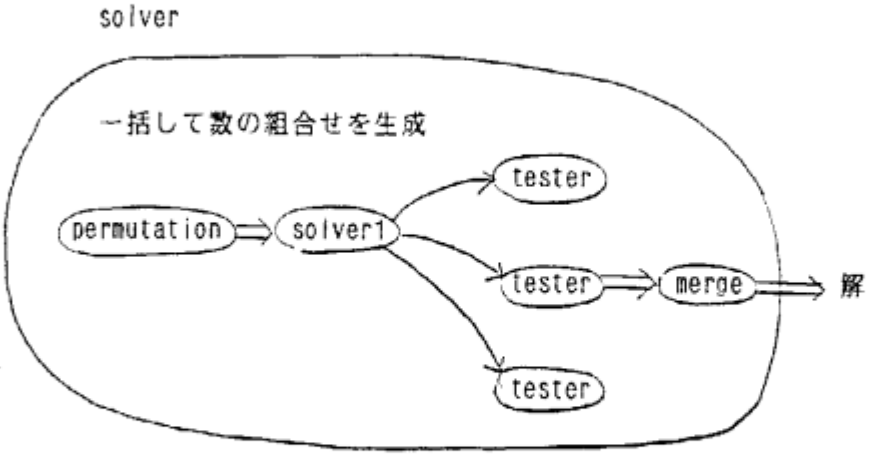


図３．５　プログラム　mask5 の概要

段階的に解を生成する方式に比べてプログラムは簡単であるが、無駄な計算を行なうため効率が悪いという欠点がある。欠点とは、すべての英文字に対してのすべての解の候補を生成するため、ある桁の英文字にとってその桁の式を決して満たさない同じ解の候補を何回も計算することである。図３．５にプログラムの概要を示す。図から明らかなように、プログラムのなかでsolverは１つしかなく、generate and test を１回実行するのみである。

(6) プログラム　mask6

プログラムmask6 はＯＲノードの並列実行をガードＯＲ方式で実現している。ガードＯＲ方式とは、２．３で述べたようにＯＲノードをガードが持つ非決定性を利用した実現方式である。ガードを使っているため、解は１個しか得ることができないが、解を集めるなどの操作が不要なため、プログラムが他の実現方式に比べて簡単になる。図３．６にプログラムの概要を示す。図からわかるように、ある桁の英文字の数を選ぶこととその英文字に対して残りの数の組から数を選ぶことが同じレベルのガードとして計算される。

(7) プログラム　mask7

プログラムmask7 はプログラムmask2 をcopy述語を使わないように書き直したも

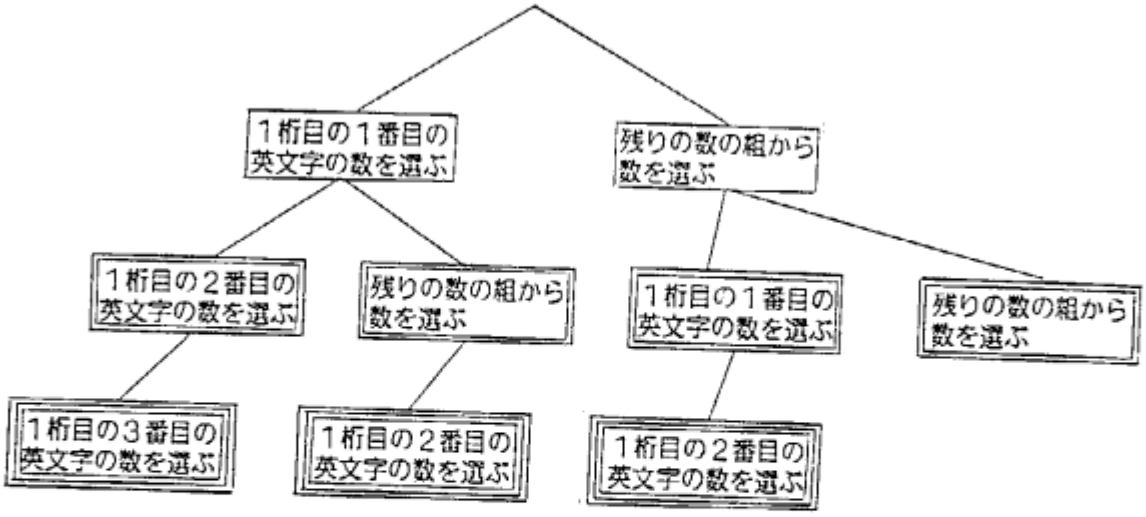


図3．6　プログラム　mask6 の概要

のである。解の候補は英文字名と値とが対となったassociated list の形で保持する。英文字名に相当する変数へ値を代入すると、更新された新しいassociated list が作られる。以降の計算ではその更新されたassociated list を使用する。

```
%               B A
%             + B A
%             -----------------
%               C B              A=1,B=2,C=4
mask :-
        Exp=[add_with_carrier(0,[A,A,B],C1),
             add_with_carrier(C1,[B,B,C],0)],
        Chars = [A,B,C],
        mask(2,Exp,Chars,Out),
        write_stream_result(Out).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mask(N,[E|Rest],Chars,Out) :-
        Nums=[1,2,3,4],
        solver([N+E+Rest+Chars+Nums],Out).
```

図３．７　呼び出し１

```
%               B A
%             + B A
%             -----------------
%               C B              A=1,B=2,C=4
mask :-
        Exp=[add_with_carrier(0,[A,A,B],C1),
             add_with_carrier(C1,[B,B,C],0)],
        Chars = [A,B,C],
        mask(2,Exp,Chars,Out),
        write_stream_result(Out).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mask(N,[E|Rest],Chars,Out) :-
        Nums=[1,2,3,4],
        solver(N+E+Rest+Chars+Nums,Out).
```

