TM-0080

日本ソフトウェア科学会第１回大会

発 表 論 文 集

（９件）

ICOT ならびに再委託先メーカ

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

Mandala Ⅱの構想

古川　康一、竹内　彰一、隈藤　進、大木　優、宮崎　敏彦、安川　秀樹

（財）新世代コンピュータ技術開発機構

1．はじめに

Mandala は、ストリーム並列実行を基本とする論理型プログラミング言語KL1 上に開発されつつある知識プログラミング言語である。KL1 は、第五世代コンピュータ・プロジェクトの中で設計を進めている言語で、Concurrent Prolog 、Prologなどがその基本になっている。

Mandala は、知識プログラミングとシステム・プログラミングの両者を念頭に置いて設計が進められている。それらの実行時の扱いは勿論異なり、知識プログラムは、解釈実行が基本となり、システム・プログラムはコンパイル実行が基本となると考えている。

本稿では、特に知識プログラミング機能に注目し、その拡張の方向について述べてみたい。知識プログラミング機能は、推論機能と知識ベース管理機能の大別される。推論機能の拡張としては、ユーザが任意の推論エンジンを定義することのできる機能を取り上げて述べたい。この機能は、Mandala 自身に拡張性を持たせることになり、推論機能を一般化することに相当する。一方、このような一般化は常に効率の低下をもたらすが、それを回避するために推論機能の特殊化、専用化することを考えている。この点についても合わせて述べたい。

知識ベース管理機能については、論理的あるいは物理的な構造物の表現について考察をしてみたい。とくに、コンストレイント・プログラミングとの関連について述べてみたい。

さらに知識ベース管理システムの実現法についても述べてみたい。その中でもとくに、知識ベースの検索、および更新に伴う無矛盾性・冗長性管理の２つの機能に焦点をあてて論じたい。

2．推論機能の拡張

Mandala の基本構成要素は、知識を格納する単位世界と個々の具体的な物を表す実体の２つである。実体はメッセージを受けとると自分がかかえている局所状態と自分のテンプレートである単位世界にある知識を用いて、そのメッセージの処理を行う（テンプレートと実体は、 instance-of リンクで結ばれている）。さらに、複数の単位世界がis-aリンクによって階層化されており、ある単位世界の上位概念（たとえば「ホ乳類」は「人間」の上位概念）にある知識は下の単位世界でも成り立つものと見なされて、性質の継承が行われる。

ここで、推論エンジンの役割は、メッセージを処理するために知識をどのように使うか、その使い方を指定することである。

推論エンジンを差し替え可能にすれば、ユーザが自分で問題に応じた推論エンジンを作ることによって、Mandala 自身を各問題に適したシステムに変更することが可能となる。その意味で、ユーザが推論エンジンを定義できる機能は、システムの拡張性を得るためになくてはならない機能である。

推論エンジンを差し替え可能とするためには、差し替えのためのインタフェースを定めることが必要である。Mandala における実体の表現は、図１の通りであるが、ここでsimulate(Name, Message, State, NewState)がConcurrent Prolog インタプリタsimulateへの呼び出しであり、それによって与えられたMessage が処理されることを表す。ここで、Concurrent Prolog のインタプリタsimulateがこのメッセージを処理するための推論エンジンとして使われているが、それを他のものに置きかえるには、simulate(Name, Message, State, NewState)をインタフェースとして、各問題に応じた問題解決機を定義すればよい。一般の問題解決機へのインタフェースとしては、"simulate"という名前はふさわしくないので、その代りに"solve" という名前を使うようにする。

```
instance(Name,[Goal | Input],World) :-
    simulate(Name,Goal,World),
    instance(Name,Input?,World).
```

図１　実体の表現

このようなインタフェースをもった推論エンジンをユーザが定義して使えるようにすればよい。実体の生成プログラムは、メタ・レベルの知識として定義されるが、その定義自身を、ユーザが行えるようにすれば、ユーザ定義推論エンジンが容易に実現できる。より詳しい実現方法については、別稿を参照されたい。

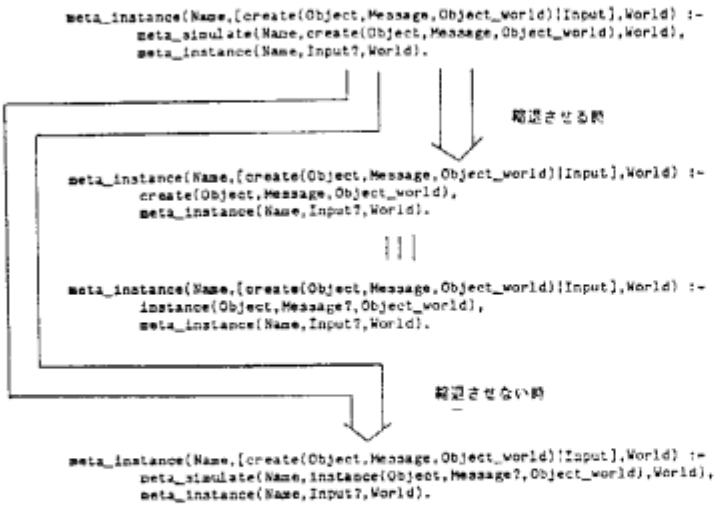


図2　2レベル推論の縮退

3．推論機能の特殊化

前節では、推論機能の一般化、拡張性について述べたが、その逆である特殊化も重要である。特殊化によって、処理効率の向上が図られるからである。特殊化は、最適化、コンパイルなどと深く係わっている。

特殊化の例としては、オブジェクト・レベルの推論エンジンがConcurrent Prolog インタプリタであるときの、メタ推論とオブジェクト推論の縮退がある。

すなわち、メタ・レベルでオブジェクト・レベルの実体を作り出すとき、その実体の推論エンジンがメタ・レベルの動作を司っている推論エンジンと同一である場合、その実体は、図2のようにメタ推論のレベルで動くプログラムとして生成される。

メタ推論エンジンを使わずにオブジェクト・レベルの実体をメタ・レベルで動かすことによって、ユーザ定義推論エンジンを使うことはできなくなるが、メタ・レベルのインタプリタの介在がなくなる分、効率のよい実行が可能になる。

一般にMandala では多くの機能をメタ・レベルで実現している。例えば個々の実体のbehaviour はオブジェクト・レベルで定義されているが、実体そのものはメタ・レベルで再帰的なプログラムとして定義されている。またMandala における属性継承はメタ・レベルの述語solve により記述されている。さらにユーザ定義推論エンジンの記述もメタ・レベルで行われる。Mandala の基本思想はオブジェクト・レベルとメタ・レベルを明確に区別し、その枠組の中で実体の抽象的表現を与えたり、単位世界間の継承関係を抽象的に記述することであると同時に、ユーザ定義推論エンジンの定義や知識の同化などの新しい概念をこの枠組の中で実現することにある。

しかし、このオブジェクト／メタ・レベルの明確な区別およびメタ・レベルにおける強力なプログラミング（メタ・プログラミングと呼ぶ）は実行効率の観点から見ると決して満足のいくものではない。我々はこの実行効率を改善する技術が部分計算であると考え、検討を進めている。言いかえると、Mandala の強力な表現力の１つはメタ・プログラミングが与えており、その実行効率を支えるのが部分計算であると考えている。

部分計算とは二村の定義では

「稼働環境に関する情報を利用して、汎用のプログラム　　をより効率の良いプログラムに特殊化すること」

である。部分計算はコンパイラの自動生成などに使われ、すでに有効であることが確かめられている。Mandala における部分計算の利用は、オブジェクト・レベルとメタ・レベルに分けられているプログラムを部分計算により効率の良い１レベルのプログラムに変換することを目的として行われる。ここで必要となる部分計算は汎用のものではなく、Mandala に特殊なもので良い。前述の例はオブジェクト・レベルの推論エンジンとそれを実行するメタ・レベルの推論エンジンが同一であるとき、これを１つのエンジンに縮退させるものであったが、これも部分計算の一例である。この他に現在検討している部分計算の対象は、次のものである。

(1) World （オブジェクト・レベル）が与えられたときのinstance述語のプログラム（メタ・レベル）の部分計算。

(2) 解くべきゴールとプログラム（オブジェクト・レベル）が与えられたときのsolve 述語のプログラム（メタ・レベル）の部分計算。

4．構造物の表現

Mandala で構造物を表現する手段としては、部分－全体の関係を表すpart-of リンクが提供されている。part-of リンクは概念レベルでは、２つの単位世界を関連づける論理式（「タイヤは車の部品である」といったような式）として表されている。そしてその論理式は“全体”を表す単位世界にメタ知識としておかれる。このメタ知識は、実体の生成時にその生成を司る“管理者”実体によって参照され、全体を表す実体とともに部分を表す実体も同時に作られる。

