Concurrent Prolog における ストリームの効率的併合・分配法

上田　和紀　（日本電気（株）C&Cシステム研究所）
近山　　隆　（（財）新世代コンピュータ技術開発機構）

1．はじめに

Concurrent Prolog [Shapiro 83-1]（以下CPと略す）のような並列実行型の論理プログラム言語で，大規模な分散型システムを記述しようとする場合，プロセス間通信路としてのストリームを効率的に併合・分配できるかどうかが，システムの性能に大きくかかわると予想される．本論文では，多数の入力ストリームの併合，および1本の入力ストリーム上のデータの多数の出力ストリームへの分配のための述語を，効率よく実現するための技法について検討する．

本論文で考察の対象とするのは，通常の逐次型の計算機上での実現法である．並列計算機上でのCPの実用性・有用性を実証するためには，もちろん逐次型計算機上における検討だけでは十分とはいえない．しかしそれは，CPを検討するうえでのひとつのステップとして重要であり，また逐次型計算機上での実用的なCP処理系の作成に直接役立てることができる．

また，かりに並列化が実現したとしても，ひとつのプロセサが複数のプロセスを受け持つというのはごく普通のことであろう．この場合，ひとつのプロセサの中における通信には，ここに示す方法がそのまま適用できる．

1．1　CPにおけるストリームの重要性

CPで並列処理，ないしはcoroutine 的な処理を実現するには，個々のプロセスを，AND 並列に実行される述語によって表現し，プロセス間通信を，それらの述語の間の共有変数によって実現すればよい．このとき，共有変数は，複数（通常はふたつ）の述語の間を流れるデータ，もしくはメッセージの列（通常リストで表現される）を表わし，プログラムの実行が進むに従って，値がうしろの方まで確定してゆく．一方，述語は，宣言的にみれば，引数の共有変数のとりうる値（の関係）を記述したものとみなせ，手続き的には，共有変数のデータ列の値を，先頭から順に（多くの場合tail recursionによって）処理するプロセスとみなせる．このような使われ方をする共有変数のことを，ストリームと呼ぶ．

プロセスやストリームは，CPの文法上の概念ではなく，運用論上の概念であることに注意されたい．

上記の説明から明らかなように，他プロセスとのデータやメッセージのやりとりは，そのプロセスの名前を指定して行なうのではなく，あらかじめ確保してあるストリームの値を検査（受信時）または具体化（送信時）することによって行なわれる．したがって，ストリームにかかわる操作――送受信，併合，分配――の効率はきわめて重要であると言える．

なお，CPにおいても，プロセス名の指定による通信方式を“模擬”することは可能である．

1．2　多分岐で動的なストリームの併合・分配の必要性

いくつかのプロセスを

```
:- in(I), process1(I,C), process2(C,O), out(O).
```

というように縦形に接続し，パイプライン的に処理を行なうような場合は，ストリームの併合・分配は不必要である．しかし，多数のプロセスからデータ・メッセージを受信する必要のあるプロセス――共有資源を管理するプロセスなどーーは，入力ストリームの併合プロセスをfront-end に用意しなければならない：

```
:-p1(C1), p2(C2), ..., pn(Cn),
  merge(C,C1,C2,...,Cn), resource(C).
```

不特定多数のプロセスからのメッセージを受けつけるためには，さらに，併合する入力ストリーム数が動的に変更できるようになっていなければならない．つまり，新たに共有プロセスと交信する必要の生じたプロセスは，front-end の併合プロセスに（他の入力ストリームか，要求専用のストリームを利用して）要求を発し，新たな入力ストリームを敷設してもらう必要がある．別法として，すでに敷設されている入力ストリームの根元に2分岐の併合プロセスを取りつける方法も考えられようが，この方法をくり返し用いると，交信するプロセスの数に比例する遅れが発生する．

一方，メッセージの分配は，それが“一斉放送”の形で行なわれるならば，各プロセスが放送用のストリームを共有するだけでよい：

```
:- sender(B), p1(B), ..., pn(B).
```

しかし，特定のプロセスと交信をしたい場合で，そこへのストリームを持っていない場合は，交信先プロセスの管理プロセスを経由して通信を行なわなければならない．このとき管理プロセスは，受けつけたメッセージを，行先の指定にしたがって，適当に分配する必要がある．

この管理プロセスについても，管理するプロセスの数を動的に変更する方法を考えておくことが望ましい．

1．3　従来の研究

Shapiro らは，[Shapiro 84]で，多入力（以降，本論文では，入力ストリーム数をnと記す）で，かつ入力数が可変な併合プロセスの問題を扱っている．彼らは，

① 2入力merge による不均衡木（新たな入力ストリームを，既存のストリームへの2入力merge の接続によって確保すると，一般に不均衡2分木ができる）において，n-bounded waiting と最悪 O(n) の遅れを保証するための方法
② 2入力と3入力のmerge によって 2-3木[Aho 74]を構成し，n-bounded waiting と最悪 O(log n) の遅れを保証するための方法

を示した．ここでn-bounded waiting とは，併合プロセスに到着したメッセージが出力に現れるまでに，高々n個の（他ストリームからの）入力メッセージにしか追いこされないことである．

①の方式の O(n) の遅れは，nが大きくてメッセージ数の多い場合は非許容的なものであろう．しかし，マルチプロセサ環境でのプロセサ間通信のような，本質的に“重い”通信においては，この方式も実用に堪えるかもしれない．

