AND OR 並列論理型言語 ANDOR-II の並列論理型言語への変換†

竹内彰一††* 高橋和子†† 坂本忠昭††

並列論理プログラミングはシステムプログラミングに代表されるような決定的なあるいは don't care 非決定的な並列システム記述に高い適合性を示す一方で，論理プログラミングが探索問題についてもつような don't know 非決定性に対して高い記述力をもたない．これは並列論理プログラミングにおいては OR 並列の計算がコミットオペレータにより著しく制限されているためである．一般的な並列問題解決システムを考えると don't care, don't know 双方の非決定性を記述できる並列言語が望まれる．先にわれわれはこのような要求を満足するものとして，AND 並列，OR 並列双方を備えた並列論理型言語，ANDOR-II，を提案した．われわれはまた，この言語の AND，OR 双方の並列性を最大限抽出した並列実行が，色付け方式という実装方式により可能であることも示した．本論文では，この色付け方式に基づき，この言語を AND 並列だけをもつ並列論理型言語に変換する方法について報告する．変換の説明は具体的に並列論理型言語の一つ KL1 への変換を示しながら行う．

1. はじめに

並列計算機が実用化，普及の時代になりつつある今日，これらのハードウェア資源を有効に活用する並列問題解決方式が強く求められている．しかし，新しい方式に望まれているのは問題解決手続きの記述が高並列に実行できることだけでなく，記述自身の抽象度の高さもある．様々な方式が近年提案されている．その中の一つ，論理フォーマリズムと並列処理を結び付けようとする試みは第5世代プロジェクトなどを中心に研究されている．

われわれはこの問題に対して，抽象度が高く，また並列実行性にも優れた問題記述言語を開発するというアプローチを展開してきた．具体的には，以下の三つを行ってきた．

- AND および OR 並列を備えた並列論理型言語，ANDOR-II，の設計とその並列プラニングへの応用[14]．
- AND および OR 並列計算の実装方式としての色付け方式の研究[13]．
- 色付け方式に基づいて，ANDOR-II プログラムを並列論理プログラムへ変換する方法の研究．

本論文では三番目について報告する．

ANDOR-II が AND と OR の両並列性を備えたことの理由は，OR 並列のみでは分散システムのような問題対象を記述できない，また AND 並列のみでは宣言的な問題記述ができないという認識による．AND と OR の両方の並列性を備えれば，探索などの don't know 非決定的（本論文では非決定性に言及する場合，Kowalski[9] にならい，don't care 非決定性，don't know 非決定性という二つの用語を使い分ける）な対象の記述に OR 並列記述を用い，並列事象の記述では AND 並列記述を用いることにより，問題対象の記述が簡潔に行えるようになる[14]．このような AND および OR 並列を備えた論理型言語の必要性は現在までに提案された様々な言語[1),3),4),6),7),10)-12),19)]にも共通してみられる．

このような記述力を持つ言語の高並列実行に関しては，既存の並列言語へ変換するという方向[1),3)]と，専用の並列処理系を開発するという方向[4),6),7),10),11),19)]の二つがある．前者は既存の並列処理系が利用できるという点で有利であるが，対象となる処理系に効率良く埋め込めるかどうかにより言語の仕様が制限を受けることがある．後者では，このような処理系からの制約を受けることはないが，自分自身の処理系を新規に開発しなくてはならない．われわれは ANDOR-II の並列処理系について，前者の方向に沿って，ANDOR-II を並列論理型言語へと変換する方式をとった．それは，

- 並列論理型言語に関しては実績のある並列処理系が開発されてきていること[8]，
- AND および OR 並列計算を粒度の細かい高並列

† Transformation of an And- and OR-Parallel Logic Programming Language, ANDOR-II, to a Parallel Logic Programming Language by AKIKAZU TAKEUCHI, KAZUKO TAKAHASHI and TADAAKI SAKAMOTO (Central Research Laboratory, Mitsubishi Electric Corp.).

†† 三菱電機(株)中央研究所
* 現在 (株)ソニーコンピュータサイエンス研究所
Currently, Sony Computer Science Laboratory, Inc.

計算として実現する方式，色付け方式，を開発したこと[13]，

- 並列論理型言語は粒度の細かい並列性を備えた汎用的な並列言語であり，高並列な色付け方式の実装に適していること，

による．本論文では色付け方式に基づいて，ANDOR-II プログラムを並列論理型言語のプログラムへと変換する方式について報告する．変換の説明にあたっては並列論理型言語として KL 1[8] を採用する．

ANDOR-II に関する一連の研究は，記述力があり実行効率も良い問題記述言語を開発するという共通の目標をもって行われている．この関連でいえば，本論文の目的は ANDOR-II の実装を具体的に与えることである．しかし，より広い視野に立てば，GHC[16] 等の並列論理型言語に効率的に埋め込むことができる AND OR 並列言語のクラスを具体的に変換法とともに示すという意義をもつ．

以下では第2章で言語 ANDOR-II について述べ，第3章で色付け方式について例を用いて概略を説明する．そして，第4章で，並列論理型言語の一つである KL 1 への色付け方式に基づいた変換について述べる．他の研究との比較については第5章で述べる．

2. AND OR 並列言語 ANDOR-II

一般に，GHC などの AND 並列計算モデルの表現力は OR 並列計算を限定するコミット選択メカニズムによってもたらされる．これに対し，本論文で取り上げる AND OR 並列計算モデルは，コミット選択メカニズムを持つ AND 並列計算とコミット選択メカニズムを持たない OR 並列計算を組み合わせたものである．この組み合わせは，ANDOR-II では，述語を AND 述語と OR 述語の二つのクラスに分割することにより実現されている．すなわち，決定的もしくは don't care 非決定的コンポーネントを記述するための AND 述語と don't know 非決定的コンポーネントを記述するための OR 述語である．

ANDOR-II のプログラムは AND 述語定義と OR 述語定義の集合である．AND 述語定義はモード宣言とガード付き節の集合からなり，モード宣言は節集合に先行しなくてはならない．OR 述語定義はモード宣言，OR 述語宣言，ガードなし節の集合という順の並びである．AND(OR) 述語定義で定義された述語はそれぞれ **AND(OR) 述語**と呼ばれる．ガード付き節，ガードなし節ともにそのボディ部で AND 述語および OR 述語の両方を呼べることに注意されたい．ガード付き節のガードに許される述語は，同一化や大小比較などのシステムが用意するプリミティブな述語に限られる．

OR 述語宣言は次の形である．

:- or_predicate P/N.