図３．８　呼び出し２

```
%%%%% solver %%%%%

solver([],Out) :-
        Out=[].
solver([0+E+Rest+Chars+Nums|Input],Out) :-
        Out=[Chars|Out1],
        solver(Input,Out1).
solver([N+E+Rest+Chars+Nums|Input],Out) :-
        N \= 0 |
        generator([N+E+Rest+Chars+Nums|Input],Channel),
        tester(Channel,Out1),
        solver(Out1,Out).

%%%%% generator %%%%%

generator([],Channel) :-
        Channel = [].
generator([N+E+Rest+Chars+Nums|Input],Channel) :-
        count_vars(E,Varsnum,Vars),
        generator1(Varsnum,Vars,N+E+Rest+Chars+Nums,Channel,Channel1),
        generator(Input,Channel1).

generator1(0,_,N+E+Rest+Chars+Nums,Channel,New_Channel) :-
        generator4([],[],Nums,Channel,New_Channel,N+E+Rest+Chars).
generator1(Varnum,Vars,N+E+Rest+Chars+Nums,Channel,New_Channel) :-
        Varnum \= 0 |
        permutation(Varnum,Nums,Permlist),
        generator2(Permlist,Vars,N+E+Rest+Chars,Channel,New_Channel).

generator2([Permvars+Permrest|Permlist],Vars,N+E+Rest+Chars,
                                Channel,New_Channel) :-
        generator3(Permvars,Permrest,Permlist,Vars,Channel,New_Channel,
                N+E+Rest+Chars).
generator2([],_,_,Channel,New_Channel) :-
        New_Channel = Channel.

generator3(Permvars,Permrest,Permlist,Vars,Channel,New_Channel,
                N+E+Rest+Chars) :-
        generator4(Permvars,Vars,Permrest,Channel,Channel1,N+E+Rest+Chars),
        generator2(Permlist,Vars,N+E+Rest+Chars,Channel1,New_Channel).
```

図３．９　mask1 のプログラム・リスト(1)

```
generator4(Permvars,Vars,Permrest,Channel,New_Channel,N+E+Rest+Chars) :-
        E = add_with_carrier(C1,_,C2) &
        (var(C1), var(C2)) &
        (generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,0,0,S1),
        generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,0,1,S2),
        generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,1,0,S3),
        generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,1,1,S4)) |
        Channel=[S1,S2,S3,S4 | New_Channel].
generator4(Permvars,Vars,Permrest,Channel,New_Channel,N+E+Rest+Chars) :-
        E = add_with_carrier(C1,_,C2) &
        (var(C1), nonvar(C2)) &
        (generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,0,C2,S1),
        generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,1,C2,S2)) |
        Channel=[S1,S2| New_Channel].
generator4(Permvars,Vars,Permrest,Channel,New_Channel,N+E+Rest+Chars) :-
        E = add_with_carrier(C1,_,C2) &
        (nonvar(C1), var(C2)) &
        (generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,0,S1),
        generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,1,S2))|
        Channel=[S1,S2| New_Channel].
generator4(Permvars,Vars,Permrest,Channel,New_Channel,N+E+Rest+Chars) :-
        E = add_with_carrier(C1,_,C2) &
        (nonvar(C1), nonvar(C2)) &
        (generator5(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,C2,S1)) |
        Channel=[S1| New_Channel].
generator4(_,_,_,Channel,New_Channel,_) :-
        otherwise |
        Channel=New_Channel.

generator5(Permvars,Vars,Permrest,
        N+add_with_carrier(C1,Exp,C2)+Rest+Chars,C11,C21,S) :-
        prolog(copy(Vars+Exp+C1+C2+Rest+Chars,
                    Vars1+Exp1+C11+C21+Rest1+Chars1)) &
        Vars1 = Permvars |
        S = N+add_with_carrier(C11,Exp1,C21)+Rest1+Chars1+Permrest.

***** tester *****

tester([],Out) :-
        Out = [].
tester([N+E+Rest+Chars+Nums|Channel],Out) :-
        call(E,success,_), N1 := N - 1 |
        separete(Rest,E1,Rest1),
        Out = [N1+E1+Rest1+Chars+Nums|Out1],
        tester(Channel,Out1).
tester([N+E+Rest+Chars+Nums|Channel],Out) :-
        otherwise | tester(Channel,Out).

separete([],_,Rest1) :- Rest1 = [].
separete([A|B],E1,Rest1) :- E1 = A, Rest1 = B.
```

図3．10 mask1 のプログラム・リスト(2)

```
%%%%% solver %%%%

solver(0+E+Rest+Chars+Nums,Out) :-
        Out=[Chars].
solver(N+E+Rest+Chars+Nums,Out) :-
        N \= 0 |
        generator(N+E+Rest+Chars+Nums,Out).

%%%%% generator %%%%%

generator(N+E+Rest+Chars+Nums,Out) :-
        count_vars(E,Varsnum,Vars),
        generator1(Varsnum,Vars,N+E+Rest+Chars+Nums,Out).

generator1(0,_,N+E+Rest+Chars+Nums,Out) :-
        generator3([],[],Nums,N+E+Rest+Chars,Out).
generator1(Varnum,Vars,N+E+Rest+Chars+Nums,Out) :-
        Varnum \= 0 |
        permutation(Varnum,Nums,Permlist),
        generator2(Permlist,Vars,N+E+Rest+Chars,Out).

generator2([Permvars+Permrest|Permlist],Vars,N+E+Rest+Chars,Out) :-
        generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out1),
        generator2(Permlist,Vars,N+E+Rest+Chars,Out2),
        merge(Out1,Out2,Out).
generator2([],_,_,Out) :- Out = [].

generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2) &
        (var(C1), var(C2)) |
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,0,0,Out1),
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,0,1,Out2),
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,1,0,Out3),
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,1,1,Out4),
        merge(Out1,Out2,Out3,Out4,Out).
generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2) &
        (var(C1), nonvar(C2)) |
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,0,C2,Out1),
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,1,C2,Out2),
        merge(Out1,Out2,Out).
generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2) &
        (nonvar(C1), var(C2)) |
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,0,Out1),
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,1,Out2),
        merge(Out1,Out2,Out).
generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2) &
        (nonvar(C1), nonvar(C2)) |
        tester(Permvars,Vars,Permrest,N+E+Rest+Chars,C1,C2,Out).
```