②の方式は，①に比べて遅れを大幅に改善するものである．しかし，手続き型言語ならば，プロセス間通信の遅れは，逐次型計算機上で模擬する限り，発信プロセス数に依存しない．だから，論理型言語の場合でも，少なくとも逐次型のvon Neumann 計算機上のimplementationにおいては，手続き型言語のときと同じオーダーの遅れを実現することが望まれる．

Gelernter は[Gelernter 84]で，マルチプロセス・システムの記述にＣＰが適しているかどうかを論じている．その中で彼は，併合プロセス網を用いたプロセス間通信は，"not only bulky but unduly constrincting" であると結論づけている（p.76）．しかしこれは，プログラムスタイルの面からの批判であって，本質的な記述能力や処理効率の観点からの批判ではない．

2．目標

我々の目標は，次のふたつである．

① 入力ストリーム数nが固定のとき，遅れが O(1) ですむような併合プロセスを逐次型計算機上に実現すること，また遅れが O(1) ですむようなn出力の分配プロセスを実現すること
② ①の解を，nが動的に増減する場合に拡張すること

①を実現するためには，2入力や3入力のmerge の組合せではうまくゆかないことは明らかである．つまり，n入力のmerge

$$
\begin{aligned}
&\mathrm{merge}([X \mid Ys],\ X_1, \ldots, X_{j-1},\ [X \mid X_j],\ X_{j+1}, \\
&\qquad \ldots,\ X_n) \\
&\ :\!-\ \mathrm{merge}(Ys,\ X_1, \ldots, X_{j-1}, X_j,\ ?,\ X_{j+1}, \ldots, X_n).
\end{aligned}
$$

をもとにしなければならない．

この述語は，インタプリタで実行すると，ひとつのメッセージの処理のために，各節の大きさ（＝O(n)）に比例する手間がかかってしまい，均衡木による方法よりかえって低効率になってしまう．しかし，3．1で述べる考察から，この述語は，コンパイルすれば，よりよい効率が得られそうである．そこで，本論文では，コンパイル技術の検討を中心的課題とする．

n入力のmerge を使用する際には，“遅れ”の定義も，併合木の段数という形ではとらえられない．ここでは，遅れを次のように定義する．

> 入力待ち状態の併合プロセスの入力ストリームにメッセージが到着してから，それを処理する節を選択し，ユニフィケーションによってメッセージを出力ストリームに移し，再帰呼出しによって再びもとの待ち状態になるまでの手間．ここで手間は，逐次型計算機で単位時間で実現できる基本操作の回数によって計算する．

分配も，併合と同じ考え方で検討することとする．

2．1　ＣＰの逐次型計算機上での実現法の概要

逐次型計算機上にＣＰを実現した例としては，[Shapiro 83-1]［新田84］があるが，これらはいずれもインタプリタである．ここでは[Shapiro 83-2]に述べられている方針に従ったコンパイラによる実現法を仮定する．以下，その方針の中の，制御に関する局面について簡単に述べる．

AND の関係にある一連のプロセス（述語呼出しの並びに対応）の記述子は，AND-loopという双方向環状リストを構成し，ORの関係にある一連のプロセス（述語を構成する各節に対応）の記述子は，OR-loop という双方向環状リスト