実体間のpart-of リンクは、概念レベルのそれと異なり、

全体から部分への通信チャネルとして実現される。

以上の機能を用いると、部分は常に全体の作成時に新たに作られるので、すでにある実体を部品として使うことができない。しかしながら、部分ー全体の例として、場面の設定とそこでの登場人物を考えてみると、登場人物は入れ替わり得るし、すでに存在する実体と考えた方が適切であるので、そのような機能は必須である。たとえば、フライトの側では、フライトを指定するとそこで用いられる飛行機の機体、パイロット、スチュワーデスなどが決まるわけである。そして、その同じパイロットが家庭での父親の役割を演ずることになる。

このような、既存の実体を部分とするような全体を作るには、通信チャネルの動的な行進機能が必要となるが、それはすべての実体を管理するgeneral manager を介して部分となる実体の入力チャネルを取り出し、それと新たに設定する全体から部分への通信チャネルをマージした新たな通信ネットワークを設定すればよい。

構造物の表現にとって、大切な点は、部品間の関係の記述である。この関係には、部品間の位置関係、接続関係などがあるが、これらは単なる性質の記述にとどまらず、構造物に対する各種の操作に対する制約条件として働く。たとえば、構造物を移動させたとき、その中の部分は、位置関係を保存するように一緒に動かなくてはならない。

この種の処理はコンストレイント・プログラミングの目指しているものである。その中でもsteeleのconstraintsはとくに有名であるが、線形連立不等式の処理をユニフィケーションに組み込むColmerauerの提案も興味深い。

5．Mandala による知識ベース管理システムの実現法

知識プログラミング・システムMandala は、単位世界に現れるプログラムを知識と見なすことによって、知識ベース管理システムの基本的枠組を与えていると考えることもできる。

知識ベース管理システムの基本機能として、知識表現、知識利用、知識獲得などの諸機能が考えられるが、知識表現の土台は、Mandala の基本機能に含まれているので、ここでは知識利用および知識獲得の両機能がどのように実現されるかを示そう。

5．1　知識ベースの検索

知識の利用は、知識ベースからいかに必要とする知識を検索するかにかかっている。ゆえに、ここでは、知識ベースの検索機能に絞って話を進めよう。

Mandala では、あるまとまった単位の知識は１つの単位世界を用いて表される。それは、たとえば関係データベースでの１つの関係に対応すると考えればよいであろう。単位世界にある情報を取り出すためには、その単位世界専用の質問処理を行う司書を置くことが必要である。そのような司書はその単位世界の実体として実現できる。すなわち、単位世界を知識ベースとして考えたとき、instance_ofリンクで結ばれた実体は、その単位世界によって表される実体と考えるよりも、むしろその知識ベースを検索する司書と考えた方がよい。

ただし、司書としての実体がもつ推論エンジンはConcurrent Prolog ではなくPure Prolog である。

知識ベースに専用の司書を置くと、その司書に知識ベースの一時的な更新機能を持たせることができる。その更新された知識は司書自身が保有していて、元の知識ベースは変化しない。この機能を用いて、仮定に基づく推論が実現され、さらに拡張して、いくつもの司書に別の仮定を置くことによって、同時に多くの世界を仮定してどの仮定が最も妥当であるかを調べるような問題を扱うことが可能となる。たとえば、HYCIN では、ある感染症を仮定したときの妥当性を、すべての感染症について計算し、その中で最も確度の高い仮定から順に、いくつかを結論として選択しているが、これはまさにその種の問題と言えよう。

知識ベースの局所的更新は、司書自身が持っている状態の変化として実現できる。これは、知識ベース管理者が行う全域的更新とは異なることに注意しよう。全域的更新では、実際に単位世界自身を書き替えてしまう。そのため、その種の更新の方がシステム全体に及ぼす影響はずっと大きく、厳重な検査が必要となる。次節では、この問題を取り上げよう。

5．2　知識ベース管理と知識の同化

知識ベースの管理のための知識獲得機能としては、種々のレベルのものが考えられるが、論理プログラミング言語との整合性から見て、論理としての無矛盾性および非冗長性の維持を図りながら知識を獲得していく同化の機能を取り上げて考えることにする。

われわれは、通常の逐次実行に基づくPrologの上で無矛盾性管理および冗長性除去を行う知識同化プログラムを実現した。そのプログラムは、知識を個々の実体に関する具体的事実を与える肯定的知識と、そのような具体的事実が満足しなければならない条件を与える否定的知識に分け、新たな肯定的知識を獲得するときに否定的知識を調べるという方法を採っている。しかしながら、逐次実行に基づくProlog上の知識同化プログラムは、実行効率の点で必ずしも現実的でないことが明らかとなった。すなわち、無矛盾性について言えば、知識を獲得するた

びにすべての否定的知識についてそれらが成り立っていることを示さなければならないとすれば、その実行時間は否定的知識の量に比例することになる。また冗長性除去のアルゴリズムは、1つ1つの肯定的知識の冗長性を調べるので、その実行時間は肯定的知識の量に比例する。

われわれは、Mandala 上で、より実行効率のよいアルゴリズムを実現する見通しを得た。第1に、無矛盾性の検査についていえば、肯定的知識および否定的知識を適当に分割管理することによって、1つの肯定的知識を獲得する際に調べる必要のある否定的知識の量を減らすことができる。また、各否定的知識の検査自身はPrologのゴール文の証明によってなされるがその実行自身、並列化が可能であることも指摘しておきたい。

第2に、冗長性除去について言えば、制限された範囲内であれば、各肯定的知識について独立にその冗長性を調べ、その後で冗長であると判明したすべての知識を除去することができる。いま、含意関係"→"によってグラフを作ったとき、閉じたループがないと仮定し、知識ベースTからPとQが独立に冗長であることが示されたとする。冗長性は、たとえばT－PからPが導かれることによって示されるが、含意関係のループがないとしたので、Pの証明にQが使われると同時にQの証明にPが使われるようなことはない。ゆえに、このような場合には上に述べた並列アルゴリズムによる冗長性除去が可能となる（このような条件が満たされない例としては、A，B．A→B，B→Aを考えればよい）。そして、一般に無限ループに陥らないようなPrologあるいはConcurrent Prologプログラムは、含意関係のループがないので、この手法が適用できる。

並列化による実行速度の向上は、Concurrent Prolog を並列に実行できる計算機の出現が必要条件であり、それはこれからの研究に待たなければならない。が、少くともそのような並列化の有用性を示しているものと言えよう。知識同化プログラムをMandala 上で実現する方法の概略を、図3に示す。

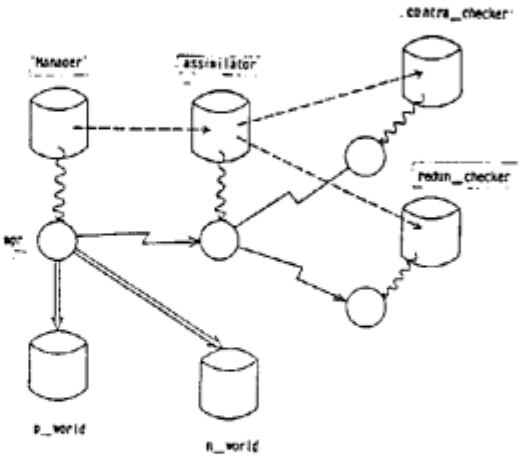


図3　知識同化プログラム

この図から分かるとおり、知識ベース管理プログラム'Manager' は、その部品としてassimilator を持つように拡張されている。assimilator 自身は2つの部品モジュールcontra_checker と redun_checker を持つ。'Manager' の実体mgr は、 p_world および n_world と呼ばれる2つの単位世界を管理している。 p_world は肯定的知識から成り、 n_world は否定的知識から成る。否定的知識は、データベースの用語で保全性制約条件(Integrity Constraint)と呼ばれているものであり、それは、いくつもの異なる p_world に共通して使える。それが、 n_world を p_world と分離して管理する理由である。Smalltalk80 にも、ここで述べた管理者の考えが使われており、単位世界に対応するクラスの1つ1つにそれを管理するためのオブジェクトが付随している。しかし、その対応は1対1であり、しかもその拡張性が乏しい。Mandala においても、原則的には、知識ベース管理者は特定の単位世界を管理することを考えているが、しかしそれは何も1つに限ったことではない。管理対象を1つに限ったとしても、図3のように、2つの異なる単位世界を参照していることになる。Mandala の優位性は、このような柔軟性にあると言ってよい。

6．　参考文献

Furukawa, K. et al.; Mandala : A Logic Based Knowledge Programming System, FGCS'84, 1984, Tokyo.