図3．11 mask2 のプログラム・リスト(1)

```
%%%%% tester %%%%%

tester(Permvars,Vars,Permrest,
        N+add_with_carrier(C1,Exp,C2)+Rest+Chars,C11,C21,Out) :-
        prolog(copy(Vars+Exp+C1+C2+Rest+Chars,
                    Vars1+Exp1+C11+C21+Rest1+Chars1)),
        Vars1 = Permvars,
        call(add_with_carrier(C11,Exp1,C21),success,_) |
        N1 := N - 1,
        separete(Rest1,E2,Rest2),
        solver(N1+E2+Rest2+Chars1+Permrest,Out).
tester(Permvars,Vars,Permrest,
        N+add_with_carrier(C1,Exp,C2)+Rest+Chars,C11,C21,Out) :-
        otherwise | Out = [].

separete([],_,Rest1) :- Rest1 = [].
separete([A|B],E1,Rest1) :- E1 = A, Rest1 = B.
```

図3．12 mask2 のプログラム・リスト(2)

```
%%%%% tester %%%%%

tester(Permvars,Vars,Permrest,
        N+add_with_carrier(C1,Exp,C2)+Rest+Chars,C11,C21,Out) :-
        prolog(copy(Vars+Exp+C1+C2+Rest+Chars,
                    Vars1+Exp1+C11+C21+Rest1+Chars1)),
        Vars1 = Permvars |
        call(add_with_carrier(C11,Exp1,C21),Result,Control),
        N1 := N - 1,
        result_control(Result,Control1),
        separete(Rest1,E2,Rest2),
        call(solver(N1+E2+Rest2+Chars1+Permrest,Out1),_,Control1),
        result_out(Control1,Out1,Control),
        select_out(Result,Out,Out1).

tester(Permvars,Vars,Permrest,
        N+add_with_carrier(C1,Exp,C2)+Rest+Chars,C11,C21,Out) :-
        otherwise | Out = [].

separete([],_,Rest1) :- Rest1 = [].
separete([A|B],E1,Rest1) :- E1 = A, Rest1 = B.

result_control(fail,Control) :- Control = stop.
result_control(success,_).
result_control(stopped,_).

result_out(stop,_,_).
result_out(_,[],Control) :- Control = stop.
result_out(_,Out1,_) :-
        Out1 \= [] | true.

select_out(fail,Out,_) :- Out = [].
select_out(success,Out,Out1) :- Out = Out1.
```

図3．13 mask3 のプログラム・リスト

```
%%%%% solver %%%%%

solver(0+E+Rest+Chars+Nums,Out) :-
        Out=[Chars].
solver(N+E+Rest+Chars+Nums,Out) :-
        N \= 0 |
        generator(N+E+Rest+Chars+Nums,Out).

%%%%% generator %%%%%

generator(N+E+Rest+Chars+Nums,Out) :-
        count_vars(E,Varsnum,Vars),
        generator1(Varsnum,Vars,N+E+Rest+Chars+Nums,Out).

generator1(0,_,N+E+Rest+Chars+Nums,Out) :-
        generator3([],[],Nums,N+E+Rest+Chars,Out).

generator1(Varnum,Vars,N+E+Rest+Chars+Nums,Out) :-
        Varnum \= 0 |
        permutation(Varnum,Nums,Vars,N+E+Rest+Chars,Out).

generator2([Permvars+Permrest],Vars,N+E+Rest+Chars,Out) :-
        generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out).
```

以下generator3は図３．１２と同じ

```
%%%%% permutation %%%%%

permutation(Num,Numlist,Vars,N_Data,Out) :-
        permutation1(Num+Numlist+[],Vars,N_Data,Out).

permutation1(0+Numlist+List,Vars,N_Data,Out) :-
        generator2([List+Numlist],Vars,N_Data,Out).
permutation1(Num+Numlist+List,Vars,N_Data,Out) :-
        Num \= 0 |
        N1 := Num - 1,
        permutation2(N1,Numlist,[],List,Vars,N_Data,Out).
permutation2(_,[],_,_,_,_,Out) :-
        Out= [].
permutation2(N1,[X|Numlist],Rest,List,Vars,N_Data,Out) :-
        ghcap(Rest,Numlist,Numlist1),
        permutation1(N1+Numlist1+[X|List],Vars,N_Data,Out1),
        permutation2(N1,Numlist,[X|Rest],List,Vars,N_Data,Out2),
        merge(Out1,Out2,Out).
```

図３．１４　mask4 のプログラム・リスト

```
mask(N,Exp,Chars,Out) :-
        Nums=[1,2,3,4],
        solver(Exp,Chars,Nums,Out).

%%%%% solver %%%%%

solver(Exp,Chars,Nums,Out) :-
        count_vars(Chars,Varsnum,Vars),
        permutation(Varsnum,Nums,Permlist),
        solver1(Permlist,Vars,Exp,Chars,Out).

solver1([],_,_,_,Out) :- Out = [].
solver1([Permvars+_|Permlist],Vars,Exp,Chars,Out) :-
        prolog(copy(Vars+Exp+Chars,Vars1+Exp1+Chars1)),
        Vars1 = Permvars |
        tester(Exp1,Chars1,Out1),
        solver1(Permlist,Vars,Exp,Chars,Out2),
        merge(Out1,Out2,Out).

%%%%% tester %%%%%

tester(Exp,Chars,Out) :-
        tester1(Exp,Result),
        out_control(Result,Chars,Out).
out_control(success,Chars,Out) :- Out = [Chars].
out_control(fail,_,Out) :- Out = [].

tester1(Exp,Result) :-
        call(tester2(Exp),Result,_).

tester2([]).
tester2([E|Exp]) :-
        call(E,success,_),
        tester2(Exp).
```