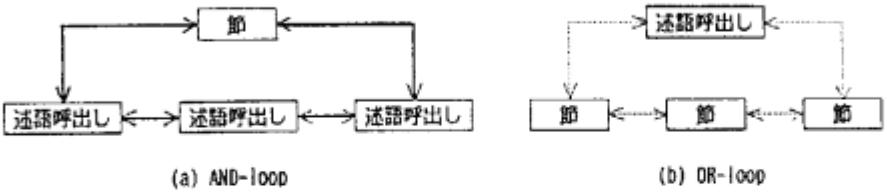

図1－1　AND-loopとOR-loop

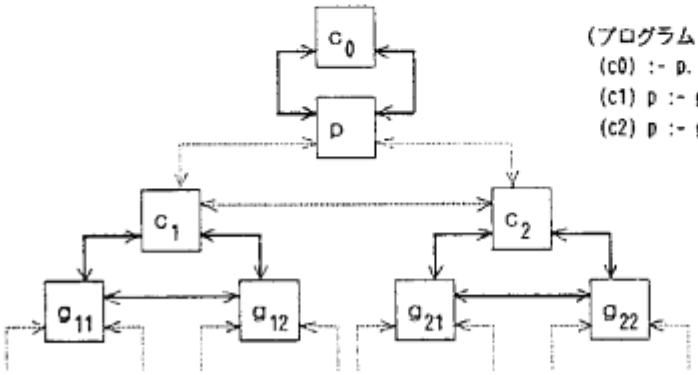

図1－2　AND/OR-loop による木構造

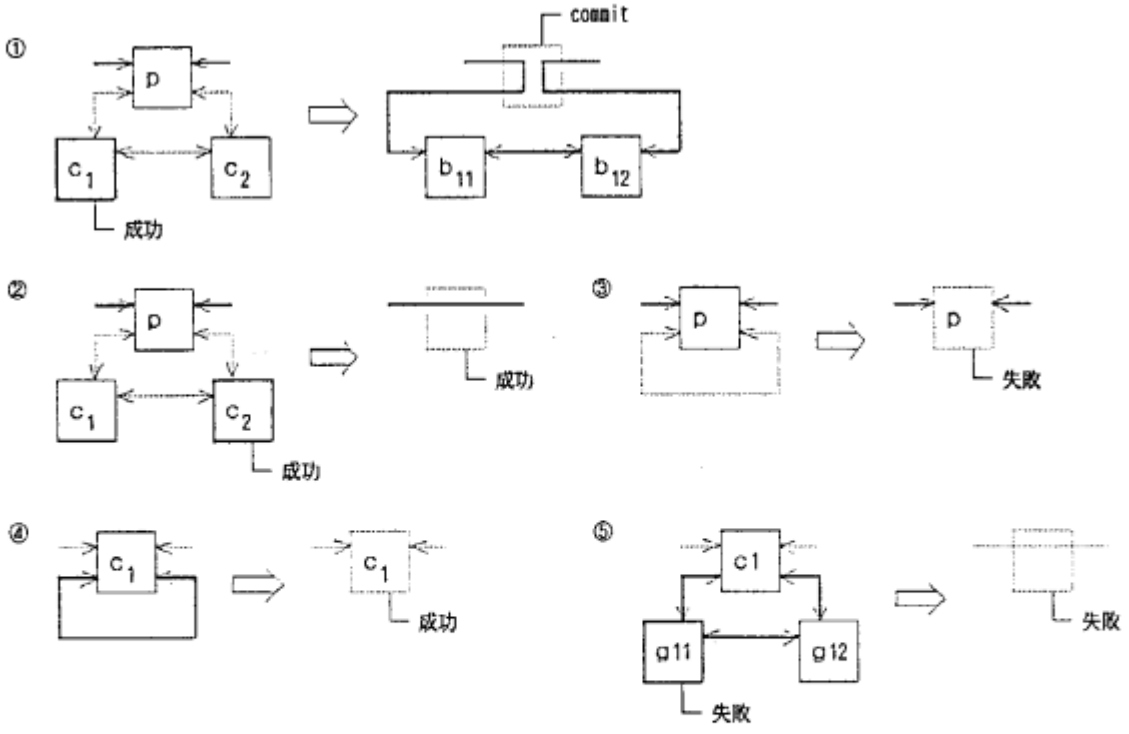

図1－3　AND/OR-loop の変化とその伝播

を構成する（図 1-1,2）．

AND-loopの要素は，commitされる前は，候補節からなるOR-loop の親となり，commitされると，本体部の述語呼出しに対応する双方向リストでおきかえられる．本体部が空（述語の成功）ならば，もとのAND-loopの要素は消滅する．要素がひとつもなくなったAND-loopの親は，成功である．逆にAND-loopの要素の失敗は，親の失敗とみなされる（図1-3）．

OR-loop の要素は，commitされていない候補節を表わし，ガード部の述語呼出しを要素とするAND-loopの親となる．OR-loop の要素の成功は，対応する節のcommitを意味する．逆に，OR-loop の要素が失敗したときは，その要素は消滅する．要素がひとつもなくなったOR-loop は，親の失敗とみなされる（図1-3）．

システムにはプロセスの待ち行列があり，AND/OR-loopからなる木の末端の要素，つまりloopの親になっていない要素が並んで，スケジュールされるのを待つ．ユニフィケーションの際に，読出し専用変数による待ちにはいった節は，プロセス待ち行列には並ばずに，その読出し専用変数に付随する待ちリストに付加される．それらの節は，読出し専用変数が具体化したときに，再びスケジュールの対象となる．

上記の方式に対して，OR-loop をなるべく作らないようにする改良が考えられる．それは，各節の頭部のユニフィケーションと，ガード部の簡単な検査を，不可分操作として，（書込み可能変数のコピーを作ることなく）行なってしまおうというものである．これをimmediate check とよぶ．Immediate check だけでcommitできれば，OR-loop を作らずにすむ．そうでない場合は，immediate check で待ちにはいった節と，immediate check は成功したが複雑なガードを持っている節に関して，OR-loop を作成し，待ちにはいる．

3．併合プロセスの実現法

3．1　n入力merge 述語に対する考察

n入力merge は，終端条件を考えなければ，下の形のn個の節で表現できる（終端条件の扱いについては，3．4で述べる）．

```
merge([X | Ys], X1, ..., [X | Xk], ..., Xn)
:-  merge(Ys, X1, ..., Xk?, ..., Xn).
```

この述語には，次のような特徴がある．

① merge の呼出しに対して，第c節が選択可能であるかを調べるには，第0引数と第c引数のユニフィケーションだけ試みればよい（以下，引数の番号づけは0からとする）．

② 第c節を用いたtail recursionの際，もとの呼出しのときから変化する引数は，第0引数と第c引数だけである．したがって，もとの呼出しのときの引数表があれば，そのなかの2か所を書きかえるだけで，新しい引数表ができる．

③ すべての節が待ち状態にあるとき，入力引数となっている読出し専用変数の具体化によって，再検査が必要となる節は，ひとつ（終端条件の節を含めてもふたつ）である．

さらに，tail recursionについて考察してみよう．tail recursionで変化しない引数に関しては，その引数に起因する（各節の）待ちの状況も変化しないという性質がある．たとえば，あるmerge の呼出しで，第1～6節が待ち状態となり，第7節が選択されたとする．この場合，もとの呼出しとtail recursiveな呼出しとでは，第1～6引数は全く変わらないから，tail recursion後も，第1～6節は待ち状態とならざるを得ない（ただし，第0引数が具体化することの影響が第1～6引数のいずれかに及べぶという特殊な状況では，待ちが解ける可能性もある）．