ここで P は述語記号であり，N はそのアリティである．

モード宣言は次の形である．

:- mode P $(m_1, \cdots, m_N)$.

ここで，P, N, m_i はそれぞれ述語記号，そのアリティ，i 番目の引数のモードである．ここで m_i は + あるいは − でなくてはならない．これらはそれぞれ「参照専用モード」，「書き込みモード」を表す．すなわち，m_i = + は i 番目の引数は P の計算の最中には決して具体化されることがないことを示す．−モードはそれ以外の場合を表す．

ANDOR-II プログラム中のすべての述語はモードを持ち，それぞれの述語の呼び出しはこのモードに違反する動作をしてはならない．モードに反した振る舞いをしようとしたプログラムの動作は保証されない．しかし，参照専用モードの意味が示唆するように，頭部の参照専用モードを持つ引数に出現した変数はガード部およびボディ部においてもアトムの参照専用モードの引数にしか出現することができない．このためプログラムがモードによる制約を満足しているかどうかは容易に検査できる．

このモードによる制約と並んで ANDOR-II プログラムを制約するものとして単一生成者制約がある．節は，その節中のいかなる変数もその右辺において書き込みモードを持つ引数中に出現する回数が高々一回であるときおよびそのときに限り単一生成者制約を満足するという．また，プログラムは，各節が単一生成者制約を満足するときおよびそのときに限り単一生成者制約を満足するという．この単一生成者制約と上のモード宣言とがわれわれの AND OR 並列言語を特徴付けるものである．

以下の議論を簡単にするために組込み述語として同一化のみを考慮する．そして，ボディ部で呼ばれる同一化の形を〈変数〉=〈項〉の形式に制限し，同一化の左辺は書き込みモード，右辺は参照専用モードとして扱かう．(通常使われる組込み述語に関しても，容易に同様の議論を展開することができる．)

3. 色付け方式

ANDOR-II のプログラムの AND OR 並列計算はアトムの論理積をもらってスタートする．アトムの論理積は世界と呼ばれる．一つの世界の中のアトムは並列に実行される．これは，AND 並列に相当する．何通りかのリゾリューションの可能性を持ったアトムが呼び出された場合には，その世界は概念的にいくつかの世界に分岐する．これらの世界は各リゾリューションが行われる世界に相当している．そして，これらの世界は並列に実行される．これは，OR 並列に相当する．ここで最も重要なのは分岐である．分岐のナイーブな実装はすべてのアトムの論理積をコピーすることであるが，予想されるオーバヘッドのために受け入れ難い．そこで，実装に工夫をしたり，元の論理プログラムに制限を加えたりする必要が出てくる．

ANDOR-II から KL 1 への変換は色付け方式という方式に基づいている．この変換方式においては色付きベクタと呼ぶ変数への多重バインディングを表現する構造を導入することにより OR 並列の計算を AND 並列の計算に埋め込む．この色付きベクタの各要素は値とその値の帰属する世界の識別子との対になっている．この識別子は色と呼ばれる．（これらの詳細については文献 13) を参照のこと．）

図 1 の例を用いてこれを説明する．述語 compute は入力リストから任意の要素をピックアップし，その二乗および三乗値の合計を返す．ここで，pickup は非決定的な操作であると見なされている．入力が [1, 2, 3] であれば，可能な解は $2(1^2+1^3)$, $12(2^2+2^3)$, $36(3^2+3^3)$ である．AND 述語と OR 述語の区別は述語が常にコミットするか全くコミットしないかであることに注意されたい．すなわち，述語の解の多重度とは直接的に関係していない．それゆえ，compute は解をたくさんもっても AND 述語と定義されるのである．

compute が入力 [1, 2, 3] で呼び出されると，このリストは直接 pickup に送られる．pickup は非決定的プロセスであり，最終的に 1, 2, 3 の三つの可能な解を生成する．ナイーブな実装方式では世界に対応するデータ構造を保持し，pickup が三つの解を生成する場合には，pickup の三つの可能な解を別々に処理するために，現在の世界に対応するデータの三つのコピーが生成されることが考えられる．

色付け方式では，一つの世界に対応した一つのデータ構造というようなものは存在しない．世界は同一のデータ構造を共有し，各世界は色と呼ばれる識別子を伴い，色 α の世界に所有される項は α でペイントされるようにし，ある世界のデータと他の世界のデータの混同を回避している．したがって，pickup が三つの解を生成する場合には，それらは別個の色でペイントされ，$\langle 1|\alpha_1, 2|\alpha_2, 3|\alpha_3\rangle$ のように任意の順序でパックされる．ここで，$\langle\cdots\rangle$ は色付きベクタという特別なデータ構造を表し，$X|\alpha$ は色 α の値 X を表している．各出力値がバラバラに送られるかわりに，この色付きベクタが Y を経由して square と cube に送られる（**図 2** 参照）．そして色付きベクタの各要素に対して，square と cube が演算を行い，それぞれ新しい色付きベクタ $\langle 1|\alpha_1, 4|\alpha_2, 9|\alpha_3\rangle$ と $\langle 1|\alpha_1, 8|\alpha_2, 27|\alpha_3\rangle$ を作成する．square と cube は決定的な述語であるため，色に影響を与えない．これら二つの色付きベクタで add が呼び出される．add は概念上 X と Y が同一の色を持っている場合にのみ，X^2 と Y^3 の和を計算することが求められている．すなわち，X と Y が同一の pickup 動作から導かれている場合である．そうでない場合には加算に適用されない．したがって，二つのベクタは事前処理されて同一色のペアの集合を作り，

```
:- mode compute(+,-), square(+,-), cube(+,-).
compute(X,Z) :-
     true |
     pickup(X,Y), square(Y,Y2), cube(Y,Y3), add(Y2,Y3,Z).

square(X,Y) :- true | multiply(X,X,Y).
cube(X,Y)   :- true | multiply(X,X,Z), multiply(Z,X,Y).

:- or_predicate pickup/2.
:- mode pickup(+,-).
pickup([X|L],Y) :- Y=X.
pickup([_|L],Y) :- pickup(L,Y).
```

図 1 ANDOR-II プログラムの例
Fig. 1 An example of an ANDOR-II program.