図3．15 mask5 のプログラム・リスト(1)

```
%%%%% add_with_carrier %%%%%

add_with_carrier(C,[A|Exp],C1) :-
        Exp \= [] | add_with_carrier1(C,A,Exp,C1).
add_with_carrier(C,[A],C1) :-
        add_with_carrier1(C,0,A,C1).
add_with_carrier1(C,N,[A],C1) :-
        check_carrier(C,N,A,C1).
add_with_carrier1(C,N,[E|Exp],C1) :-
        Exp \= [] |
        N1 := N + E,
        add_with_carrier1(C,N1,Exp,C1).
check_carrier(C,N,A,C1) :-
        mod(N,10,M), A = M |
        C = 0, check_carrier1(N,C1).
check_carrier(C,N,A,C1) :-
        mod(N,10,M), A := M + 1, N1 := N + 1 |
        C = 1, check_carrier1(N1,C1).
check_carrier(C,N,A,C1) :-
        mod(N,10,M), A := M + 2, N1 := N + 2 |
        C = 2, check_carrier1(N1,C1).
check_carrier1(N,C1) :-
        N < 10 | C1 = 0.
check_carrier1(N,C1) :-
        20 > N, N >= 10 | C1 = 1.
check_carrier1(N,C1) :-
        30 > N, N >= 20 | C1 = 2.
```

図3．16　mask5 のプログラム・リスト(2)

```
mask :- rel([[('A',A),('A',A),('B',B)],
             [('B',B),('B',B),('C',C)]],
                0,0,[1,2,3,4],[],[]) |
                write([A,B,C]).

%%%%% rel %%%%%

rel([[(C,X)|N]|Exp],Sum,F1,[XX|List],Rest,Data) :-
        N \= [], var(X),
        prolog(copy((X,N,Exp),(XX,N1,Exp1))),
        Sum1 := Sum + XX ,
        ghcap(Rest,List,AllRest),
        rel([N1|Exp1],Sum1,F1,AllRest,[],[(C,XX)|Data])
        | X=XX, N=N1, Exp=Exp1.
rel([[(C,X)|N]|Exp],Sum,F1,[S|List],Rest,Data) :-
        N \= [], var(X),
        rel([[(C,X)|N]|Exp],Sum,F1,List,[S|Rest],Data)
        | true.
rel([[(C,X)|N]|Exp],Sum,F1,List,_,Data) :-
        N \= [], nonvar(X),
        Sum1 := Sum + X,
        rel([N|Exp],Sum1,F1,List,[],Data)
        | true.

rel([[(C,X)|N]|Exp],Sum,F1,[XX|List],Rest,Data) :-
        N = [], var(X),
        Sum1 := Sum + F1,
        one_digit(Sum1,Sum2,F12),
        XX = Sum2,
        prolog(copy((X,Exp),(XX,Exp1))),
        ghcap(Rest,List,AllRest),
        rel(Exp1,0,F12,AllRest,[],[(C,XX)|Data])
        | X=XX, Exp=Exp1.
rel([[(C,X)|N]|Exp],Sum,F1,[S|List],Rest,Data) :-
        N = [], var(X),
        rel([[(C,X)|N]|Exp],Sum,F1,List,[S|Rest],Data)
        | true.
rel([[(C,X)|N]|Exp],Sum,F1,List,_,Data) :-
        N = [], nonvar(X), Sum1 := Sum + F1,
        one_digit(Sum1,Sum2,F12),
        X = Sum2,
        rel(Exp,0,F12,List,[],Data)
        | true.

rel([],_,_,_,_,_).

%%%%% one_digit %%%%

one_digit(S,S1,C) :-
        S < 10 | S1=S, C=0.
one_digit(S,S1,C) :-
        10 =< S, S < 20 | S1 := S - 10, C=1.
one_digit(S,S1,C) :-
        20 =< S, S < 30 | S1 := S - 20, C=2.
```

図3．17　mask6 のプログラム・リスト

```
mask1 :-
        Chars = [('A','VAR'),('B','VAR'),('C','VAR'),('C1','VAR')],
        Nums=[1,2,4],
        Exp = [add_with_carrier(0,['A','A','B'],'C1'),
               add_with_carrier('C1',['B','B','C'],0)],
        Exp = [E|Rest],
        solver(2+E+Rest+Chars+Nums,Out),
        write_stream_result(Out).

%%%%% solver %%%%%

solver(0+E+Rest+Chars+Nums,Out) :-
        Out=[Chars].
solver(N+E+Rest+Chars+Nums,Out) :-
        N \= 0 |
        generator(N+E+Rest+Chars+Nums,Out).

%%%%% generator %%%%%

generator(N+E+Rest+Chars+Nums,Out) :-
        count_vars(E,Chars,Varsnum,Vars),
        generator1(Varsnum,Vars,N+E+Rest+Chars+Nums,Out).

generator1(0,_,N+E+Rest+Chars+Nums,Out) :-
        generator4(Nums,N+E+Rest+Chars,Out).
generator1(Varnum,Vars,N+E+Rest+Chars+Nums,Out) :-
        Varnum \= 0 |
        permutation(Varnum,Nums,Permlist),
        generator2(Permlist,Vars,N+E+Rest+Chars,Out).

generator2([Permvars+Permrest|Permlist],Vars,N+E+Rest+Chars,Out) :-
        generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out1),
        generator2(Permlist,Vars,N+E+Rest+Chars,Out2),
        merge(Out1,Out2,Out).
generator2([],_,_,Out) :- Out = [].

generator3(Permvars,Vars,Permrest,N+E+Rest+Chars,Out) :-
        list_put_dict(Vars,Permvars,Chars,Chars1),
        generator4(Permrest,N+E+Rest+Chars1,Out).

generator4(Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2),
        var_dict(C1,Chars), var_dict(C2,Chars) |
        generator5(Permrest,N+E+Rest+Chars,[C1,C2],[0,0],Out1),
        generator5(Permrest,N+E+Rest+Chars,[C1,C2],[0,1],Out2),
        generator5(Permrest,N+E+Rest+Chars,[C1,C2],[1,0],Out3),
        generator5(Permrest,N+E+Rest+Chars,[C1,C2],[1,1],Out4),
        merge(Out1,Out2,Out3,Out4,Out).
```