大木他：Mandala Ⅱの拡張機能 ── ユーザ定義推論エンジンの実現方式 ── 、日本ソフトウェア科学会第1回大会予稿集、1984.

二村：プログラムの部分計算法、電子通信学会誌、66-2, 1983.

Prologによるルール型知識の知識同化について

國藤　進，坂井　公，宮地　泰造，北上　始，古川　康一

（財）新世代コンピュータ技術開発機構

[1] はじめに

計算機システム上で形式的推論をサポートするための研究としては従来、主としてオブジェクト・レベルの推論機構研究が行われてきた。しかしながら知識ベースの管理、知識獲得、や自律的問題解決といった高度の機能をその上に実現するには、ある形式的推論下でのメタ推論機構の研究が必要である。メタ推論機構の研究はWeyhrauch のFOL 研究[1]に、その起源を求めることができる。論理型言語Prologにおけるメタ推論の重要性を喚起したのがBowen やKowalskiの研究[2]であるが、彼らは実際に稼動している処理系上での実現法を示さなかった。そこで著者らは、DEC-10 Prolog 上でメタ推論用プリミティブdemo述語というものを導入[3),4),5)]し、それを用いた柔軟なメタ推論エンジンの実現法と利用法を示した。上記論文に刺激されて、国内では多くのメタ推論用プリミティブであるdemo述語[6),7)] やsimulate述語[8),9)] の提案が行われている。残念ながらそれらのほとんどは存在束縛型問合せに対するメタ推論方式の研究であった。Prologの内部データベースに蓄積すべき知識は、必ずしも存在束縛型知識（ルールの一部とファクト）ではなく、多くの場合全称束縛型知識（ルール）である。そこで更新される知識ベースを処理対象とするような応用においては、全称束縛型問合せに対するメタ推論方式の確立が必須となる。しかしながら、このような場合に対する考察としては、わずかに著者らの研究[10),11)] があるばかりで、それも必ずしも理論的基盤の不明確なまま利用されている。そこで本論文ではこれらの反省を踏まえ、全称束縛型問合せに対するメタ推論方式を提案し、その妥当性を保証する範囲を明確にする。

[2] demo述語

L を与えられた論理体系，P をL における論理式の有限集合、Q をL におけるある論理式とする。L においてP からQ が証明可能であるとき、

$$P \vdash_L Q \qquad (1)$$

と表わすことにする。

二つの論理体系L，H において、L からH への一対一の対応付け関数 ´ が定められているものとする。このときメタ推論用プリミティブであるdemo述語は、次のようにして導入される。

$$Pr \vdash_H demo(P',Q') \equiv P \vdash_L Q \qquad (2)$$

ここにPrはH における論理式の集合、P'、Q'はそれぞれP,Q の関数 ´ の適用結果である。

Weyhrauch [1)]は式(2) を反射原理(reflection principle)と呼んでいるが、著者らの立場では式(2) は、オブジェクト言語の言理をメタ言語上で実証(demonstrate) あるいは模倣(simulate)するための基本原理である。Bowen ら[2)]は二つの論理体系L とH が同一の形式的言語で表現できる場合に着目し、知識ベースの更新、すなわち知識の同化(knowledge assimilation)におけるdemo述語の重要性を喚起した。彼らはこれをオブジェクト言語とメタ言語の融合(amalgamation)と呼んだ。

さて以下の考察では論理体系L,H として（一階）ホーン論理のみを取扱う。ホーン論理でのdemo述語の実現法をBowen ら[2)]は述べているが、その方法をDEC-10 Prolog で直接インプリメントするのは、あまりに面倒で効率が悪い。著者らはこのdemo述語の実現法を次のように提案[3),4)] した。

```
demo(P,true):-!.                                  (3-1)
demo(P,(Q1∨Q2)):-!,demo(P,Q1);demo(P,Q2).        (3-2)
demo(P,(Q1∧Q2)):-!,demo(P,Q1),demo(P,Q2).        (3-3)
demo(P,Q):-clause(P,Q,R),demo(P,R).               (3-4)
```

ここに∧、∨はそれぞれオブジェクト言語上の論理積、論理和、述語clause (P,Q,R)は、論理式の有限集合P （Prologにおける節集合）においてヘッドQ を与えてそのボディR をフェッチするメタ述語である。

このメタ述語demoを基底とし、次のようなメタ推論用プリミティブを構築していくのは易しい[6),7),8),9)]。

```
demo(Program, Query, Control, Result)        (4)
```

式(4) は与えられた制御条件Control 下で、与えられた節集合プログラム Programからある問合せQuery が証明される過程を起動し、その証明の過程から必要なメタ情報Resultを抽出するメタ述語である。

実際、著者らのグループより、既に多くのdemo述語の試作とその応用がなされた[13)]。

メタ述語demoが論理体系としておかしな振舞いをしない（発狂しない）ためには、反射原理(2) が成立する範囲で使用しなければならない。 ProgramやQuery のクラス制限は、基本的なメタ述語demo(Program,Query) の構成法に依存する。

[3] 表記法

オブジェクト言語Program や問合せゴールQuery のクラスを限定するために、著者らは処理対象とするProgram や Query の構文を明示する。それに対応するメタ言語としては、DEC-10 Prolog の部分集合を適当にとり、出来るだけその構文に従う。またそれ以外の表記法はReiter[14] やShepherdson [15] に従うことにする。

さてまずホーン論理の（ルールとファクトからなる）プログラムの構文を、次のように定める。

$$p(X_1, \cdots, X_n) :- l_1 \wedge \cdots \wedge l_m . \qquad (5)$$

ここに $l_1, \cdots, l_m$ は肯定的あるいは否定的リテラルであり、(5) 式に出現する変数は全て全称束縛されているものとする。

(5) 式において変数を含まず、かつ m=0の場合をファクトと呼び、それ以外の(5) 式をルールと呼ぶ。

ついで問合せの構文を、次のように定める。

$$?- l_1 \wedge \cdots \wedge l_n . \qquad (6)$$

式(6) は $l_1, \cdots, l_n$ に出現する全ての変数 $X_1, \cdots, X_k$ に対して、$l_1 \wedge \cdots \wedge l_n$ を満足する解 $X_1, \cdots, X_k$ は存在するかという質問となる。すなわち問合せ実行の文脈において、変数 $X_1, \cdots, X_k$ は存在束縛されている。

オブジェクト言語としての論理型言語Prologのクラスを適切に選択することにより、demo述語をベースとする知識同化プログラムが発狂したり、発狂しなかったりする。著者らの関心は、反射原理(2) が成立し、融合された論理体系が無矛盾となるクラスを指摘することにある。そのための第一歩として、それぞれの論理体系が無矛盾となる場合について、既知の事実を分析していく。

[4] 知識同化

著者らの関心は与えられたホーン節プログラムの集合（ここでは簡単のため知識ベースと呼ぶが）への知識同化プログラムの実現[3),4),5)]にある。著者らの知識同化プログラムの基本的考えは、例えば次のような抽象的プログラムとして実現できる。

```
assimilate(Currkb,Input,Currkb):-
  demo(Currkb,Input).                          (7-1)
assimilate(Currkb,Input,Currkb):-
  demo(Currkb∪Input,false).                    (7-2)
assimilate(Currkb,Input,Newkb):-
  Info⊂Currkb,Interkb=Currkb-Info,
  demo(Interkb∪Input,Info),
  assimilate(Interkb,Input,Newkb).             (7-3)
assimilate(Currkb,Input,Currkb∪Input):-
  independent(Currkb,Input).                   (7-4)
```

ここに ∪, −, ⊂, ∈は通常集合論の記法である和集合、差集合、メンバシップの意味である。

著者らの考えとBowen ら[2)]の考えの違いは、次の２点にある。第一にBowen らと異なり矛盾検査プログラムを冗長性検査プログラムより先に評価したことである。第二に論理型言語Prologでの未定義プログラム(7-2) や(7-4) の実現方式を明らかにしたことにある。特に(7-2) の実現方式を明らかにする過程で、メタ・プログラミングの重要性を認識し、論理型言語DEC-10 Prolog を用いてのメタ・プログラミング実現手法を開拓した。その一つが論理データベースのための統合性制約(integrity constraint)の導入であり、そのために肯定的知識ベースと否定的知識ベースを分離して活用している。なおここで統合性制約とか否定的知識ベースといわれているものは、(5) 式で$p(X_1, \cdots, X_n)$がないような定義で、これは $l_1 \wedge \cdots \wedge l_m$ が常に不成立であることを意味する。