これを一般化していうと，次のようになる．

- ある呼出しで，第c節は，第k引数のユニフィケーションの中断（または失敗）のために選ばれず，結局第d節が選ばれたとする．このとき
  - ① 第c節の頭部の第k引数と，再帰呼出しの第k引数とが同じで，
  - ② 第d節の他の引数のユニフィケーションによって，第c節の第k引数のユニフィケーションの中断の原因となった読出し専用変数が具体化することがなければ，

  再帰呼出しの後も，第c節の第k引数のユニフィケーションは中断（または失敗）する．

n入力merge に即していうと，次のようになる．

④ 第c節を用いたtail recursiveな呼出しの際，候補となる節は，
  - a. 第c節
  - b. 前回の呼出しの際，候補であったが検査しなかった節
  - c. 読出し専用変数の具体化によって待ちの解ける節

である．c.は，通常の用法では存在しないから無視する．b.は“繰越し分”であって，検査をすれば，その節は候補でなくなる（中断または失敗する）か，選択されてtail recursionがおき，再び候補節になるかの

どちらかである．したがって，tail recursionのたびにかかる節の平均的な検査回数は，候補節の数には依存しない．

これらの考察から，n入力merge が，メッセージ１個の処理のために行なう

a. 候補節の選定，検査（＝頭部のユニフィケーション）
b. tail recursionに伴う引数表の更新と，待ち状況情報の更新

は，適切な処理方式を選べば，nによらない手間でできそうであることがわかる．

３．２　入力数固定のmerge 述語向き実現技法

n入力のmerge を効率よく実現するためには，次のことが必要である．

① たとえすべての節が待ちにはいっても，OR-loop は作らず，プロセス記述子のレベルで待つようにする（OR-loop 作成には最低 O(n) の手間がかかる）．
② 述語呼出しの引数表は再利用する．
③ 調べても無駄な節を調べないようにするために，プロセス記述子の中で，候補節の管理を行なう．

この指針に従った，プロセス記述子の構成と，述語の実現技法を以下に示す．ここに示す技法は一般的なものであり，merge にしか適用できないものではない．ただガード部のある節をもった述語は，ここでは検討対象としない．

なお，以下では，述語を構成する節の個数をM，引数の個数をNで表わす．

（ア）　プロセス記述子の構成

プロセス記述子は，次の項目を持つ．

① AND brothers：　AND-loopを構成するための２本のポインタ．
② Backward Pointer：　プロセス待ち行列のエントリ，および読出し専用変数の待ちリストのエントリを指すポインタの配列．前者はプロセスに１個，後者は節に１個必要なので，全体の要素数はM＋１．
③ Candidate Queue ：　当プロセス記述子が表わす呼出しの候補節の待ち行列．M要素．
④ Clause State：　各節の状態が，candidate，suspend，failのいずれであるかを表わす配列．M要素．
⑤ Suspend/Fail Table：　ある節のユニフィケーションの中断・失敗の原因は，呼出し側のいくつかの引数に求められる．だから，それらの引数の値がtail recursionの際に別の値になれば，その節は選択可能になるかもしれない．そこで，中断・失敗している節の番号ｃと，その原因の引数の番号Ｋとの対（ｃ，Ｋ）の表を用意する．この表は，ｃを索引とする順次検索と，Ｋをもつ要素の一斉削除とが，検索・削除する要素数のみに比例する手間でできなければならない（たとえば図２のような構造は，この条件をみたす）．最大要素数はプログラムに依存するが，merge の場合，O(N+M) = O(n) である（後述）．

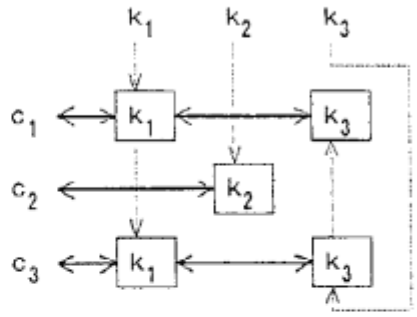

図２　Suspend/Fail Tableのデータ構造例

⑥ Fail Count：　当呼出しで選択されえない節の総数．Mになれば，呼出しの失敗である．
⑦ 述語のプログラム・コードへのポインタ．
⑧ 引数表：　Ｎ要素．

（イ）　操作

A．プロセス記述子の生成

新たに（つまりtail recursionでなく）述語が呼ばれた場合，プロセス記述子用の領域を割り当て，各エントリを設定する．ここで，

②のプロセス用Backward Pointerには，実行待ち行列のエントリを指させ，節用のBackward Pointerはすべてnilとする．また，
③のCandidate Queue にはすべての節を登録し，
④のClause Stateはすべてcandidate とし，
⑤のSuspend/Fail Tableは空とし，
⑥のFail Countは０とする．

B．節の選択

B-1.　Candidate Queue が空でなければ，その先頭の節（第ｃ節とする）の頭部と，呼出し側の引数（プロセス記述子の引数欄）とをユニファイするための命令を実行する．

例．　merge の場合は，第０引数と第Ｋ引数のユニフィケーションの命令だけが実行される．

その結果，
- 成功の場合は，本体の実行（D.参照）にゆく．
- 失敗の場合は，
  ① 生成した変数と値の結合を回収し，
  ② Fail Countを１ふやし，
  ③ この節のClause Stateをfailとし，
  ④ Suspend/Fail Tableへの登録を行ない（（ウ）参照），
  ⑤ 他の候補節を試みる．
- 中断した場合は，
  ① 生成した変数と値の結合を回収し，
  ② この節のClause Stateをsuspend とし，
  ③ Suspend/Fail Tableへの登録を行ない，
  ④ 中断の原因となった読出し専用変数の待ちリストに，プロセス記述子へのポインタpと，この節の番号cの対を登録し，
  ⑤ この節用のBackward Pointerに，③で登録したエントリを指させ，
  ⑥ 他の候補節を試みる．