図3．18　mask7 のプログラム・リスト(1)

```
generator4(Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2),
        var_dict(C1,Chars), nonvar_dict(C2,Chars) |
        generator5(Permrest,N+E+Rest+Chars,[C1],[0],Out1),
        generator5(Permrest,N+E+Rest+Chars,[C1],[1],Out2),
        merge(Out1,Out2,Out).
generator4(Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2),
        nonvar_dict(C1,Chars), var_dict(C2,Chars) |
        generator5(Permrest,N+E+Rest+Chars,[C2],[0],Out1),
        generator5(Permrest,N+E+Rest+Chars,[C2],[1],Out2),
        merge(Out1,Out2,Out).
generator4(Permrest,N+E+Rest+Chars,Out) :-
        E = add_with_carrier(C1,_,C2),
        nonvar_dict(C1,Chars), nonvar_dict(C2,Chars) |
        generator5(Permrest,N+E+Rest+Chars,[],[],Out).

generator5(Permrest,N+E+Rest+Chars,Vars,Values,Out) :-
        list_put_dict(Vars,Values,Chars,Chars1) &
        write('generator complete') &
        write(N+E+Rest+Chars1+Permrest),
        tester(N+E+Rest+Chars1+Permrest,Out).

%%%%% tester %%%%%

tester(N+add_with_carrier(C1,Exp,C2)+Rest+Chars+Nums,Out) :-
        add_with_carrier(C1,Exp,C2,Chars),
        N1 := N - 1 |
        write('testor complete') &
        write(N+add_with_carrier(C1,Exp,C2)+Rest+Chars+Nums),
        separete(Rest,E1,Rest1),
        solver(N1+E1+Rest1+Chars+Nums,Out).
tester(N+E+Rest+Chars+Nums,Out) :-
        otherwise | Out = [].

separete([],_,Rest1) :- Rest1 = [].
separete([A|B],E1,Rest1) :- E1 = A, Rest1 = B.

%%%%% add_with_carrier %%%%%

add_with_carrier(C,[A|Exp],C1,Chars) :-
        Exp \= [] |
        get_dict(C,Cv,Chars), get_dict(A,Av,Chars),
        N := Cv + Av, add_with_carrier(N,Exp,C1,Chars).
add_with_carrier(N,[A],C1,Chars) :-
        N < 10 |
        get_dict(A,Av,Chars), get_dict(C1,C1v,Chars),
        C1v = 0, Av = N.
```

図3．19　mask7 のプログラム・リスト(2)

```
add_with_carrier(N,[A],C1,Chars) :-
        N >= 10 |
        get_dict(A,Av,Chars), get_dict(C1,C1v,Chars),
        C1v = 1,
        Av := N - 10.
add_with_carrier(N,[A],C1,Chars) :-
        N >= 20 |
        get_dict(A,Av,Chars), get_dict(C1,C1v,Chars),
        C1v = 2,
        Av := N - 20.

%%%% var_dict %%%%
var_dict(Var,[(Var,'VAR')|_]).
var_dict(Var,[(Var1,_)|Dict]) :-
        Var \= Var1 |
        var_dict(Var,Dict).

%%%% nonvar_dict %%%%
nonvar_dict(Var,[]).
nonvar_dict(Var,[(Var,S)|_]) :- S \= 'VAR'.
nonvar_dict(Var,[(Var1,_)|Dict]) :-
        Var \= Var1 |
        nonvar_dict(Var,Dict).

%%%% get_dict %%%%
get_dict(X,Y,[]) :- Y = X.
get_dict(Name,Value1,[(Name,Value)|Dict]) :-
        Value1 = Value.
get_dict(Name,Value,[(Name1,_)|Dict]) :-
        Name \= Name1 |
        get_dict(Name,Value,Dict).

%%%% put_dict %%%%
put_dict(Name,Value,[(Name,'VAR')|Dict],New_Dict) :-
        New_Dict = [(Name,Value)|Dict].
put_dict(Name,Value,[(Name,Value)|Dict],New_Dict) :-
        New_Dict = [(Name,Value)|Dict].
put_dict(Name,Value,[(Name1,Value1)|Dict],New_Dict) :-
        Name \= Name1 |
        New_Dict = [(Name1,Value1)|Dict1],
        put_dict(Name,Value,Dict,Dict1).

%%%% list_put_dict %%%%
list_put_dict([],[],Chars,Chars1) :-
        Chars1 = Chars.
list_put_dict([Name|Namelist],[Value|Valuelist],Chars,Chars1) :-
        put_dict(Name,Value,Chars,Chars0),
        list_put_dict(Namelist,Valuelist,Chars0,Chars1).
```

図3．20 mask7 のプログラム・リスト(3)

```
%%%%% add_with_carrier %%%%%

add_with_carrier(C,[A|Exp],C1) :-
        Exp \= [] | N := C + A, add_with_carrier(N,Exp,C1).
add_with_carrier(N,[A],0) :-
        N < 10 |  A = N.
add_with_carrier(N,[A],1) :-
        N >= 10 | A := N - 10.
add_with_carrier(N,[A],2) :-
        N >= 20 | A := N - 20.

%%%%% permutation %%%%%

permutation(Num,Numlist,Return) :-
        permutation1(Num+Numlist+[],Return).

permutation1(0+Numlist+List,Return) :-
        Return = [List+Numlist].
permutation1(Num+Numlist+List,Return) :-
        Num \= 0 |
        N1 := Num - 1,
        permutation2(N1,Numlist,[],List,Return).
permutation2(N1,[],Rest,List,Return) :-
        Return = [].
permutation2(N1,[X|Numlist],Rest,List,Return) :-
        ghcap(Rest,Numlist,Numlist1),
        permutation1(N1+Numlist1+[X|List],Return1),
        permutation2(N1,Numlist,[X|Rest],List,Return2),
        merge(Return1,Return2,Return).
```