[5] 肯定的リテラルからなる確定節の場合の知識同化

前述の否定的知識の導入、およびルール型知識の同化の際に必須となる全称束縛型変数の式(6) への導入に伴い、著者らはdemo述語およびassimilate述語に種々の手直しをして使用している。そのための根拠となる理論を、最も簡単な場合から列挙していく。

まず(5) 式の l_i (i=1, ⋯,m) が全て肯定的リテラルのみで、与えられた知識ベースKBが全て確定節(definite clause) の場合を考える。この場合、否定的知識ベースが存在しないので、著者らの関心のあるのは、(i) Pure Prolog の場合、(ii) ClarkのCompleted Data Base (CDB)が成立する場合、(iii) ReiterのClosed World Assumption (CWA) が成立する場合である。明らかに次の二定理が成立する。

［定理１］ Pure Prolog の場合、任意のKBは常に無矛盾(consistent)である。

［定理２］ もしKBが肯定的リテラルからなる確定節だけから成れば、CWA ∪ CDB は無矛盾である。

従ってどちらの場合も、知識同化プログラムから式(7-2) の検査が不要となる。

[6] Pure Prolog with Negation の場合のルール同化

著者らの関心は、知識ベースにファクトのみならずルールを知識同化することにある。そのためには式(7) のInput として式(5) のタイプを許容する場合を詳細に分析しなければならない。幸い、坂井らはCWA もCDB も仮定しない古典論理の場合のルール型知識同化アルゴリズム[16)] を提案している。本アルゴリズムは著者らのグループでは以前から知られていたが、その理論的根拠は上記論文[16)] でPure Prolog with Negation(PPN)として与えられた。

PPN は肯定的知識ベースと否定的知識ベースとからなる。ここに肯定的知識ベースは(5) 式の右辺に肯定的リテラルのみをもつ確定節の集合として定義される。また否定的知識ベースは(5) 式の右辺が肯定的リテラルのみからなり、

(5) 式の左辺が空な節の集合として定義される。PPN は全称束縛型問合せ、すなわち確定節が証明可能かの実証も許す。著者らの立場ではPPN の場合、ルールを許容するdemo述語を構築することができる。

〈Procedure 1 〉

(a) Input が存在束縛型問合せならば、Pure Prolog の場合と同様に扱う。

(b) Input が全称束縛型問合せ、すなわち $\forall X_1 \cdots \forall X_n$ (not(l_1) $\vee \cdots \vee$ not(l_s) $\vee$ p($X_1, \cdots, X_n$)), ならば、各変数にこのKB中に含まれない定数（いわゆるスコーレム関数）を割当てる。この割当てをθとする。

(c) 肯定的知識ベースに $l_1\theta, \cdots, l_s\theta$. というファクトをアサートする。否定的知識ベースに :-Gp. という節をアサートする。

(d) PPN の場合の矛盾検査アルゴリズム[15] を適用し、もしそれが矛盾を検出すれば、上述の全称束縛型問合せは与えられた知識ベースから証明可能である。

なおPPN の場合の矛盾検査アルゴリズムとは、否定的知識ベースから任意の一つを取ってきて、それを肯定的知識Query として（存在束縛して）用いた時に証明されれば、その知識ベースは矛盾するというアルゴリズムである。

〈Procedure 1 〉はPPN の場合のdemo述語に相当する。従って式(7) で与えたdemo述語を、上述の手続きで置換すれば、PPN の場合の知識同化プログラムができる。

[7] Query Evaluation Procedureの場合のルール同化

ReiterのCWA やClark のCDB のもつ性質については、証明論的にはかなりいろいろなことが分ってきた。しかしながら、否定的知識の取扱いの問題が派生すると、分らないことの方が多い。すなわち肯定的知識ベースが共存するばあい、CWA やCDB 向きのdemo述語の構築法はまだよく分っていない。

しかしながら最近Shepherdson [15] が、上述のCWA とCDB 、およびClark のQEP(Query Evaluation Procedure) のもつ性質を詳細に分析し、次の定理を証明した。

［定理３］ もし全ての基礎リテラルに対して、QEP のもとで失敗あるいは成功するような選択ルール(selection rule)が存在すれば、CDB は無矛盾である。しかもかつ、CWA が無矛盾なら、CDB　CWA も無矛盾である。

［定理３］は、QEP がCDB やCWA のサブシステムであることを示している。［定理３］の選択ルールの存在の仮定が最も本質的であり、これを用いるとQEP 向きのdemo述語の構築が容易である。

```
demo(P,true):-!.                                    (8-1)
demo(P,not(Q)):-
 !,ground_literals(Q),\+ demo(P,Q).                 (8-2)
demo(P,(Q1∨Q2)):-
 !,deferred_not(Q1∨Q2,Q3∨Q4),
 (demo(P,Q3);demo(P,Q4)).                           (8-3)
demo(P,(Q1∧Q2)):-
 !,deferred_not(Q1∧Q2,Q3∧Q4),
 demo(P,Q3),demo(P,Q4).                             (8-4)
demo(P,Q):-clause(P,Q,R),demo(P,R).                 (8-5)
```

ここにground_literals (Q)はリテラルQ が基礎例かどうか、すなわち変数を含まないことを判定する述語、deferred_not(Q1#Q2, Q3#Q4) (#は∨あるいは∧) は、Q1がnot(_) 型ならばその評価を後送りにし、Q1とQ2を入替える。そうでなければそのまま評価する述語である。

上述のdemo述語(8) 式が与えられている時、ルール型知識の証明可能性を判定するuniversal_demo述語の意味を次のようにして付与する[10),11)] 。

```
universal_demo(P,(E:-B)):-
 demo(P,B),(demo(P,not(E)),!,fail;fail).            (9-1)
universal_demo(P,_):-!,true.                        (9-2)
```

ここに２引数demo述語は、(8) 式で与えられる。

(9) 式で与えられるルール型知識の証明可能性の意味は、Prologルールのボディを真ならしめる有限の具体値(instantiated value)がヘッドをも真ならしめることを、ことごとく確認することにある。これによって言えることは、高々与えられた知識ベースからそのルールを適用することによって得られる基礎例の全てにおいて、そのルールが成立することである。従って本当の意味でのルールという概念を証明したことにはならないので、知識同化プログラムに組込むには若干の手直しが必要である。ここではその結果のみ、簡単に述べる。

〈Procedure 2 〉

(a) Input が存在束縛型問合せならば、(8) 式のdemo述語を(7) 式の知識同化プログラムに組込む。

(b) Input が全称束縛型問合せならば、次のアルゴリズムを適用する。

(a-1) Input が否定的知識ベースに矛盾すれば、矛盾する新知識Input を採り込まない。

(a-2) Input が否定的知識ベースに矛盾しなければ、(9) 式を呼び出し、そのルールが基礎例レベルで証明できるかどうか検証する。

(a-2-1) もしその結果が真ならば、（Naive Induction の原理を導入し、）そのルール型知識Input を肯定的知識ベースに採り込む。しかる後、肯定的知識ベースのファクト型知識の冗長性を除去する。次にルール型知識の冗長性除去に関するユーザとの対話過程に進行する。

(a-2-2) そうでなければ、そのルール型知識Input を肯定的知識ベースに採り込む。

[8] おわりに

本論文では否定的知識やルール型知識の取扱いに関連を

則し、それらが無矛盾に知識同化できる論理のクラスを明らかにしてきた。特にPure Prolog の自然な拡張であるPPN の場合やCWA やCDB のサブシステムであるQEP の場合のルール同化プログラムを明らかにした。しかしながら論理型言語Prologの既存処理系の動きをより自然に説明するのは、CWA やCDB が成立する場合である。従ってCWA やCDB 向きのdemo述語やassimilate述語を構成するのが、今後最も重要な研究課題である。

ここにCWA のような非単調な論理体系を扱うのは、極めて困難な問題が横たわっているのを理解しておきたい。例えばひとつの知識を肯定的知識ベースに追加すると否定的知識ベースとの間に矛盾を生じるにもかかわらず、二つ以上の知識を前者に追加すると後者との間に何ら矛盾を生じなくなる場合がある。また実際の応用場面においては、或世界のある種の関係においてはCWA が成立するが、別種の関係においてはOWA (Open World Assumption) しか成立しない場合がある。このような場合、人間はそれぞれの関係ごとにCWA とOWA を使い分けるが、CWA とOWA が混在する論理体系での知識獲得問題が全く解けていない。

以上述べてきたように、Pure Prolog の世界を一歩踏み出ると否定的知識やルール型知識の知識同化問題は、極めて慎重な取扱いを必要とし、しかも論理体系として扱かねばならない多くの未解決問題をかかえている。そのような場面に有効なメタ推論方式を確立するためにも、CWA やCDB が成立する場合の知識同化プログラムを研究しなければならない。

［謝辞］ 本研究の機会を与えられた渕一博ICOT研究所長、日頃ご討論いただくICOT研究所第２研究室の方々、および東京理科大学溝口文雄助教授に感謝します。

（参考論文）

1) Weyhrauch,R.W.,Prolegomena to a Theory of Hechanized Formal Reasoning,Artificial Intelligence 13(1980),133-170.