B-2. Candidate Queueが空のときは，Fail CountがM（節の数）ならばプロセス（述語呼出し）の失敗処理を行なう．さもなければ，当プロセスの実行を中断する．

C．読出し専用変数の具体化

読出し専用変数が具体化したら，それに付随する待ちリストの各エントリ（p，c）について，次のことを行なう．

① pの指すプロセス記述子について，以下のことを行なう．
- cのClause Stateをcandidate とし，Candidate Queue にcを登録する．
- Suspend/Fail Tableから（c，－）（－はdon't careを表わす）の形の要素をすべて削除する．

② pをプロセス待ち行列に登録し，そのエントリをBackward Pointerに指させる．

D．本体部の実行

Commitした節（第c節とする）の本体に，再帰呼出しが含まれている場合，次の作業を行なう．

① この呼出しの引数は，頭部の引数と第k1，k2，...，ki引数が異なっているものとする．各ki（i=1，...，l）について，以下のことをおこなう．
- Suspend/Fail Tableから（c，ki）の形の要素を検索し，各cについて，
  - cのClause Stateがfailならば，Fail Countを１へらす．suspend ならば，第c節用のBackward Pointerの指す待ちリストのエントリを削除し，このBackward Pointerも消去する．
  - cのClause Stateをcandidate とし，cをCandidate Queue に登録する．
  - （c，－）の形の要素をSuspend/Fail Tableから削除する．
- 引数表の第ki要素を書きかえる．

② 第c節をCandidate Queue に登録する．

③ 節の選択（B.参照）を行なう．

再帰呼出し以外の呼出しが含まれている場合には，それに対して新たなプロセス記述子を生成すればよい．

再帰呼出しがない場合は，もとのプロセス記述子の領域は不要となる．これは，読出し専用変数の待ちリストからのポインタを回収したあとで，解放する．ただ，この領域を最適化にうまく役立てることができる場合もある（３，４参照）．

（ウ）Suspend/Fail Tableの管理

n入力merge の第c節を

```
          (0)       (c)
:- merge(Ys, ..., Xs?, ...).
```

と呼ぶと，第c引数のユニフィケーションが中断する．この場合の中断の原因は，呼出し側の第c引数のみにあり，他の節が選択されてtail recursionがおきたとしても，この原因は解除されない．しかし，第c引数のユニフィケーションの中断・失敗の原因を，いつでもその引数だけに求めてよいわけではない．たとえば，

```
              (0)     (1)          (c)
:- merge( [3|Ys],[3|Zs],...,[2|Xs],...).
```

と呼ぶと，ユニフィケーションを左から行なった場合，第c引数のユニフィケーションが失敗する．このときの原因は第0引数にも求められるべきである．実際，第c節は，第1節が選択されてtail recursionがおきれば，直ちに選択可能になる．

以上の考察を一般化すると，次のことがいえる．

- 第c節の第k引数のユニフィケーションが中断または失敗した場合，第c節において第k引数に“関連する”すべての引数（番号をk1，...，kl とする）について，（c，ki）のSuspend/Fail Tableへの登録を行なうべきである．

ここで項a，bが“関連する”（aRb と表記する）とは，a中の変数とb中の変数のなかに“関連する”ものがあることであり，変数v1とv2が“関連する”とは，v1とv2が

関係S： ガード部の，ある述語呼出しの中に，両変数が共に現れる

の反射推移的閉包（reflexitive transitive closure）で関係づけられていることとする（ガード部が空ならば，Sは変数の同一性を表わす）．

例． 引数の集合Aの，Rによる商集合A／Rは，mergeの第c節については

{{0,c}, {1}, ..., {c-1}, {c+1}, ..., {n}}

となり，

p(I,J,K,L,M):-a(I,J),b(J,K),c(L,M) | true.

では

{{0,1,2}, {3,4}}

となる．

ただ，上述のSuspend/Fail Tableへの登録規則には，例外を設ける必要がある．それは，merge の普通の呼出しで第c節が中断節となったときに（0，c）をSuspend/Fail Tableに登録すると，この節が，他の節によるtail recursionの際にも起こされてCandidate Queue に戻ってしまい，目的の効率が達成できないからである．そこで，

① 呼出し側の第k引数が（実行時にみて）読出し専用変数
② 頭部の第k引数が（プログラム上で）非変数項

という場合の中断だけは，（c，k）のみをSuspend/Fail Tableに登録することとする．それは，この中断の責任が，第k引数に“関連する”他の引数にはないことが明らかだからである．

なお，Suspend/Fail Tableに同時に登録される要素数は，

Σ（“引数集合／R”の要素の大きさの最大値）
（和は節についてとる）

を越えない．この値は，n入力merge の場合，O(n) である．

3．3 入力数固定のmerge 述語の性質

3．2で示したコンパイル技法を適用したときのn入力merge の性質を考察する．ここでは，終端条件を表わす節の存在は考えない．終端条件を表わす節がある場合については，3．4で考察する．