図3．21　add _with_carrier とpermutation のプログラム・リスト

３．４　ふく面演算のプログラミングのまとめ

２章で述べたプログラムの方式に従い、いくつかの方式でふく面演算のプログラムを作成した。以下、いくつかの項目について考察する。

(1) 記述性

作成したどのプログラムもPrologによるプログラム（付録１参照）に比べてステップ数が多く、かつ手続き的に記述する必要があった。一方、Prologによるプログラムの多くの部分は宣言的な記述となっており、書き易く読み易い。ＫＬ１で大規模なアプリケーションを容易に記述するには、モジュールの導入など以外にステートメント・レベルの記述性の向上を図る必要があると思われる。

(2) 並列性／reduction 数

各プログラムの実行結果を表３．３に示す。プログラムmask7 は数字の組を[1,2,3,4] ではなく[1,2,4] にして実行した結果である（メモリ不足のため）。

並列性（およその並列性であり、正確な並列性ではない。）に注目すると、ＯＲ並列ＡＮＤ並列実行方式のプログラムの方（mask3 ～ 5）が明らかに並列性が高い。これは、ＡＮＤノードを並列に実行したため並列性が上がったからである。並列性を向上させるためにはＯＲノード以外にＡＮＤノードも並列に実行することは有効であると思われる。

表３．３　プログラムの実行結果

program	reduction	suspension	cycles	parallelism	time
mask1	239	187	46	5.2	6223
mask2	248	154	36	6.9	11034
mask3	508	515	41	12.4	22189
mask4	446	498	36	12.4	19909
mask5	655	484	47	13.9	22192
mask6					
mask7	283	303	51	5.5	

reducation数に注目すると、プログラムmask2 とプログラムmask3 のreducationの数が大きく異なる理由は、プログラムmask2 では式の検査がガード内で行なわれているため、そのreducationがreducation数に反映されていないためである。また一括的解の生成の方式でreducation数が大きい理由は、無駄な解の候補を生成し計算しているためである。

(3) copy述語やvar 述語の使用

プログラムmask7 を除きプログラムを作成する際には付録２に示す方法に従ってcopy述語やvar 述語を使ったが、プログラムmask7 ではそれらの述語を使用しない方針でプログラムを作成した。プログラムmask7 以外のプログラムでもプログラムmask7 と同様にcopy述語やvar 述語を使わずにプログラムを作成することが可能である。プログラムmask7 の方式ではできないcopy述語がどうしても必要となる場合は、解の候補の中に変数が含まれている場合である。

(4) heuristicsの導入の可能性

どのプログラムもheuristicsは導入することが可能である。しかし、ＯＲ並列の実現方式としてガードＯＲ方式はＯＲ並列をガードで実現しているため、ＯＲ並列に関する実行制御ができず、ガードＯＲ方式ではＯＲ並列の実行に関するheuristicsの導入は難しいと思う。

４．今後の課題

検索問題の大規模なアプリケーションを開発する言語としてはＫＬ１では記述性が低いので、記述性を高める工夫が必要である。その１つの解はＯＲ並列の実行方式としてpure Prolog からＯＲ並列実行方式のＫＬ１プログラムへのコンパイラであると思う。しかし、ＡＮＤノードも並列に実行するＫＬ１プログラムを生成するコンパイラを作ることは難しいと思われる。これは、今後、検討する必要がある。

本検討では、個々の問題に対するＫＬ１の並列性を生かすアルゴリズムについては何ら検討していない。個々の問題には、並列性を向上させる多くのheuristicsもあると思われる。そのような検討も今後必要であろう。特に、評価関数を使った検索方式（αβ法やＡ

アルゴリズムなど）は基本アルゴリズムであるので、その並列化には充分な検討が必要と思われる。

５．参考文献

[Nilsson 80] Nils J.Nilsson: Principles of Artificial Intelligence, Springer-Verlag, 1980

[Kornfeld 81] William A. Kornfeld:The Use of Parallelism to Implement a Heuristic Search, MIT A.I. Memo No.627, 1981

[Ueda 85] Ueda: Guarded Horn Clauses, The Logic Programming Conference, 225-236(1985)

[Mukai85] Mukai: Unification over Complex Indeterminates, The Logic Programming Conference,271-276(1985)

［太細８４］ 太細、他： Ｐｒｏｌｏｇ入門、啓学出版

［古川８５］ 古川： コメント

［ＩＣＯＴ８５］ ＩＣＯＴ ＫＬ１グループ：核言語第１版説明資料、１９８５

付録１． Prolog及びＣＩＬによるふく面演算のプログラム

Prologによるふく面演算のプログラムとＣＩＬ[Mukai 85]によるふく面演算のプログラムをそれぞれ図付録１．１と図付録１．２に示す。

Prologによるふく面演算のプログラムはmask1 とmask2 の２種類のプログラム［太細８４］がある。どちらも桁単位で英文字に数字を割り当てて式の検査を行なっている。この２つのプログラムの違いは、英文字に割り当てる数字がすべてお互いに異なっていなくてはならないという条件をすべての桁の検査を終わって行なうか（mask1 ）、英文字に変数を割り当てる度にその条件の検査を行なうか（mask2 ）の違いである。当然のことであるが、mask2 のプログラムの方が効率が良い。