2) Bowen,K.A.,Kowalski,R.A.,Amalgamating Language and Meta-language in Logic Programming,TR 4/81, Syracuse University,June 1981.

3) 國藤 進，麻生盛敬，竹内彰一，坂井 公，宮地泰造，北上 始，横田治夫，安川秀樹，古川康一，Prologによる対象知識とメタ知識の融合とその応用，情報処理学会知識工学と人工知能研究会資料30-1，1983年 6月(TR-009).

4) Miyachi,T., Kunifuji,S., Kitakami,H., Furukawa,K., Takeuchi,A., and Yokota,H., A Knowledge Assimilation Hethod for Logic Databases, Proc.of the 1984 International Symposium on Logic Programming,Atlantic City ,U.S.A.,Feb. 6-9, 1984.

5) 北上 始，麻生盛敬，國藤 進，宮地泰造，古川康一，知識同化機構の一実現法，情報処理学会知識工学と人工知能研究会資料 30-2,1983年 6月 (TR-010).

6) 國藤 進，竹内彰一，古川康一，上田和紀 他，核言語第１版概念仕様書（案），新世代コンピュータ技術開発機構，1983年 8月.

7) 國藤 進，宮地泰造，北上 始，古川康一，知識獲得とメタ推論，情報処理学会第27回全国大会，名古屋大学，1983年10月(TH-0016).

8) 國藤 進，竹内彰一，古川康一，上田和紀，メタ推論とその応用－並列メタ推論与メタ述語disjointについて－，情報処理学会第28回全国大会，電気通信大学，1984年 3月(TH-0036).

9) 古川康一，國藤 進，竹内彰一，上田和紀，核言語第１版概念仕様書，新世代コンピュータ技術開発機構，ICOT TR-054 , March 1984.

10) Kitakami,H., Kunifuji,S., Miyachi,T., and Furukawa,K., A Methodology for Implementation of A Knowledge Acquisition System, Proc. of the 1984 International Symposium on Logic Programming,Atlantic City, U.S.A., Feb. 6-9, 1984(TR-037).

11) Miyachi,T., Kunifuji,S., Kitakami,H., Furukawa,K., Takeuchi,A., and Yokota,H., A Knowledge Assimilation Hethod for Logic Databases, New Generation Computing ,Vol.2,No.4,1984.

12) Pereira,F.,Warren,D.H.D., Definite Clause Grammars Compared with Augmented Transition Networks,DAI,Univ. of Edinburgh,1978.

13) 北上 始，國藤 進，宮地泰造，古川康一，知識の同化・調面・均衡化などのユーザ・インタフェースを備えた知識獲得システム，日経エレクトロニクス，1984年 11月 5日号，no.355,pp.261-286.

14) Reiter,R.,On Closed World Databases,in Logic and Data Bases(Gallaire,H.,Minker,J.,eds.),Plenum Press,New York/London,1978,pp.55-76.

15) Shepherdson,J.C.,Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumptions,J. Logic Programming,No.1,1984,pp.51-79.

16) Sakai,K.,Miyachi,T.,Incorporating Naive Negation into Prolog,ICOT TR-028,Nov. 1983.

Concurrent Prologのシーケンシャル・インプリメンテーション
（Copy方式による多環境の実現）

田中二郎、　　宮崎敏彦、　　竹内彰一
（富士通国際研）　（新世代コンピュータ技術開発機構）

［概要］
Concurrent Prolog の処理系（インタプリタ）をMaclispで作成した。本発表ではLazy copy 方式に基く多環境の実現について述べる。

1．はじめに

Concurrent Prolog [1]（以下CPと略称）はShapiroにより提案された言語であり、Prologにガードと読出専用標記を付加することにより並列処理機能を持たせている。

一般に一つのGoalにunify 可能なHeadを持つclauseの集合をOR-clausesという。CPの処理系試作にあたっては、OR-clausesのガード部に対応する環境（ここでは変数にその値を結びつける束縛情報を環境と呼んでいる。詳しくは[2]を参照のこと）をどう作るかが問題になる。

Shapiro がPrologの上に実現したCPの処理系[1] では、OR-clausesは原則として逐次的に実行され、環境はbacktrackingにより修復された。したがってこの処理系では多環境(multiple environment)を実現する必要がなかった。

しかしながらOR-clausesを並列に実行する処理系の作成にあたっては、多環境をなんらかの形で作ってやる事が必要となる。多環境は、あるGoalにunify 可能なOR-clausesのそれぞれに対し作られる。OR-clausesは並列に実行され、その中で一番早くcommitに達したclauseがcommitされる。そのclauseの環境は親のGoalの環境とunify され、同時に並行して走っている他のOR-clausesはabort される。

多環境の実現方式については以下に示すような幾つかの方式が考えられる。

(1)Shallow binding方式

(2)Deep binding 方式

(3)Copy 方式
　(Lazy copy方式，Eager copy方式)

これらの方式についてそれぞれ簡単に解説すると以下のようになる。

(1) のShallow binding 方式とは多様な環境をcontext switching により切換えながら実現しようというものである。環境を切換えながら多環境を実現すると言うのは真の意味で多環境を実現する事にはならないが、sequential implementation にはそれでさしつかえない。このShallow binding 方式においてある変数の値を知りたい時にはその環境にcontext をswitchしてから値を見ればよい。

(2) のDeep binding方式とは、親の環境とそれぞれのガード内部の環境の差にあたる部分だけをassociation listの形で保存するものである。ある環境において変数の値を見るときには、まずassociation listを調べ、無いときには親の環境をみる。（この方法の変型としてまず親の環境を調べその値が未定義であるときに限りassociation listを調べるという方法がある。）　一般にguard がnestしているときには、環境をどんどんさかのぼってみていく事になるのでこの方式をDeep binding方式という。

(3) のCopy方式とはOR-clausesが呼ばれるとき、それぞれに対応するlocal 環境をHead側のtermのcopyを作ることにより実際に作る方式である。その際、最初から親の環境をlocal 環境にcopyしてしまうというのがEager copy方式であり、最初はOR-clause から親のGoalの環境にpointer だけつないでおき、必要に応じて少しずつ親の環境をcopyするというのがLazy copy 方式である。

我々はこのShallow 方式、Deep方式、Copy方式のそれぞれの方式に基き、Maclisp で処理系（インタプリタ）を作成した[2][3]。本論文ではCopy方式のうち特にLazy copy方式に基く処理系作成について述べる。

2．Copy方式による多環境の実現

Copy方式には前述のようにEager copy方式とLazy copy方式の２つがあるが、この２方式の違いは以下の通りである。

(1) Eager copy方式

HeadとGoalのunify 時にGoalに含まれる変数に対し、それが読出専用でなければその環境をcopyする。Unification により計算が進み変数がinstantiate されるが、それらはすべてcopyした環境を書きかえる。Commit時にはGoalとcopyされ書きかえられた環境とを再びunify し、親の環境に値が届かれる。

(2) Lazy copy 方式

この方式では、Goal側の引数は最初からcopyされる事はなく、特殊なポインタ（get ポインタ）で指されている。get ポインタはその先が親のGoalの環境であることを意味

する。Unification により計算が進み getポインタの先を書き換える時には、必要な部分だけ getポインタの先をcopyしながらたぐり、copyした変数の方を書きかえて親の環境を汚さないようにする。Commit時にはcopyした環境と親の環境を直にunify し、親の環境への書込みが行われる。このLazy copy 方式はJacob levyにより最初に提案されたものである。[4]

3．Jacob Levy方式の問題点

一般にConcurrent Prolog ではunification を通じて異なる引数間にcommunication が生じる事がある。

```
(Goal)    f（X，X）    ／＊　Xは未定義　＊／
―――――――――――――――――――――――
(Head)    f（b，C）
```

上例において定数bと変数CとはGoal側のXを通じてcommunication が生じCはbに等しくなる。こういった問題を扱うには単純なcopy戦略ではだめで、Levyはこの問題を解決するため、GoalからHeadにポインタ（ref ポインタ）を一時的に張り、後でポインタを切る（undoする）事を考えた。