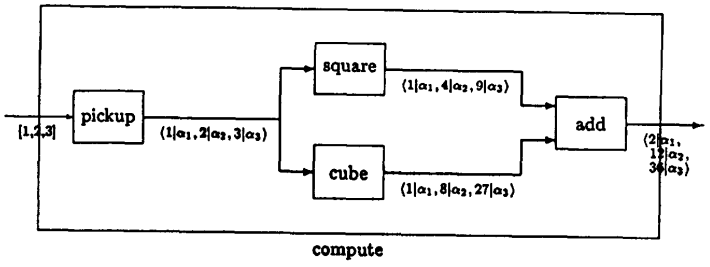

図 2 色付け方式
Fig. 2 Coloring scheme.

各ペアに対して加算が適用される．最終的には，解の色付きベクタ $\langle 2|\alpha_1, 12|\alpha_2, 36|\alpha_3\rangle$ が得られる．

4. KL1 による実現

ANDOR-II プログラムは並列論理プログラミング言語の一つである KL1[8] のプログラムへと変換される*．

ANDOR-II プログラムの変換は「モード解析」，「単一生成者制約検査」，「KL1 への変換」の三つからなる．モード解析と単一生成者制約検査の二つはソースプログラムの入力とともに即座に並行して行われ，KL1 への変換はモード解析の結果を受けて行われる．

4.1 ソースプログラム解析

モード解析は，各節ごとにモードと定義から頭部同一化中およびガード部の計算中に参照専用引数のデータがどのように参照されるかについてそのパターンをすべて抽出する．AND 述語の場合このパターンは述語全体で一つにまとめられるが，OR 述語の場合は各節ごとにまとめられる．各パターンは

(X, T)

という対で表現されており，頭部の引数中の項 T の中の X の値を参照するということを意味する．例えば，述語 p がある定義節の頭部で参照専用モードを持つ引数に項 $f(a, g(b), C)$ をもつとき，モード解析により以下の四つの対が抽出される．

$(X, X), (X, f(X, Y, Z)), (Y, f(X, Y, Z)),$
$(Y, f(X, g(Y), Z))$

これらはそれぞれ頭部同一化の際に

- ゴールの対応する項が $f(\ )$ という項かどうか
- $f(\ )$ の第一引数が a かどうか
- $f(\ )$ の第二引数が $g(\ )$ かどうか
- その $g(\ )$ の引数が b かどうか

が調べられることに対応している．また，これらの情報は実は頭部同一化時およびガード部の計算時に後で述べる「色付きベクタ」の出現する可能性のある場所を示しており，実行時に色付きベクタの検出をするためのコードを生成するために使われる．

単一生成者制約検査はオブジェクトコードの生成とは独立して行われ，違反があった場合には変換を途中で放棄する．

4.2 色付け方式の実装

最初に色と色付きベクタの実現について述べる．実行時に OR 述語が呼び出され，その定義節に沿って計算が進行すると，その各節につき一つの原色 (P, A_i) が割り付けられる．ここで P は OR 述語の呼び出しの識別子，A_i はそこで選ばれた節の識別子である．原色は一つの OR 分岐点における一つの選択を記録するものである．われわれの実装では P は実行環境の中に存在する ID サーバにより与えられる．すなわち，OR 述語はそれが呼び出される度に ID サーバに対して新しい識別子を要求し，ID サーバはそのような要求に対して 1 から N までの番号をユニークに割り付ける．このような ID サーバは生成する識別子が重ならない限りは一つの実行環境に複数あっても構わない．節の識別子 A_i については変換時に静的に用意する．すなわち，各述語定義 D につき，変換プログラムは節を 1 から D 中の節の総数 $|D|$ までの数で番号付けする．以上のようにした結果，原色は 1 から N までの数と 1 から $|D|$ までの数の対で表現されることになる．

色は原色の集合である．したがって，ある世界が過去に各 OR 分岐点においてどの節を選んだかという情報は，その世界の色の中に保持されている．現在の実装では色 α を n ビット文字からなる N 文字ストリングで実装している*．

$c_1 \cdots c_N$

ただし，c_i は n ビット文字であり，n は 2^n がプログラム中の最大の $|D|$ の値よりも大きくなるような数とする．色 α が上のようなストリング表現をもつとき，その中の i 番目の文字 c_i は，$P=i$ であるような原色 (i, m) についての情報を格納するフィールドであり，以下のような意味を持つ．

$$c_i = \begin{cases} m & \text{iff}(i, m) \in \alpha \\ 0 & \text{それ以外} \end{cases}$$

すべての色は同じ長さであり，すべての分岐点に関する情報を格納するフィールドを備えていることに注意．

N および n は実装に存在した数である．N は一回の実行で可能な最大 OR 分岐数を表し，n は一つの OR 述語定義に含まれる節の数を制限する．n の値はコンパイル時に十分余裕のある値が決定できるが，一方，解を出すのに十分な N の値というのは事前には

---

* ここで示す変換方式は例えば文献 14) で述べた方式とは異なる．そこではソースプログラムを徹底的に解析する方式が示されたが，この新しい方式では変換時の解析は行わず完全に動的な処理を行う．