ＣＩＬによるふく面演算のプログラムは先の条件をコンストレイントとして記述している。英文字に数字が割り当てられる度にコンストレイントが起動され条件をチェックする。

```
mask1(S,E,N,D,M,O,R,Y) :-
        relation(D,E,O,Y,F1),
        relation(N,R,F1,E,F2),
        relation(E,O,F2,N,F3),
        relation(S,M,F3,O,F4),
        relation(O,O,F4,M,O),
        notequal([S,E,N,D,M,O,R,Y]).

relation(X,Y,F1,Z,F2) :-
        int(X), int(Y), int(Z), carrier(F1), carrier(F2),
        add(X,Y,F1,Z,F2).
add(X,Y,F1,Z,F2) :-
        Z1 is X + Y + F1, add1(Z1,Z,F2).
add1(Z,Z,0) :-  < 10, !.
add1(Z1,Z,1) :- Z is Z1 - 10.
notequal([]).
notequal([X|Y]) :- \+(member(X,Y)), notequal(Y).

int(0).
int(1).
int(2).
int(3).
int(4).
int(5).
int(6).
int(7).
int(8).
int(9).
carrier(0).
carrier(1).

mask2(S,E,N,D,M,O,R,Y) :-
        relation(D,E,O,Y,F1,[0,1,2,3,4,5,6,7,8,9],L2),
        relation(N,R,F1,E,F2,L2,L3),
        relation(E,O,F2,N,F3,L3,L4),
        relation(S,M,F3,O,F4,L4,L5),
        relation(O,O,F4,M,O,L5,_).

relation(X,Y,F1,Z,F2,L1,L4) :-
        extraction(X,L1,L2),
        extraction(Y,L2,L3),
        add3(X,Y,F1,Z,F2,L3,L4).
add3(X,Y,F1,Z,F2,L3,L4) :-
        var(Z), !,
        add(X,Y,F1,Z,F2),
        del(Z,L3,L4).
add3(X,Y,F1,Z,F2,L3,L3) :- !, add(X,Y,F1,Z,F2).
extraction(X,L1,L2) :-
        var(X), !, extraction1(X,L1,L2).
extraction(X,L,L).
extraction1(X,[X|Y],Y).
extraction1(X,[W|Y],[W|Z]) :- extraction1(X,Y,Z).
```

図　付録１．１　Prologによるふく面演算のプログラム・リスト

```
send([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]])<-
        different([S,E,N,D,M,O,R,Y]) &
        dif(M,0) & dif(S,0) &
        sum(R1,0,0,M,0)&
        sum(R2,S,M,O,R1)&
        sum(R3,E,O,N,R2)&
        sum(R4,N,R,E,R3)&
        sum(0 ,D,E,Y,R4).

remainder(1)<-true.
remainder(0)<-true.

digit(0)<-true.
digit(1)<-true.
digit(2)<-true.
digit(3)<-true.
digit(4)<-true.
digit(5)<-true.
digit(6)<-true.
digit(7)<-true.
digit(8)<-true.
digit(9)<-true.

different([])<-true.
different([X|Y])<-out_of(X,Y)&different(Y).

out_of(X,[])<-true.
out_of(X,[A|L])<- dif(X,A) &out_of(X, L).

sum(R,X,Y,Z,R1)<-
        remainder(R)&
        digit(X) &
        digit(Y) &
        value(X,Vx)&
        value(Y,Vy)&
        value(R,Vr)&
        T is Vr+Vx+Vy &
        Vr1 is T/10 &
        Vz is (T mod 10)&
        Vr1=R1 &
        Vz=Z.
```

図　付録１．２　ＣＩＬによるふく面演算のプログラム

付録２．　検索問題のプログラムでのcopy述語やvar 述語の使用方法

一般にcopy述語やvar 述語は、並列実行環境下においてその動作がスケジュールに依存してしまい、動作が保証できないうという問題を持っている。この理由は、copyしようとする変数あるいはvariableかどうか調べる変数が、本来instantiate されるべき変数であるのに、スケジューリングによってはcopyしようとする際あるいはvariableか調べようとする際、variableのままであることがあり、copy述語やvar 述語の動作がスケジュールに依存してしまうからである。しかし、ふく面演算などの検索問題のプログラムを記述する際、copy述語やvar 述語を使う方がプログラムの記述が容易になることがある。そこで、ここでは、検索問題をＫＬ１でプログラミングする場合に限ってcopy述語やvar 述語の動作をスケジュールによらず保障する方法について述べる。ただし、この方法ではユニフィケーションやヘッド・ユニフィケーションなどに次のような制約を加える必要がある。

(1) ユニフィケーションはプリミティブである。構造体などのユニフィケーションの場合、構造のユニフィケーションと構造の中のユニフィケーションは同時に行なわれるものとする。

(2) ヘッド・ユニフィケーションはガードの実行前に行なわれ、ガードを実行する時には完了しているものとする。

(3) コミットされたならば、ガードの外ではガード内でのユニフィケーションの結果が保障されているものとする。

検索問題のプログラムでのcopy述語とvar 述語について見ると、次のような使われ方をしていることが多い。

```
if    解の候補を格納する変数がvariableであれば        ①
  then
      解の候補を格納する変数をcopyし、                ②
      copyした変数に解の候補を格納する。              ③

      copyした解の候補を格納する変数を                ④
      testerに渡す。
```

```
else
        解の候補を格納する変数をtesterに渡す。        ⑤
```

上のプログラムでは、copy述語とvar 述語の両方とも解の候補を格納する変数に限って使われ、解の候補を格納する変数へのinstantiation は③の部分だけで行なわれている。そこでvar 述語（①）とcopy述語（④）の動作をガード・オペレータを使って逐次性を保障し、それらの述語の動作をスケジュールに関係なく保障する。実際にＫＬ１で記述したプログラム例を以下に示す。

```
generator(X,Y) :- var(X)   |   p(X,Y)                    ①

                          copyした変数に解の候補を
                          格納する。

P(X,Y) :- copy(X,X1), permutation(X1,Y) |                ②と③

          tester(X1,Y).                                  ④

generator(X,Y) :- nonvar(X) | tester(X,Y).               ⑤
```