しかしながらその後、この解決法では問題を正しく扱えない事が明らかになっており[5][6]、我々はこの問題点を解決するため、Head側の環境にcopyした変数のリスト(association list)を持つ事にした。このassociation listはcopyされた親の環境のcellのaddress と新しく作ったcopyのaddress のpairを並べたものである。copyを作るときには、まずこのassociation listを調べ、すでにcopyされたセルがある時には新しくcopyを作らず、すでに作られたセルを共有する。また、まだcopyが作られていない時にはcellを新しく作る事になるが、その際には必ずassociation listに登録するようにする。こうしてLevyの提案した方式の問題点を解決した。

4．我々のCPの処理系(interpreter) の特色

我々のCPの処理系(interpreter) の特色は以下のようにまとめられる。

①我々の処理系はOR-clausesを並列に（といってもあくまでsequential implementation であるので疑似的にOR-clausesをscheduling queueにいれて実行するに過ぎないが）実行する。そのため多環境とcopyを作る過程とを正しくimplement した。

②処理速度向上のため、ガードの内部がbuilt-in predicateだけからなる場合(immediate guard) には直ちに解くようにした。

③処理速度向上のためsuspend されたunification について、従来のShapiro [1] のようなbusy wait 方式でなく、suspend する原因となった変数にsuspend されたprocessをつないでおき、変数がinstantiate され次第scheduling queueに入れられるような方式をとった。

④簡単化のためpredicate やfunctor の引数、リストの要素のunification は左から順に行われると仮定した。

5．Lazy copy 方式の概要

我々はCPの処理系をlazy copy 方式に基いてimplementしたが、本章ではlazy copy 方式によるdereferencing とunification のアルゴリズムについて解説する。

5．1．Dereferencing

一般にPrologのimplementationにおいては変数同志のunification の際に片方の変数セルから他方の変数セルにポインタが張られ変数セルのチェーンが生成される。[7] Dereferencing （以下deref と略称する）とはそのポインタをたどり変数の値を求めることである。

Prologのimplementationにおいてはポインタは一種類(refポインタ）のみであったが、我々はその他に新しく次の２つのポインタを導入した。

①roref ポインタ。読出専用標記に対応する。

②get ポインタ。ポインタの先が親のGoalの環境である事を意味する。

またPrologの場合deref された結果は①undef （変数）②non variable term （atom,vect,list）のどちらかであったが、我々のimplementationでは以下のようになった。

①undef （未定義の変数）

②ro variable （読出専用変数）

③non variable term (atom,vect,list)

④get （先が親の環境）

②のro variable はCPが読出専用変数を含む事から生じたもので、deref の過程でroref ポインタをたどり未定義の変数にいきあたった場合に対応する。また④のget は、とりあえずderef はget でやめ、あとはunification の過程に応じてcopyを作っていくことに対応する。

5．2．Unification

こうしてあらかじめderef された二つの項をunify するときのアルゴリズムを示したのが次ページの図である。図において、本質的にＴ１とＴ２に関してunification が対称になっていることに留意されたい。この図の意味は以下のようにまとめられる。

①変数同志のunify の場合には片方から他方にポインタを張る。

②片方が変数である時には常にその変数から他方にポインタを張る。

③片方がget ポインタであるときには相手が変数であれば

変数からget ポインタへref ポインタを張る。相手が変数以外であればget の一段中身のcopyと相手をunify する。
④双方がnon variable term の時はそれぞれのelement についてunify すればよい。
なお実際のunification ではGoal側の引数にすべて"get"を被せてからunification を行う。（すなわちGoal側の変数は汚さない。Commit前には、get を被せる前のGoalの変数とHead側の引数を再びunify しなおす必要がある。）

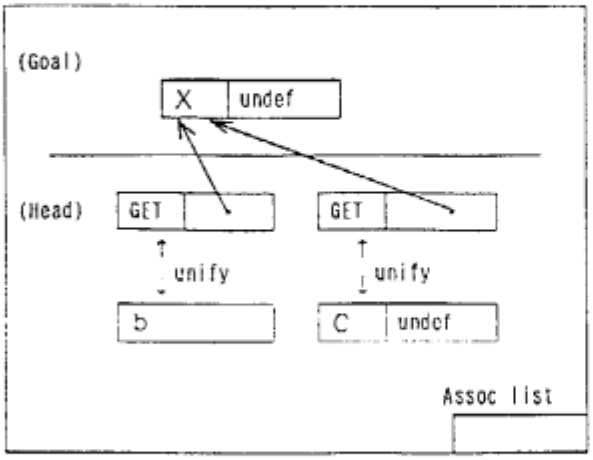
T1 \ T2	VAR	ROV	TERM	GET
VAR	T1 := Ref (T2)			
ROV	T2 := Ref(T1)	SUSP	SUSP	Unify (T1, Copy(T2))
TERM		SUSP	U-TERM	
GET		Unify(Copy(T1),T2)		

5．3．Unification の例題

前節までで簡単にderef 及びunification について説明したが、以下これを簡単な例題について考えてみる。例として次のようなケースを考える。（この例は３章に示した例と同じである。）

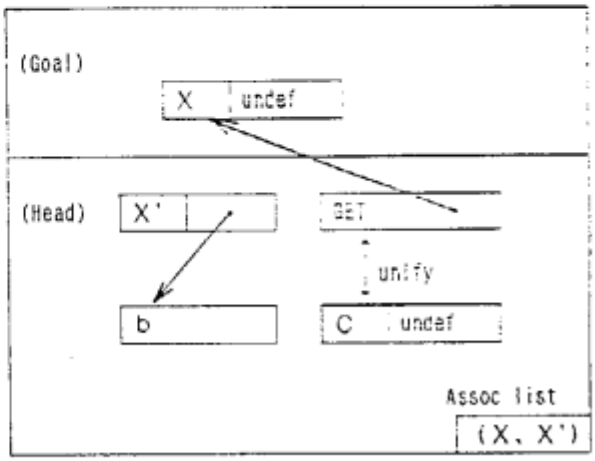
```
(Goal)   f（X，X）   ／＊  Xは未定義  ＊／
(Head)   f（b，C）
```

このunification の様子を示したのが下図である。（以後、図の表記法については[6][7]を参考にした。）

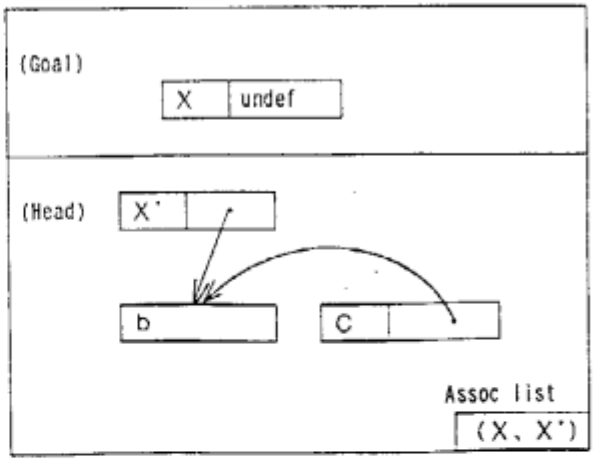


我々の処理系では、引数のunification は逐次的に行われるので、XとＢ、XとCの順でunification が行なわれる。まずGoal側の引数にすべてget が被せられ、get(X)とＢ、get(X) とCの間でunification が行われる。

次にunification の過程を説明する。まずget(X) とＢのunify においては、get の中身のcopyをつくる必要がある。assoc listがまず捜されるが、まだassoc listは空なのでget の中のcopy X' が作られassoc listにXとX'のpairを登録したあとＢとX' がunify される。



さて次にCとget(X) のunify であるが、ここでもgetの中のcopyを作る必要がある。しかしながらassoc listにXのcopyはすでに作られているので、X' の値とCがunify されることになる。X' のderef した値はＢなのでCからＢにポインタを張る。その結果を下図に示す。



以上でf(X, X) とf(b, C) のunification が終了したことになる。この段階でGoal側の変数は未だ汚されていない事に留意されたい。Commit時にget を被せていないGoal側の引数とHead側の引数とが再びunify し直され、Goal側の引数に初めて書込みがおこる。

5．4．Lazy copy 方式の長短

前節までで簡単にLazy copy 方式について解説した。Lazy copy 方式の長短についてまとめると以下のようになる。

［長所］

(1) 親の環境とガードとはget ポインタを通してつながっ

ている。必要に応じてcopyがつくられるので作るセルの数が少なくてすむ。
(2) get ポインタは必ず一段上の親の環境をさす。get ポインタの数を数えればガードが何段ネストしたかがわかることになり他の shallow方式、deep方式にみられるようなガードシステムナンバーなどの管理が不要である。
［短所］
(1) get ポインタをもちこむのでderef, unification等のアルゴリズムが複雑になる。
(2) 一度copyしたかどうかをassoc listに記憶させるのでcopyをつくるたびにassoc listを調べ、すでにcopyを作ったかどうか調べなくてはいけない。
(3) lazy copy 方式ではGoal側の変数が少しずつget ポインタを通してcopyされる。そのためskelton とenvironment からなるmoleculeを使うstructure sharing の方式になじみにくい。