* もっとも最近の実装では色は整数の集合として表現されている．これは長いストリングを短い固定長のものに切り分けたことに相当する．このようにすることにより and, or, exclusive or といった論理演算が使えるようになり効率の改善が計れた．本質は変わらないので本論文ではストリングを用いた方式の説明をする．

分からない．

原色 (P, A), (Q, B) は $P=Q \wedge A \neq B$ のとき直交するという．すなわち同じ OR 分岐点で異なる選択肢をとったときである．色は原色の集合であるが，その中のどの二つの原色も直交することはない．二つの色 α, β があり，その和集合 $\alpha \cup \beta$ が直交する原色を含んでしまう場合，α と β は直交するという．二つの色が与えられた時にそれらが直交するかどうかの判定は，異なる世界に属するデータを一緒に扱わないための基本的な手続きであるが，オーダ N で計算できる．

色を扱う述語 *add_color*(C_1, P, A, C_2) を導入しておく．*add_color* は C_1, P, A にそれぞれ色，分岐点の識別子，そこで選択した節の識別子をとり，C_1 と $\{(P, A)\}$ が直交しない時に C_2 として色 $C_1 \cup \{(P,A)\}$ を返す．

色付きベクタ $t(=\langle a^1|\alpha^1, \cdots, a^n|\alpha^n\rangle)$ は KL 1 のベクタというデータ構造 $\{\cdots\}$ を用いて次のように実現される．

$$t ::= \{\{v(a^1, \alpha^1), \cdots, v(a^n, \alpha^n)\}\}$$

色付きベクタは長さ 1 のベクタとして表現されており，同時に長さ 1 のベクタは他での使用を禁止している．したがって，サイズを調べることにより，色付きベクタか，通常のベクタかが直ちに分かるようにしている．

KL 1 ではベクタの要素は 0 から番号付けられており，ベクタ生成時には各要素が 0 で初期化されている．以下では KL 1 のベクタを操作する四つの述語，*vector*(X, L), *new_vector*(Y, L), *vector_element*(X, I, E), *set_vector_element*(X, I, E, N, Y) を使用する．これらの詳細については文献 8) を参照されたい．

4.3 KL1 への変換

4.3.1 AND 述語の変換

アリティ N の AND 述語 p の変換を考える．p が以下のようなモードを持つと仮定しよう．

$$:- \text{mode } p(\underbrace{+, \cdots, +}_{k}, \underbrace{+, \cdots, +}_{l}, \underbrace{-, \cdots, -}_{m}).$$

ただし，$k+l+m=N$ であり，最初の k 個の引数は少なくとも一つの節で頭部同一化時に参照されるか，あるいはガード部の計算中に一回は参照されると仮定する．さらに，次の l 個の引数はすべての節でボディ部へ直接パスされることを仮定する．**図 3** に変換の模様を図示する．

述語 p/N はアリティ $N+2$ の新しい述語 $p/(N+2)$ に変換される．増えた二つの引数は，*Col*, *Ids* と

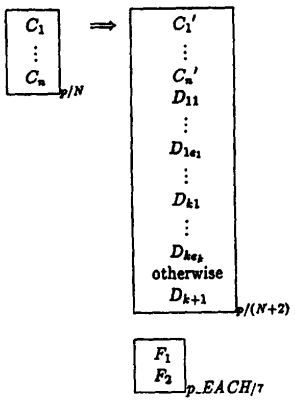

図 3 AND 述語の変換
Fig. 3 Transformation of an AND predicate.

通常表記され，それらはそれぞれ，色，ID サーバとの通信チャネルを保持する．$C_1, \cdots, C_n$ を p/N の定義とすると，$p/(N+2)$ の定義は

$$C_1', \cdots, C_n', D_{11}, \cdots, D_{1e_1}, \cdots, D_{k1}, \cdots, D_{ke_k},$$
$$otherwise, D_{k+1}$$

となる．

$C_1', \cdots, C_n'$ の各節は基本的に元の $C_1, \cdots, C_n$ の各々に対応している．またこの変換により新しい述語が導入される．それは *p_EACH/7* であり，F_1, F_2 で定義される．$D_{ij}(1 \leq i \leq k, 1 \leq j \leq e_i)$ の各節とこの新述語は参照専用引数に出現した色付きベクタが頭部同一化やガード部の計算に干渉する可能性のある場合を処理する．節 D_{k+1} は他の節がすべて頭部同一化あるいはガード部の計算に失敗した場合を扱う．

C_i が以下の形であるとする．

$$p(X_1, \cdots, X_N) :- G_1, \cdots, G_a | B_1, \cdots, B_b, E_1, \cdots, E_c.$$

ただし，$B_i(1 \leq i \leq b)$ はユーザ定義述語 $q_i(Y_1, \cdots, Y_{l_i})$ であり，$E_j(1 \leq j \leq c)$ は同一化アトムである．このとき，C_i' は**図 4** のようになる．*merge*$(\{X_1, \cdots, X_n\}, X_0)$ は KL 1 の組込み述語であり，n 本のストリーム $X_1, \cdots, X_n$ をマージした出力ストリーム X_0 を生成する．