３章でのふく面演算のプログラミングに際しては、上記に示した方法にしたがってcopy述語やvar 述語を使用した。

付録３．　merge とＫＬ１プログラムの実行サイクル数との関係

３．３のプログラムmask4 の説明で述べたように、プロセス間のデータ通信をmerge を使ってストリームで行なうか、ゴールの呼び出しかの差は、実行サイクル数に関して大きな差はない。実際にはその差はゴールmerge 分の１サイクルのみである。理想的な並列実行環境下ではmerge を使うことは実行時間に対して大きな影響を与えないと思われる。そのため、merge の数だけを減らすようなプログラム上の工夫は必要ないと思われる。

merge によるＫＬ１プログラムの実行サイクル数の増加を次の２つのプログラムで調べた。プログラム１は２つのプロセス間のデータ通信をmerge を使ってストリームで行なっており、プログラム２はゴール呼び出しで行なっている。サイクル数はプログラム１がプログラム２より１だけ多いだけであった。この差は各プロセスのリダクションの数によらない。図付録３．２にプログラム１と２の実行経過を示す。

付録４．　分割ユニフィケーションでのメタコールの問題点

ここで分割ユニフィケーションとは、２つのtermのユニフィケーションが一度に行なわれるのではなく、分割して行なわれるようなユニフィケーションのことを言う。分割ユニフィケーションでの構造体のユニフィケーションは構造のユニフィケーションと構造のなかのユニフィケーションが同時に起こるとは限らない。このような分割ユニフィケーションはプリミティブなユニフィケーションに比べて不可分な部分が小さいため並列実行に適している。しかし、この分割ユニフィケーションはメタコールの動作を保障しないものになる。例えば、次のようなゴールは分割ユニフィケーションの元では動作が保障されない。

```
B=(a(b)=a(c)), call(B,Result, _), …
```

正しい実行ではResultの値はfailにならなければならないが、分割ユニフィケーションではB の値が実行途中で次のような値になることを許しているため、

```
B=(a(X)=a(Y))
```

このようなB の値でメタコールが実行されると、Resultの値はsuccess になる。

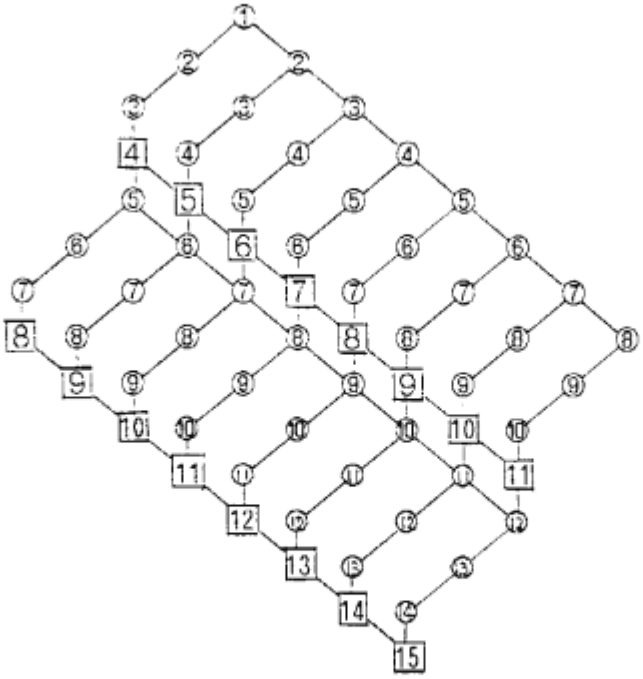
```
%
% 83 reductions and 19 suspensions in 16 cycles and 289 msec. (287 rps.)
%

test :- process1(8,X), process2(X,Y).

process1(Number,Out) :-
        Number \= 0, Number1 := Number - 1 |
        process1_goals(Number,Out1), process1(Number1,Out2),
        merge(Out1,Out2,Out).
process1(0,Out) :- Out = [].
process1_goals(A,Out) :- goal(A,B), Out = [B].

process2([A|Input],Out) :-
        process2_goals(A,Out1), process2(Input,Out2),
        merge(Out1,Out2,Out).
process2([],Out) :- Out = [].
process2_goals(A,Out) :- goal(A,B), Out = [B].

goal(A,B) :- B = A.
```

(a) プログラム1

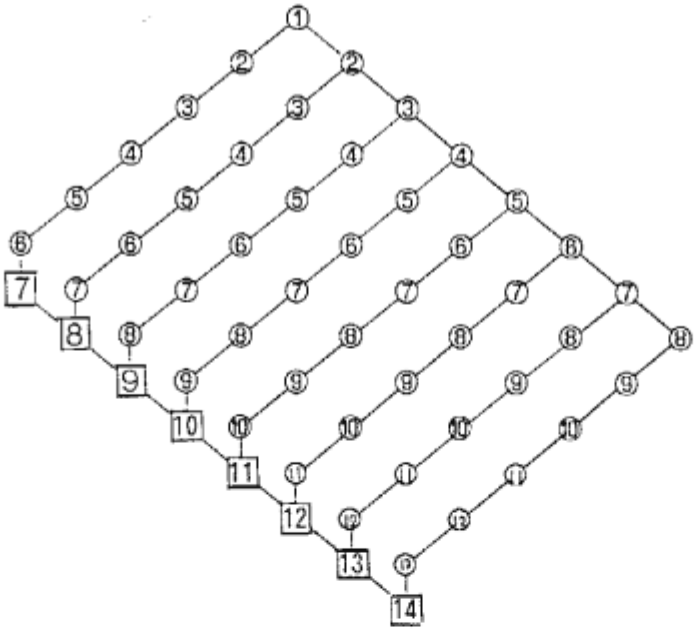
```
%
% 66 reductions and 22 suspensions in 15 cycles and 233 msec. (283 rps.)
%

test :- process1(8,X).

process1(Number,Out) :-
        Number \= 0, Number1 := Number - 1 |
        process1_goals(Number,Out1), process1(Number1,Out2),
        merge(Out1,Out2,Out).
process1(0,Out) :- Out = [].
process1_goals(A,Out) :- goal(A,B), process2(B,Out).

process2(A,Out) :-
        process2_goals(A,Out).
process2_goals(A,Out) :- goal(A,B), Out = [B].

goal(A,B) :- B = A.
```

(b) プログラム2

図　付録3．1　プログラム例



(a) プログラム１の実行経過



(b) プログラム２の実行経過

図　付録３．２　　プログラムの実行経過