6. Implementaion とその評価

上述のlazy copy 方式に基きCPの処理系(interpreter)の試作をMaclisp で行った。主に性能検証のためのシステムであるので、parserについてはまずPrologで前処理し、list構造に変換したあとLispに渡す。またbuilt-in predicateについては最小限のみを作った。

処理系は主にscheduling部分とunification 部分からなるが、scheduling部分に対しては、bench markをとる都合等から他のdeep, shallow 方式と共用する事とし、主としてunification 部分を試作した。

Lazy copy version についてはunification 部分の処理系の programはsizeにして約 900行であった。他の方式(deen やshallow)によるprogram のsizeは約 400行である。(なおDeep, Shallow, Copy 各方式に共通するscheduling部分のcodingは約 600行である。) 速度については、これら3方式はそれぞれ異なった特性を持っており単純に比較はできないが、幾つかのbench markプログラムを走らせたところ、speed は200 lips程度であった。他のdeepやshallow 方式が450 lips程度である事を考えると、lazy copy 方式がかなりのcomputation complexityを持つ事を示していると思われる。

7. まとめ

以上lazy copy 方式に基くCPのinterpreter の概略について解説を行った。我々の処理系はOR並列とsuspend されたprocess の変数へのhookを実現したため処理系としては大きなものになった。

Lazy copy 方式については計算するプログラムのガード部の複雑さなどにもよるが、残念ながら逐次implementationの場合、deref やunification のoverheadが大きく、bestの方法とは言えなそうである。Lazy copy 方式は、並列環境でのimplementationを考える際に参考になると考えられ、さらにICOTのPIM(Parallel Inference Machine) とのかかわり等について考察する必要がある。

［謝辞］

本論文をまとめる機会を提供して下さった富士通国際研の北川会長及びICOTの吉川第二研究室長、codingの際にadviceを戴いた三菱総研の市吉伸行氏、本稿についてcomment を戴いた日本電気の上田和紀氏に感謝する。またICOTのKL1 検討会に参加されKL1 のsequential implementation について意見を述べられた関係諸氏に感謝する。

［参考文献］

[1] Shapiro, E.Y., A Subset of Concurret Prolog and Its Interpreter, ICOT Technical Report, TR-003, Feb. 1983, 73p.
[2] 宮崎敏彦, 他, Concurrent Prolog のシーケンシャル・インプリメンテーション, (Shallow bindingによる多環境の実現), 日本ソフトウェア科学会, 第一回大会, 3D-2, Dec. 1984.
[3] 佐藤裕幸, 他, Concurrent Prolog のシーケンシャル・インプリメンテーション, (Deep binding による多環境の実現), 日本ソフトウェア科学会, 第一回大会, 3D-3, Dec. 1984.
[4] Levy, Jacob, A Unification Algorithm for Concurrent Prolog, Proc. of Second International Logic Programming Conference, Uppsala Univ., Uppsala, Sweden, July 1984, pp.333-341.
[5] Shapiro, E.Y.,Sequential Implementation of Concurrent Prolog, private communication, 1984.
[6] 上田和紀, Demand Copying方式のCP処理系について, kl1 検討会資料, 1984-06-29.
[7] Warren David H.D., Implementing Prolog - compiling predicate logic programs, D.A.I. Reserch Report, No.39,40, 1977.

並列論理型言語の実用処理系

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．あらまし

並列論理型言語Concurrent Prolog のプログラムを，逐次Prologのプログラムに変換する処理系を，DEC-2060上に作成した．変換されたプログラムは，さらにPrologコンパイラで機械語に変換されて実行される．DEC-10 Prolog インタプリタを十分上回る実行速度を持っており，並列論理プログラミングの実践に大いに役立つと期待できる．

2．経緯

Ehud Shapiroが[Shapiro 83]でConcurrent Prolog （ＣＰ）を提案して以来，同論文に示されたPrologによるインタプリタを基本としたＣＰ処理系が，ICOTを中心に利用されてきた．この処理系は，手軽な並列プログラミングの実験道具として重宝ではあったが，実行速度に大きな問題があった．

並列プログラミング言語としてのＣＰの実用性を立証するためには，インタプリタではなくコンパイラを試作することによって

① プログラムの中からどれだけの静的情報が引き出せ，利用できるかを調べ，
② 同時に実用規模のプログラムを動かす環境を提供する

ことが重要である．この認識に立って，ＣＰプログラムのコンパイラを作成することにした．

コンパイラのターゲット言語と記述言語には，DEC-10 Prolog [Warren 77] を選んだ．これは，

① DEC-10 Prolog はすぐれたコンパイラを持っている
② ソース言語とターゲット言語のギャップが小さいため，コンパイラ作成が容易である
③ コンパイラの移植性が高い
④ ＣＰとPrologのインターフェース機能を用意すれば，システム述語作成の手間が大幅に減る

からである．高級言語へのコンパイラの例としては，最近では Lisp を Pascal に翻訳する例［梅村84］などがあるが，今回の処理系はソース言語，ターゲット言語，記述言語の３者間のギャップの小さいことが特徴である．

3．言語仕様

作成した処理系は，基本的には[Shapiro 83]のインタプリタをコンパイラ化したものと言える．言語仕様上の拡張点は，次のとおりである．

① 逐次AND を導入した．
② メタ呼出し
　call(Goals, Result, Interrupt)
　を提供した．仕様はPARLOG [Clark and Gregory 84] に沿ったものであり，第２引数には，Goalの実行が成功したときに値'success' が返る．第３引数の値を'stop'に具体化すると，Goals の実行が強制終了させられる．このとき第２引数には値'stopped' が返る．
③ モード宣言機能を提供した．目的は，コードの大きさを減らして実行速度を上げることと，節のindexing（ランダム・アクセス）を行なうコードを生成することにある．節のindexing機能は，第１引数のモードを“＋”にしたときに，その主ファンクタによって行なわれる．これは，DEC-10 Prolog コンパイラのindexing機能をそのまま利用したものである．

一方，次の諸点も[Shapiro 83]のインタプリタから引きついでいる．

① 複数のガードの（擬似）並列実行は行なわない．すなわち，ひとつのガードの実行が中断するか失敗するまで，他の節のガードの実行は行なわない．
② Read-only annotation によって実行が中断したゴールは，動的待合せ(busy wait) によって再開を待つ．
③ ゴールの失敗と中断の区別がない．あるゴールをreduceできる節がひとつもなかったときは，その原因が中断であっても，失敗であっても，そのゴールを再びスケジュールすることがある．

しかし，これらの諸点は，現存する実用的なＣＰプログラムを逐次計算機上で実行する上での不都合にはほとんどならない．まず

① ガードの擬似並列実行を要するプログラムは今までほとんど書かれていないし，そのようなプログラムも他の方法で記述できる．また
② 動的待合せ方式であっても，ゴールのスケジューリングの工夫（後述）によって，中断の回数をきわめて少

なくすることができる．さらに

③ ＣＰの特徴であるストリーム通信を行なうプログラムでは，ガード内で呼ぶ小さなゴールを別にすれば，各ゴールが成功裡に終了するようにプログラムを書くのが普通である．

ゴールのスケジューリングに関しては，bounded depth-first を標準仕様とした．n-bounded depth-first とは，各ゴールが連続ｎ回reduction できることである．ゴールＧが連続ｎ回reduction できるとは，Ｇがreduceされて本体部のゴール $B_1 \sim B_m$ に展開されたとき，$B_1 \sim B_m$ 以外のゴールの実行に先立って各 B_i (i=1,...,m) が連続ｎ－１回reduction できることである．ｎの値は実行時に指定できる．ｎ＝１ならばbounded depth-first はbreadth-first と同じであり，ｎ→∞とするとdepth-first になる．ただし効率の必要なプログラムのために，depth-first 専用のコードも生成できるようにした．

コンパイラではあるが，実行トレースをとることもできる．ただし，実行トレースをとるためには，ソースプログラムをトレースモードでコンパイルしておくことが必要である．

４．コンパイル技法

コンパイルによって高速化できるのは，主としてスケジューリングとユニフィケーションである．このふたつについて解説する．

４－１．スケジューリング

ＣＰの各述語は節ごとにPrologの節に翻訳される．そして述語ごとに，中断および失敗の処理をする節が付加され，さらに節のindexingやトレースを行なうときはそのための節が付加される．図１(a)，(b)にコンパイル例を示す．