モード解析により，最初の k 個の入力引数についてはそれぞれの参照パターンが抽出されている．i 番目 $(1 \leq i \leq k)$ の引数の参照パターンを $T_{i1}, \cdots, T_{ie_i}$ とすると，各 $T_{ij}(1 \leq i \leq k, 1 \leq j \leq e_i)$ に対応して $D_{ij}(1 \leq i \leq k, 1 \leq j \leq e_i)$ が生成される．D_{ij} は i 番目の引数の j 番目の参照パターンに対応した色付きベクタの監視を行い，そこに色付きベクタが出現した時に *p_EACH* を起動してベクタの処理にあたらせる．$T_{ij}=(X_i, A_i)$ とすると D_{ij} は図 4 のようになる．

```
C_i' = { p(X_1,...,X_N,Col,Ids) :-                    if b ≥ 2
           G_1,...,G_a |
           merge({Ids_1,...,Ids_b},Ids),
           q_1(Y_1,...,Y_{i_1},Col,Ids_1),
             ⋮
           q_b(Z_1,...,Z_{i_b},Col,Ids_b),
           E_1,...,E_c.
       p(X_1,...,X_N,Col,Ids) :-                    if b = 1
           G_1,...,G_a |
           q_1(Y_1,...,Y_{i_1},Col,Ids),E_1,...,E_c.
       p(X_1,...,X_N,Col,Ids) :-                    if b = 0
           G_1,...,G_a | Ids = [ ],E_1,...,E_c.

D_ij = p(X_1,...,A_i,...,X_N,Col,Ids) :-
         vector(X_i,1),vector_element(X_i,0,X_i'),vector(X_i',L) |
         trim_vector(L,X_i',Col,L',X_i''),
         (L' =:= 0 →
             Ids = [ ],
             X_{k+l+1} = $void$,...,X_N = $void$;
          L' > 0 →
             new_vector(Y_1,L),
               ⋮
             new_vector(Y_m,L),
             p_EACH(X_i'',{ρ,A_i},0,i,X_v,Z_v,Ids).
       ただし
         X_v = p(X_1,...,X_{k+l},Y_1,...,Y_m)
         Z_v = {X_{k+l+1},...,X_N}

D_{k+1} = p(X_1,...,X_N,Col,Ids) :-
            true |
            X_{k+l+1} = $void$,...,X_N = $void$,Ids = [ ].
```

図 4 AND 述語の変換後の KL 1 コード
Fig. 4 KL 1 codes of a transformed AND predicate.

この節は，まずガード部で i 番目の引数 A_i の中の X_i が色付きベクタかどうかをサイズ 1 のベクタかどうかの検査 ($vector(X_i, 1)$) で調べ，そうである時にはその中身のベクタ X_i' をサイズ L とともに取り出す．次にシステムで用意している述語 *trim_vector* を使って X_i' の要素のうち当該ゴールアトムの色 Col と直交しない色を持つ要素だけを抽出し，そのリストを X_i''，その数を L' に具体化する．そして，$L'=0$ の時は何もせずに処理を終了し，$L'>0$ の時は，サイズ L' のベクタを出力モードをもつ引数の数だけ新しく生成し *p_EACH* を起動する．この新たに生成されたベクタは，生成時にはまだ全要素が 0 で初期化されているだけである．それらはこの後の *p_EACH* における処理の中で入力された色付きベクタの各要素に対応した出力変数のコピーを要素とする色付きベクタとなって最終的に各出力変数を具体化する．

D_{k+1} は上のどれにもコミットできない場合を扱う．このことは D_{k+1} の直前に置かれた *otherwise* により保証される．ここでの処理はすべての出力引数を *$void$* で具体化することである．

p_EACH は次の形で呼び出される．

$p\_EACH(X'', \{\rho, A\}, J, I, X_v, Z_v, Ids)$

ただし，

I＝現在処理中の入力引数の番号

J＝色付きベクタ中の現在処理中の要素番号

X''＝色付きベクタの中身をリストに詰め替えたもの

$\{\rho, A\}$＝ゴールの I 番目の引数 A とその中の色付きベクタの出現位置を示すパス情報 ρ の対

$X_v = p(X_1, \cdots, X_{k+l}, Y_1, \cdots Y_m)$

$Z_v = \{X_{k+l+1}, \cdots, X_N\}$

p_EACH の役割は検出された色付きベクタの各要素（第一引数のリストの各要素）に p を適用するということである．*p_EACH* は図 5 のように二つの節 F_1, F_2 で定義される．

p_EACH は第一引数に色付きベクタをリストに変換したものをもち，その要素を順番に処理する．F_1 はそのリストが空になって終了する場合を記述している．重要なことは元の出力変数にここで初めて色付きベクタが具体化されるということである．

F_2 では要素 $v(E, Col)$ を取り出すと（これは J 番目の要素である）まず出力用に新たに生成した各ベクタの J 番目の要素に新しい変数 Y をもつ $v(Y, Col)$ を埋め込み，残りの要素の処理をする *p_EACH* を呼び出す．次に E の値を調べ，それが *$void$* のときは各 Y を *$void$* に具体化して処理を終了し，そうでないときは以下で述べるような引数を持った p を起動する．入力引数については I 番目を除くと元のものと同じ．I 番目はシステムが用意した述語 *melt* により A の中の ρ で示される部分に E を埋め込んだもの A'．出力引数についてはそれぞれ対応する Y を使う．

4.3.2 OR 述語の変換

アリティ N の OR 述語 p の変換を考える．p の

```
F_1 = p_EACH([ ],CP,J,I,p(X_1,...,X_N),{Z_1,...,Z_m},Ids) :-
        true |
        Ids = [ ],Z_1 = {X_{k+l+1}},...,Z_m = {X_N}.

F_2 = p_EACH([v(E,Col)|Ls],CP,J,I,p(X_1,...,X_N),Z_v,Ids) :-
        true |
        set_vector_element(X_{k+l+1},J,_,v(Y_1,Col),M_1),
          ⋮
        set_vector_element(X_N,J,_,v(Y_m,Col),M_m),
        J_1 := J + 1,
        p_EACH(Ls,CP,J_1,I,p(X_1,...,X_{k+l},M_1,...,M_m),Z_v,Ids_1),
        (E = $void$ →
            Ids_1 = Ids,Y_1 = $void$,...,Y_m = $void$;
         otherwise;
         true →
            merge({Ids_1,Ids_2},Ids),randor:melt(CP,E,A'),
            p(X_1,...,X_{I-1},A',X_{I+1},...,X_{k+l},Y_1,...,Y_m,Col,Ids_2).
```

図 5 *p_EACH* の定義
Fig. 5 Definition of *p_EACH*.

定義を $C_1, \cdots, C_n$ とし，以下のモード宣言をもつとする．

:- mode $p(\underbrace{+, \cdots, +}_{k}, \underbrace{-, \cdots, -}_{l})$.