（ア）空間効率

プロセス記述子の大きさは O(n) である．なぜなら，Suspend/Fail Table以外の各項目の大きさは明らかに O(n) 以下であり，Suspend/Fail Tableの大きさは，3．2（ウ）に示したとおり， O(n) であるからである．

プログラム・コードの大きさについては，（エ）で論じる．

（イ）時間効率

A. プロセス記述子の生成： O(n) であるが，これは最初に1回だけ行なえばよい．
B. ユニフィケーション： 各節の頭部のユニフィケーションの手間は O(1) である．ユニフィケーションを試みるべき引数がふたつで，それぞれの手間がnによらないからである．ユニフィケーションが中断・失敗したときの作業（Suspend/Fail Tableと読出し専用変数の待ちリストへの登録）の手間も，図2のようなデータ構造を仮定すれば， O(1) である．
C. 読出し専用変数の具体化時： 各作業とも O(1) ．
D. Tail recursion： 第c節が選ばれたときに“変化する”引数は，第0引数と第c引数のみであるが，merge の通常の使い方では，第0引数は待ちや失敗の原因にはならないし，第c引数で待つ節は，第c節以外には存在しない．したがって，新たに候補となるのは第c節のみである．また，引数表の書きかえも2か所である．したがって，全体の手間は O(1) である．

以上より，入力待ち状態のn入力merge に到着したメッセージの処理に要する時間は，nに依存しないことがわかる．

（ウ）節の検査順序

merge の個々の節は，Candidate Queue に並んでいる順序に従って検査される．一度選択された節は行列の最後に並び直すので，n-bounded waiting が実現される．なお，3．1ですでに示したように，一旦中断・失敗した節は，その原因が消滅するまで行列からはずれるので，プロセスの処理効率を下げる要因とはならない．

（エ）プログラムの大きさ

個々の節に対応するコードは，3．2（イ）に示したB.とD.の作業を記述するものであるから，その大きさは O(1) である．また，A.の作業のためのコードは，述語に対してひとつあればよく，しかもその大きさは O(1) である．

したがって，n入力merge のコードの大きさはO(n)である．

しかし，このコード量は大幅に小さくすることができる．それは，各節に対応するコードがみなほとんど同じで，節番号cに関してパラメタ化できるからである．これを行なうと，述語全体のコード量が O(1) となる．

このパラメタ化は，各節の間の類似性を検出する機能をもった，非常に凝ったコンパイラによっても達成できよう．しかし，かりにそれを用意したとしても，ソースプログラムの大きさ（ $O(n^2)$ ）が，小さくできるわけではない．そしてそのコンパイルには最低 $O(n^2)$ の時間がかかる．しかも，この最適化機能の恩恵にあずかれるプログラムはごく限られたものであろう．これらのことを考えあわせると，n入力merge のコードは，システムで提供するのが最も現実的なアプローチであると言える．

さて，n入力merge が O(1) のコード量で実現されたとする．これで十分かというと，そうではない．なぜなら，システムは，あらゆるnのためのmerge を提供すべきだからである．これを個々に用意すると，nの最大値をnmaxとして， O(nmax)のコード量になってしまう．

ところが，ここでさらに最適化を行なうことができる．merge のコードは，nがちがってもほとんど共通であるから，nについてパラメタ化できるのである．これを行なうと，あらゆる入力数のmerge のためのコードが，全体で O(1) で表現できる．

なお，このコードのもつ機能をソースプログラムで表現すると，$O(nmax^3)$ のプログラム量となる．したがって，このコードはシステムが提供することが必須である．

3．4　入力ストリーム数の動的変更

上で検討した入力数固定のmerge は，入力数が静的にわかっているときにしかうまく使えない．これを拡張して，入力ストリームを追加したり，使用済みの入力ストリームを除去したりできるようにしてみよう．プログラムは下のようになる．なお，このプログラムは，入力ストリームの追加要求を，専用の引数（第(-1)引数とする）から受けつけている．この引数の表わすものは，追加ストリームのストリームである．

```
・第ｋ節（出力）
 merge(S, [X | Ys], X1, ..., [X | Xk], ..., Xn)
 :- merge(S, Ys, X1, ..., Xk?, ..., Xn).
・第０節（追加）
 merge([Xnplus1 | S], Ys, X1, ..., Xn)
 :- merge(S?, Ys, X1, ..., Xn, Xnplus1).
・第－ｋ節（削除）
 merge(S, Ys, X1, ..., [], ..., Xnminus1, Xn)
 :- merge(S, Ys, X1, ..., Xn, ..., Xnminus1).
・終端条件
 merge([], []).
```

これらの節は，tail recursiveではない．本体部で呼んでいるのはｎ±１入力のmerge だからである．しかしながら，これらの呼出しのためのプロセス記述子を作成するのに，もとのｎ入力merge の呼出しの際のプロセス記述子がうまく流用できれば，全く新たに作るのに比べてはるかに効率的な処理が可能であろう．

ＣＰの場合，各プロセス記述子の生成時刻を消滅時刻との関係は，プログラムによってまちまちであるから，その領域管理にはスタック方式ではなく，Buddy 法[Knuth 68][佐々83]のような一般的な記憶管理方式を用いる必要がある．

ここで，Buddy 法の利用を仮定しよう．すると，切り出される領域の大きさは２のベキとなる．これにｎ入力merge のプロセス記述子を割りつけるわけだが，その際，各欄の配置は，「切り出された領域に格納可能な，最も入力数の多いmerge のためのプロセス記述子」の配置に合わせる（つまり，切り出された領域の大きさに依存して決める）．すると，入力数が増えても，同じ領域が使用可能な限り，ほとんどの情報を再配置せずにすむ．