ただし，$k+l=N$ である．p/N は C_0' を定義節として持つ $p/(N+2)$ へと変換される．**図6**に OR 述語の変換の模様を図示する．増えた二つの引数は前と同様に，それぞれ色と ID サーバとの通信チャネルを保持する．C_0' は $C_1, \cdots, C_n$ の各節に対応した n 個のゴール $p\_1/(N+2), \cdots, p\_n/(N+2)$ を起動する．

$p\_i/(N+2)(1 \leq i \leq n)$ の定義は，p_i/N を C_i だけを定義節（ガード部は空とみなす）とする AND 述語としてコンパイルすることによって得られる．

$p/(N+2)$ は**図7**のように定義される．まず ID サーバに *get_bp(Bp)* というメッセージを送り，新しい OR 述語の起動の識別子 *Bp* を得，それをもとに n 個の定義節に対応した n 個の新しい色 $Col_1, \cdots, Col_n$ を一連の *add_color* により生成する．またサイズ n の新しいベクタを出力引数の数 l だけ作り，それぞれを一連の *set_vector_element* で具体化し，最終的に元の出力引数 $X_{k+i}(i=1, \cdots, l)$ を以下のように具体化する．

$X_{k+i} = \{\{v(Z_{i1}, Col_i), \cdots, v(Z_{in}, Col_i)\}\}$ $i=1, \cdots, l$

これと並行して，n 個のゴール $p\_1, \cdots, p\_n$ を起動する．このときゴール p_i には新しく生成した色 Col_i を与え，またその出力引数には $Z_{1i}, \cdots, Z_{li}$ を与える．

4.3.3 実行時オーバヘッドに関する考察

ごく小規模な例題を用いて ANDOR-II プログラムのオーバヘッドを調べてみる．用いた計算機はマルチ PSI であるが，一台の PE しか使用してない．

例題は A，B の二つである．A はリストの append を繰り返し行うものであり（**図8**），B は一本のリストを二つに分割するものである（**図9**）．A は完全に決定的な問題であり，A の ANDOR-II プログラムは AND 述語だけからなる．すなわち，

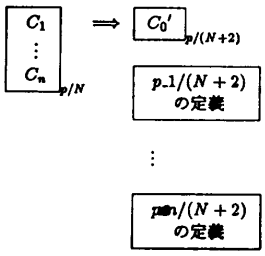

図6 OR 述語の変換
Fig. 6 Transformation of an OR predicate.

KL1 と ANDOR-II ではモード宣言があるかないかだけでしか違わない．一方 B は don't know 非決定的な問題である．ANDOR-II ではこの問題は append プログラムを or 述語として宣言し，モードを逆にし，頭部とボディ部で同一化を若干交換するだけで良い．一方 KL1 プログラムとして文献17）にある六つの節からなるプログラムを示す．KL1 プログラムの方はすべての解を求めることを明示的に記述しているが，

```
p(X1,...,XN,Col,Ids) :-
    true |
    Ids = [get_bp(Bp)|Idss],
    merge({Ids1,...,Idsn},Idss),
    new_vector(Y10,n),...,new_vector(Yl0,n),
    add_color(Col,Bp,1,Col1),...,add_color(Col,Bp,n,Coln),
    set_vector_element(Y10,0,_,v(Z11,Col1),Y11),
    set_vector_element(Y11,1,_,v(Z12,Col2),Y12),
     :
    set_vector_element(Y1(n-1),n-1,_,v(Z1n,Coln),Y1n),
     :
     :
    set_vector_element(Yl0,0,_,v(Zl1,Col1),Yl1),
    set_vector_element(Yl1,1,_,v(Zl2,Col2),Yl2),
     :
    set_vector_element(Yl(n-1),n-1,_,v(Zln,Coln),Yln),
    p_1(X1,...,Xk,Z11,...,Zl1,Col1,Ids1),
     :
    p_n(X1,...,Xk,Z1n,...,Zln,Coln,Idsn),
    Xk+1 = {Y1n},...,XN = {Yln}.
```

図7 OR 述語の変換後の KL1 コード
Fig. 7 KL1 codes of a transformed OR predicate.

ANDOR-II によるリストのアペンドの定義

```
:- mode append(+,+,-).
append([],Y,Z) :- true | Z=Y.
append([A|X],Y,Z) :- true | Z=[A|Zz], append(X,Y,Zz).
```

KL1 によるリストのアペンドの定義

```
append([],Y,Z) :- true | Z=Y.
append([A|X],Y,Z) :- true | Z=[A|Zz], append(X,Y,Zz).
```

図8 例題 A：リストのアペンド
Fig. 8 Example A: Appending lists.

ANDOR-II によるリストの分割の定義

```
:- or_predicate app/3.
:- mode app(-,-,+).
app(X,Y,Z) :- X=[], Y=Z.
app(X,Y,[A|Z]) :-  X=[A|Xx], app(Xx,Y,Z).
```

KL1 によるリストの分割の定義

```
% calling form ... app(Z,'L0',S,[])
app(Z,Cont,S0,S2) :- true | app1(Z,Cont,S0,S1), app2(Z,Cont,S1,S2).
app1(Z,Cont,S0,S1) :- true | cont(Cont,[],Z,S0,S1).
app2([A|Z],Cont,S0,S1) :- true | app(Z,'L1'(A,Cont),S0,S1).
app2(Z,_,S0,S1) :- otherwise | S0=S1.
cont('L1'(A,Cont),X,Y,S0,S1) :- true | cont(Cont,[A|X],Y,S0,S1).
cont('L0',X,Y,S0,S1) :- true | S0=[(X,Y)|S1].
```

図9 例題 B：リストの分割
Fig. 9 Example B: Splitting a list.