ここで，第(-n)～０節が選択された場合で，プロセス記述子が流用可能なときの操作を具体的に示そう．なお，プロセス記述子の流用を考える場合は，Clause Stateのとりうる状態として"unused"を追加し，Clause State用領域を割りあてた際，未使用部分は unused に設定しておくものとする．

A．第０節が選択されて入力ストリームが増えるとき

① （第±(n+1) 節の追加に伴う処理）第(n+1)，-(n+1)節のClause Stateがcandidate でない場合は，それらをcandidate とし，これらの節をCandidate Clauseに登録する．
② 第０節をCandidate Queue に登録する．
③ 引数表の第(-1)引数を書きかえる
④ プログラム・コードを取り替える（プログラムがｎに関してパラメタ化されていれば，そのパラメタ値を取り替える）．

B．第 (-c) 節(c>0) が選択されて入力ストリームが減るとき

① （第ｃ引数の変化に伴う処理）Suspend/Fail Tableから，（ｃ'，ｃ）の形の要素を検索する（実はこのような要素は，あっても（ｃ，ｃ）だけである），各ｃ'について，次のことを行なう．

- c' のClause Stateがfailならば，Fail Countを１へらす．suspend ならば，第c' 節用のBackward Pointerの指す待ちリストのエントリを削除し，このBackward Pointerも消去する．
- c' のClause Stateをcandidate とし，Candidate Queue にc' を登録する．
- （c' ，－）の形の要素（あっても（c，c）だけ）をSuspend/Fail Tableから削除する．

② （第±n節の消滅に伴う処理）
- 第n節のClause Stateがfailならば，Fail Countを１へらす．第(-n)節についても同じことを行なう．
- Suspend/Fail Tableから，（±n，－）の形の要素を削除する．
- （Candidate Queue に登録されている第±n節については，何もしない．これらが検査対象となったら，Clause Stateをundefined にかえるほかは何もしない．）

③ 第(-c)節をCandidate Queue に登録する．

④ 引数表の第c引数を書きかえる．

⑤ プログラム・コードを取り替える．

A，Bのいずれも， O(1) の手間でできることは明らかであろう．

さて，ストリームを追加しようとしたが，既存のプロセス記述子が流用できない場合は，２倍の大きさの領域を新たに確保し，そこに引越す必要がある．逆に，ストリームを削除しつづけた結果，現在の半分の領域でプロセス記述子が表現可能になったら，各項目の間をつめあわせて，不使用領域を解放すべきである．それらの手順を示す．

A' ，引越しを伴うストリームの追加

① 現在のプロセス記述子用領域の２倍の領域を確保する．

② もとのプロセス記述子の各項目をコピーする．

③ Backward Pointerが押さえている，プロセス待ち行列と読出し専用変数の待ちリストのエントリを書きかえる．

④ 上記Aの作業を行なう．

B' ，詰め直しを伴うストリームの削除

① 上記Bの作業を行なう．

② Candidate Queue を調べ，第±n節が登録されていたら，それを削除する．

③ もとのプロセス記述子の項目を，現在の領域の前半に詰めあわせる．

④ 領域の後半を解放する．

A' ，B' の手間を考えてみよう．領域の確保，解放の手間を考慮しなければ，手間は，いずれも O(n) である．Buddy 法による領域の確保，解放には，

O(log(Buddy 法で管理する領域の大きさ))

の手間がかかるが，この値はプログラムの実行環境のみによって定まり，nには依存しない．したがって，A' ，B' の手間は，実行環境を固定すれば O(n) である．

さて，ストリームの追加，削除が平均的に O(1) でできるためには，A' またはB' を行なう頻度が，高々 O(n) 回に１回であることを保証する必要がある．しかし，これは容易である．現在の（たとえば）1/4 の記憶領域でプロセス記述子が表現可能になったときはじめて，B' を行なうようにすればよいのである．

4．分配プロセス実現法の指針

分配プロセスについては，大まかな概要のみを記すこととする．

4．1 出力数固定のdistribute述語

n出力の分配用述語distributeは，次のように表現される．

・第k節
```
distribute([[k,X) | Xs],Y1,..., [X | Yk],...,Yn)
 :- distribute(Xs?, Y1, ..., Yk, ..., Yn).
```
・第0節
```
distribute([], [], ..., []).
```

まず，待ちがない場合について考えよう．第１～k節を個々に検査したのでは， O(n) の手間がかかってしまうから，節のランダム・アクセスを行なわなければならない．Dec-10 Prolog のコンパイラ [Warren 77] は，第１引数のprincipal functor によるhashing によって節を選択するようなコードを生成する機能をもっている．しかし，ストリームの考え方に基づくプログラミングでは，これだけでは機能不足である．このdistributeの場合，第１引数のtertiary functor（２レベル下のfunctor ）によるhashing を実現してはじめて， O(1) の手間による節の選択が可能になる．

次に，merge の場合と同様，コード量を O(1) にすることを考えよう．各節のコードをパラメタ化してひとつにまとめることはもちろん必要だが，distributeの場合はさらに，節の選択が，hashing を伴わない，単なるindexingでできることを利用しなければならない．Hash表をもつと，

O(n) の領域が必要となるからである．

待ちがある場合はどうなるだろうか．この述語の通常の用法では，第0引数が待ちの原因となるが，第1～k節が個々に待ちにはいったのでは，目的の効率は達成できない．第1～k節は，indexingのときだけでなく，待つときも一括管理すべきである．つまり，節の群として読出し専用変数の待ちリストに登録し，待ちが解除したら，indexingによって節を選択するようにする．