ANDOR-II の記述は一つの解の求め方だけである．さらに，KL1 は全解をリスト形式で集めるコードを含んでいるが，ANDOR-II プログラムの方は全解を集める機能はトップレベルのインタプリタが提供しているのでそのようなコードを含まない．一般にBのような don't know 非決定的問題では，二つの言語でプログラムが大きく異なる．記述力に関していうとこのような問題では ANDOR-II の方が格段に優れている．

実行時間の比較を表1に示す．例題Aに関して見ると，KL1 と ANDOR-II ではリダクション数，実行時間ともにあまり変わらないことが分かる．わずかな差は ANDOR-II プログラムに含まれるプロセッサ割り付け用のコードのためである（このプログラムではプロセッサ割り付けは行わないのであるが，現在のコンパイラは割り付けのための乱数管理用のコードを付加してしまう）．すなわち AND 述語に関する限り現在の変換方式は余分なオーバヘッドを生成していないといえる．

一方例題Bを見ると

- ANDOR-II ではリダクション数が KL1 の1割程度増えている
- 一方，実行時間に関しては ANDOR-II は KL1 の 2.47 倍である

という点が注意を引く．KL1 と比べて，ANDOR-II の計算で増えた計算は明らかに色の処理である．そこで，近似的に

ANDOR-II 版の計算＝KL1 版の計算＋色の処理

と仮定してみる．そうすると，色の処理はリダクション数でいうと1割程度であるが，時間で見ると KL1 版の計算の1.5倍近いボリュームであるということになる（しかし，元の計算がリスト処理というかなり軽い仕事であることに注意）．これは現在の処理系における色処理のユニフィケーションというものがかなり重いということを示している．これの原因の一つは色表現するデータのサイズであろう．

こうした面についてハードウェアからどのようなサポートが可能であるかという検討をする必要がある．またこのような実行時間，そしてメモリ量に関する不利を記述力という面でどれくらいカバーできるかについてもう少し規模の大きな問題で検討する必要がある．

また，変換されたコードの大きさは元の大きさに比例すると考えて良い．しかし，変換されたコードの大きさに影響する他の要因として，参照専用モード引数の数があることに注意されたい．一般に頭部同一化で詳細なマッチングを行えば行うほどそれに対応した変換コードは大きくなる．

表1 ANDOR-II プログラムの実行時オーバヘッド
Table 1 Runtime overhead of ANDOR-II programs.

プログラム	A	B
KL1		
リダクション数 (R)	905005	463555
時間 (S) sec	14.822	4.939
R/S	61K	94K
ANDOR-II		
リダクション数 (R)	922177	505756
時間 (S) sec	15.075	12.190
R/S	61K	41K
ANDOR-II/KL1		
リダクション数	1.019	1.091
時間 sec	1.017	2.468

5. 他の研究との比較

AND OR 並列性を備えた記述を AND 並列の記述へと変換するという研究の先例には Ueda[18], Tamaki[15] の仕事がある．

Ueda は AND OR 並列の論理プログラムのうち入出力がグラウンドで AND 並列に関してはコルーチンのみを備えた論理プログラムの AND 並列への変換法を提案した[18]．この方式は逐次実行の実装などで用いられる継続に基づいた変換である．事前にソースプログラムを静的に解析することにより実行時のコルーチンのスケジューリングを完全に決定し，そこから逐次的な実行列を抽出し，これを継続によりコード化する．しかし，一般の並列プログラムに対してスケジューリングを完全に決定するような静的解析はあり得ない．この方式は効率の良いプログラムを生成するが，ごく限られた状況でしか適用できない．

Tamaki はコルーチンより一般的な AND OR 並列の論理プログラムの変換にについて報告している[15]．このプログラムは Ueda のと同様にグラウンド入出力を前提としているが，アトムの論理積が変数を共有しないときに並列実行が行えるようになっている．この変換法では非決定的なゴールアトムからの解の集合をストリームの形で他のゴールに伝搬する．このため，動的スケジューリングが容易となり，ソースプログラムの AND および OR 並列を自然に実現することが

できる．この手法はわれわれの方式に近いが，重要な差異がある．それはTamakiの言語は変数を共有する論理積の並列実行を許さないのに対し，ANDOR-IIはそれを許すという点である．アトムの論理積が変数を共有できるかどうかは，並列事象の記述という観点から見ると，並列事象間で相互作用が可能かどうかに相当し，われわれの言語の目的からはこの相互作用はなくてはならないものである．われわれは，ANDOR-IIにおいて，変数を共有する論理積の並列実行が色という概念を導入することにより並列論理型言語に埋め込めることを示した．

また，これらの二つの変換法ではソースプログラムについてグラウンド入出力という仮定をしているが，本論文ではこれに代わりもっと一般的な「参照専用かそれ以外か」というモードを提案した．すなわち，入力は，それが内部で参照しかされない場合はグラウンドでなくても許される．

D. H. D. Warrenによって最近提起された論理プログラムの新しいAND OR並列計算モデルにAndorraモデルがある[6]．Andorraモデルは決定的にリダクションが行える場合は変数を共有する論理積中のゴールアトムも並列にリダクションすることができるという考え方に基づいている．Andorraモデルでは，実行は二つのフェーズを交互に繰り返す．決定的フェーズでは，すべての決定的ゴールは並列にリダクションされる．決定的ゴールがなくなったときは，実行は非決定的フェーズに入る．非決定的フェーズでは，最左ゴールが選択され，世界が候補節に沿っていくつかの世界に分岐する．Pandora[1]は並列論理型言語PARLOG[2),5)]に基づいたAndorraモデルの一つの実現である．この点でPandoraはANDOR-IIに良く似たアプローチをとっているといえるが，両者のベースとなっているAND OR並列計算モデルが異なるため，両者の中身は大きく異なる．Pandoraでは，Andorraモデルに忠実に従い，OR分岐は非決定的フェーズで一回だけ行い，しかもその時に世界を一括してコピーする．すなわち，Pandoraでは，分岐はなるべく後まわしにし，分岐がどうしても必要になった時は世界をアトミックに(非分割に)コピーしてしまうという戦略をとる．これと対照的にANDOR-IIは，分岐は必要に応じて随時行い，代わりに世界のコピーを段階的に行うという戦略をとっている．この結果，ANDOR-IIのほうが，OR分岐に由来する並列性が高く，また，コピー操作に関連したアトミック操作が小さくて済むため，粒度の小さい高並列な計算が実現される．

6. おわりに

現在ANDOR-IIシステムはマルチPSI上に実装されている．また実装に際しては継続ベースの変換方法[17]との組み合わせも行っている．このほかにも実用的な問題解決記述言語であるために次のような拡張を施している．

- otherwiseおよびalternatively
  KL1で採用されている節の非対称的扱いを可能にするプリミティブotherwiseとalternativelyをAND述語に導入した．
- モジュール
  KL1で採用されているのと同様のモジュールを導入した．
- ランダムなプロセッサ割り付け
- OR世界間通信

今後に残された問題がいくつかある．その中でもっとも肝心なのは本格的な応用による評価である．また，効率の改善のために，二つの色の直交性テストを一定時間に行えるようなハードウェアサポートとそれに合致した新しい色表現方式の研究も重要な課題である．

謝辞 本研究を行うに当たりご指導をいただいた新世代コンピュータ技術開発機構古川康一研究次長，長谷川隆三第5研究室室長にここで深く感謝いたします．この研究は第5世代コンピュータプロジェクトの一環として行われた．

参考文献

1) Bahgat, R. and Gregory, S.: Pandora: Nondeterministic Parallel Logic Programming, Levi, G. and Martelli, M. eds., *Logic Programming, Proceedings of the Sixth International Conference*, pp. 471-486, The MIT Press (1989).
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) Clark, K. L. and Gregory, S.: PARLOG and PROLOG United, Lassez, J.-L. ed., *Logic Programming, Proceedings of the Fourth International Conference*, pp. 927-961, The MIT Press (1987).
4) Conery, J. S. and Kibler, D.F.: AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing*, Vol. 3, No. 1, pp. 43-70 (1985).

5) Gregory, S.: *Parallel Logic Programming in PARLOG*, Addison-Wesley (1987).
6) Haridi, S. and Brand, P.: Andorra Prolog, an Integration of Prolog and Committed Choice Languages, Institute for New Generation Computer Technology ed., *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pp. 745-754, Institute for New Generation Computer Technology, OHMSHA, LTD. (1988).
7) Haridi, S. and Janson, S.: Kernel Andorra Prolog and Its Computation Model, Research Report, Swedish Institute of Computer Science (1989).
8) ICOT PIMOS Development Group: PIMOS Manual, Institute for New Generation Computer Technology, 2.5 ed. (Jan. 1991) (in Japanese).
9) Kowalski, R. A.: *Logic for Problem Solving*, Elsevier, North-Holland (1979).
10) Naish, L.: Parallelizing NU-Prolog, Kowalski, R. A. and Bowen, K. A. eds., *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pp. 1546-1564, The MIT Press (1988).
11) Saraswat, V. A.: The Concurrent Logic Programming Language CP: Definition and Operational Semantics, *Proceedings of the Symposium on Principles of Programming Languages*, pp. 49-62, ACM (1987).
12) Takeuchi, A.: On an Extension of Stream-Based AND-Parallel Logic Programming Languages, *Proceedings of the 1st National Conference of Japan Society for Software Science and Technology*, pp. 291-294, Japan Society for Software Science and Technology (1984) (in Japanese).
13) Takeuchi, A. and Takahashi, K.: An Operational Semantics of AND- OR-Parallel Logic Programming Language, ANDOR-II, *LNCS 491, Concurrency: Theory, Language, and Architecture—UK/Japan Workshop, Oxford, UK, September 1989 Proceedings*— (1991). Also appearing as ICOT Tech. Report TR-511 (1990).
14) Takeuchi, A., Takahashi, K. and Shimizu, H.: A Parallel Problem Solving Language for Concurrent Systems, Tokoro, M., Anzai, Y. and Yonezawa, A. eds., *Concepts and Characteristics of Knowledge-based Systems*, pp. 267-296, North Holland (1989). Also appearing as ICOT Tech. Report TR-418 (1988).
15) Tamaki, H.: Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages, Lassez, J.-L. ed., *Logic Programming, Proceedings of the Fourth International Conference*, pp. 376-393, The MIT Press (1987).
16) Ueda, K.: Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, Tech. Report TR-208, Institute for New Generation Computer Technology (1986).
17) Ueda, K.: Making Exhaustive Search Programs Deterministic, *New Generation Computing*, Vol. 5, No. 1, pp. 29-44 (1987).
18) Ueda, K.: Making Exhaustive Search Programs Deterministic, Part II, Lassez, J.-L. ed., *Logic Programming, Proceedings of the Fourth International Conference*, pp. 356-375, The MIT Press (1987).
19) Yang, R. and Aiso, H.: P-Prolog: A Parallel Logic Language Based on Exclusive Relation, Shapiro, E. ed., *LNCS-225, Third International Conference on Logic Programming*, pp. 255-269, Springer-Verlag (1986).

(平成3年3月20日受付)
(平成3年11月5日採録)

竹内 彰一 (正会員)

1953年生. 1977年東京大学計数工学科卒業. 1979年同大学院修士課程修了. 同年三菱電機(株)入社. 以来, 人工知能, 論理プログラミングの研究に従事. この間, 1982～1986年(財)新世代コンピュータ技術開発機構に出向. 1991年4月より(株)ソニーコンピュータサイエンス研究所. ACM, IEEE, 日本ソフトウェア科学会各会員.

高橋 和子 (正会員)

1958年生. 1982年京都大学理学部卒業. 同年三菱電機(株)入社. 現在同社中央研究所勤務. 時制論理に基づくシステムの記述と推論方式に関する研究, 並列論理型言語による問題解決手法の研究に従事. 電子情報通信学会, 日本ソフトウェア科学会各会員.

坂本 忠昭 (正会員)

1962年生. 1985年大阪大学工学部通信工学科卒業. 1987年同大学院修士課程修了. 同年三菱電機(株)入社. 以来, 同社中央研究所において, 並列論理型言語, 論理プログラムの帰納学習や帰納検証などの研究に従事. 日本ソフトウェア科学会会員.