4．2　出力ストリーム数の動的変更

出力ストリーム数を動的に変更する機能は，merge の場合と同様に重要であろう．これは，4．1のプログラムに次の節を追加することで実現できる．

・追加
```
distribute([grow(Ynplus1) | Xs], Y1, ..., Yn)
:- distribute(Xs?, Y1, ..., Yn, Ynplus1).
```
・削除
```
distribute([shrink | Xs], Y1, ..., Ynminus1, Yn)
:- distribute(Xs?, Y1, ..., Ynminus1).
```

Distributeにおける出力数変更を効率よく実現するためには，3．4でmerge について示したのと同様の手法を適用すればよい．

4．3　分配プロセスの実現技法の応用

Prologの問題点として，書きかえ可能な配列が用意されていない点がしばしばあげられる．もちろん，assert/retractを用いれば配列は模擬できるが，それは“論理的”な配列ではない．論理的な配列の実現するひとつの方向は，

・配列：　　配列型のデータ
・配列に対する操作：　配列を引数とする述語

という対応づけを行ない，データ構造の工夫によって効率向上を図るというものであるが，

・配列：　　述語（プロセス）
・配列に対する操作：　メッセージのストリーム

という対応づけを行ない，分配プロセスと同様の実現技法で効率を得ることも考えられる．配列をmutable objectとみなすならば，これはむしろ自然な解である．プログラムは，次のようになろう．

```
array(n, S) :- array(S, X1, ..., Xn).
array([read(k,Xk) | S], X1, ..., Xk, ..., Xn)
    :- array(S?, X1, ..., Xk, ..., Xn). (k=1,...,n)
array([write(k,Yk) | S], X1, ..., Xk, ..., Xn)
    :- array(S?, X1, ..., Yk, ..., Xn). (k=1,...,n)
```

さらに，要素数の問合せ・変更等のための節を追加することも可能である．

5．結論と今後の課題

ＣＰで書いたｎ入力merge の性質を調べ，各メッセージをｎによらない手間で処理する実現方式を示した．さらに，入力ストリーム数の変更も，ｎによらない手間でできることを示した．分配プロセスについても，ｎによらない手間でのメッセージ処理・出力ストリーム数変更ができることを示唆した．

ただ，これらの述語のコードについては，システムが用意すべきであるという結論に達した．システム側で用意すれば，あらゆるｎのためのmerge/distributeが，ｎによらない大きさのコードで実現できるからである．

逆に，ソースプログラムをユーザに用意させ，コンパイラによってコードを得ることは，得たコードの効率ではなく，ソースプログラム量・コンパイルの手間等の点で非現実的である．しかし，システムが提供したコードの機能が，ＣＰのプログラムとして表現可能であるということは，プログラミング・システムの構築など，いろいろな点で好都合であろう．

提案したmerge プログラムの実現技法には，出力ストリームにbounded buffer［竹内　83］が接続されていると，効率よく動かないという問題がある．しかし，この問題は，節の待ちとスケジューリングの方式の改良によって，解決可能であると考えている．

今後の課題として最も重要なものは，ＣＰによって大規模なシステムを記述し，プロセス間通信のための費用を見積り，提案した機能の有用性を確認することであろう．また，並列環境におけるプロセス間通信の効率的実現法を考えることも重要である．

謝辞

本研究の機会を与えて下さった日本電気（株）Ｃ＆Ｃシステム研究所の箱崎部長，山本課長，ならびに（財）新世代コンピュータ技術開発機構の淵所長，古川室長に感謝いたします．また本検討のきっかけを作って下さったＩＣＯＴ招聘研究員のDr. E. Y. Shapiro，有益な助言をいただいたＩＣＯＴの竹内彰一氏に謝意を表します．

参考文献

[Aho 74] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer

Algorithms, Addison-Wesley, Reading, Mass. (1974).

[Gelernter 84] David Gelernter, "A Note on Systems Programming in Concurrent Prolog", Proc. 1984 Int. Symp. on Logic Programming, pp.76-82 (1984).

[Knuth 68] D. E. Knuth, The Art of Computer Programming, Vol.1: Fundamental Algorithms, Addison-Wesley, Reading, Mass (1968).

[Shapiro 83-1] E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech Report TR-003, Institute for New Generation Computer Technology (1983).

[Shapiro 83-2] E. Y. Shapiro, Notes on Sequential Implementation of Concurrent Prolog: Summary of discussions in ICOT (1983) (unpublished).

[Shapiro 84] E. Y. Shapiro, C. Mierowsky: "Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog, Proc. 1984 Int. Symp. on Logic Programming, pp. 83-90 (1984).

[Warren 77] D. H. Warren, Implementing PROLOG--Compiling Predicate Logic Programs, Vol.1-2, D.A.I. Research Report No.39, Dept. of Artificial Intelligence, University of Edinburgh (1977).

[佐々 83] 佐々政孝，動的記憶領域割付け法，情報処理，Vol.24, No.4, pp.408-412 (1983).

[竹内 83] 竹内彰一，古川康一，"Concurrent Prologにおけるプロセス間通信の実現"，情報処学会第２７回全国大会3t-7 (1983).

[新田 84] 新田克己，Concurrent Prolog インタプリタについて，情報処理学会第８回ソフトウェア基礎論研究会発表予定．