コンパイルされた節の引数は，もとのＣＰの節の引数の対応するもののほかに，５個追加される．ひとつめはn-bounded depth-first ｎを表わすカウンタ，ふたつめとみっつめは継続（未終了のゴールの待ち行列）を表わす差分リスト，よっつめはデッドロック検出用のフラグ，いつつめがbounded depth-first のbound の初期値である．ＣＰの

```
Head :- Guard | Body.
```

という節は，

```
（引数の受取り） :-
    (Head のユニフィケーションとGuard の実行)，!，
    (Body の実行)．
```

の形に変換される．“(Head のユニフィケーションとGuard の実行)”のところでは，処理中のゴールが１回以上reduction 可能かどうかも調べる．

“(Body の実行)”のところに生成されるコードは，次のことを行なう．

① Bodyが空（true）であれば，継続の先頭のゴールを呼び出す．

```
qsort([X|Xs], Ys0, Ys2) :-
  partition(Xs?, X, S, L),
  qsort(S?, Ys0, [X|Ys1]), qsort(L?, Ys1, Ys2).
qsort([], Ys, Ys).
:- mode partition(?, ?, -, -).
partition([X|Xs], A, S, [X|L]) :-
  A <  X | partition(Xs?, A, S, L).
partition([X|Xs], A, [X|S], L) :-
  A >= X | partition(Xs?, A, S, L).
partition([], _, [], []).
```

(a) CP source program

```
:-fastcode.
:-public qsort/8.
:-mode qsort(?,?,?,+,?,-,+,+).
qsort(A,B,C,D,E,F,G,H) :-
  ulist(A,I,J), D>0, !, K is D-1,
  partition(J?,I,L,M,K,
    [$(qsort(L?,B,[I|N],K,O,P,Q,H),O,P,Q),
     $(qsort(M?,N,C,K,R,S,T,H),R,S,T)|E],
    F,nd,H).
qsort(A,B,C,D,E,F,G,H) :-
  unil(A), unify(B,C), D>0, !,
  E=[$(I,J,F,nd)|J], incore(I).
qsort(A,B,C,D,[$(E,F,G,H)|F],
  [$(qsort(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).
:-public partition/9.
:-mode partition(?,?,?,?,+,?,-,+,+).
partition(A,B,C,[D|E],F,G,H,I,J) :-
  ulist(A,D,K), F>0,
  cpwait(B,L), cpwait(D,M), L<M, !,
  N is F-1, partition(K?,B,C,E,N,G,H,nd,J).
partition(A,B,[C|D],E,F,G,H,I,J) :-
  ulist(A,C,K), F>0,
  cpwait(B,L), cpwait(C,M), L>=M, !,
  N is F-1, partition(K?,B,D,E,N,G,H,nd,J).
partition(A,B,[],[],C,D,E,F,G) :-
  unil(A), C>0, !,
  D=[$(H,I,E,nd)|I], incore(H).
partition(A,B,C,D,E,[$(F,G,H,I)|G],
  [$(partition(A,B,C,D,J,K,L,M,J),K,L,M)|H],
  I,J) :- incore(F).
```

(b) Object program in Prolog-10

```
:- public '$END'/3.
'$END'([], _, _) :- !.
'$END'([$(G,Ch,Ct,d)|Ch],
  [$('$END'(Ch2,Ct2,Dnd2),Ch2,Ct2,Dnd2)|Ct],
  nd) :- incore(G).
```

(c) System predicate for the detection of deadlock and termination

Fig. 1 Compiling CP into Prolog

② Bodyのゴールがひとつのときは，そのゴールに，受けとった継続と同じものを与えて呼び出す．

③ Bodyのゴールがふたつ以上のときは，受けとった継続の先頭に，ふたつめ以降のゴールを追加し，それを1番目のゴールの引数に渡して呼び出す．

これからわかるように，CPのtail-recursiveなプログラムは，Prologに変換されてもtail-recursiveであり，このことが高速化の鍵となっている．

中断および失敗の処理をする節は，各述語の最後の節として生成される．この節は，処理中のゴールを継続の最後に登録し，継続の先頭のゴールを呼び出す．

端末からゴールを呼び出すときには，継続として，実行終了およびデッドロックを検知するためのシステム述語 '$END'（図1(c)）の呼出しを与える．

逐次AND やメタ呼出しを含む節をコンパイルすると，逐次処理や割込検知用のゴールが生成し，継続の処理が複雑化するが，基本的な技法は上と同様である．

4－2．ユニフィケーション

この処理系では，read-only annotationをPrologのファンクタで表現しているので，CPのユニファイアは，一般には述語として定義しなければならない．しかし，ユニファイすべき項の一方はソースプログラムの頭部に書かれているのだから，その形に応じた専用のユニファイアを呼出すようにすれば，一般のユニファイアを常に用いるよりも効率がよくなる．この工夫は，DEC-10 Prolog コンパイラにならったものである．

この処理系ではさらに，節のindexingを行なう引数と，出力モード（ゴール側が変数であることを保証するモード）の引数については，Prologのユニフィケーションをそのまま利用するようにして効率化を図っている．

CPのindexing機能は，DEC-10 Prolog のそれを利用したものなので，DEC-10 Prolog によるindexingの前にあらかじめread-only annotationの処理を行なっておかねばならない．そこで，indexingを変する述語については，二段階の述語呼出しによって節を選択するようなコードが生成される．

図2は，モード宣言の有無による目的プログラムの違いを，appendプログラムを例にとって示したものである．ただし，appendのように節の数が少ないものに関しては，第1引数でindexingを行なわない方が効率がよい．

5．性能

いくつかのCPプログラムをコンパイルし，実行した結果を表1に示す．表1には，従来のCPインタプリタによる実行結果，および各CPプログラムと同じ入出力関係をもつPrologプログラムをDEC-10 Prolog 処理系で実行した結果も，参考のために掲げた．各プログラムは繰り返し実行し，1回目の実行時間を除外して残りの平均を求めた．1回目のデータを除外するのは，DEC-2060のメモリ管理機構の関係で，2回目以降よりかなり大きな値を示すことがしばしばあるからである．

表1からわかるように，モード宣言を与えてdepth-first 用にコンパイルしたものは，スケジューリングに違いがあるとはいえ，CPインタプリタの12～ 220倍の実行速度を得ている．DEC-10 Prolog インタプリタと比べても 2.7 ～ 4.4倍の実行速度である．DEC-10 Prolog コンパイラから得られる最高速のコードと比べた速度低下も，1/2.7 ～ 1/5.3 にとどまっている．実行速度の最大値としては，appendプログラムで，11.5kRPS (Reductions Per Second, ガード部のないプログラムについては，通常のLIPS値と同じ）を越す値が得られた．

モード宣言による速度向上は19% ～84% であった．スケジューリングをdepth-first にすることによる速度向上は，最大27% であった．

長さ1のBounded bufferを用いて通信するプログラムが

```
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
append([], X, X).
:- mode append2(+, ?, -).
append2([A|X], Y, [A|Z]) :- append2(X, Y, Z).
append2([], X, X).
```

(a) CP source program

```
:-fastcode.
:-public append/8.
:-mode append(?,?,?,+,?,-,+,+).
append(A,B,C,D,E,F,G,H) :-
  ulist(A,I,J), ulist(C,K,L), unify(I,K),
  D>0, !, M is D-1, append(J,B,L,M,E,F,nd,H).
append(A,B,C,D,E,F,G,H) :-
  unil(A), unify(B,C), D>0, !,
  E=[$(I,J,F,nd)|J], incore(I).
append(A,B,C,D,[$(E,F,G,H)|F],
  [$(append(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).

:-public append2/8.
:-mode append2(?,?,?,+,?,-,+,+).
append2(A,B,C,D,E,F,G,H) :-
  cpwait(A,I), D>0,
  '$$$append2'(I,B,C,D,E,F,G,H).
append2(A,B,C,D,[$(E,F,G,H)|F],
  [$(append2(A,B,C,I,J,K,L,I),J,K,L)|G],H,I) :-
  incore(E).
'$$$append2'([A|B],C,[A|D],E,F,G,H,I) :- !,
  J is E-1, append2(B,C,D,J,F,G,nd,I).
'$$$append2'([],A,A,B,C,D,E,F) :- !,
  C=[$(G,H,D,nd)|H], incore(G).
```

(b) Object program in Prolog-10

Fig. 2 The effect of mode declaration

Table 1. CP Benchmark on DEC2060

Program	Processing (*1)	Reductions	Suspensions	Time(*2)/RPS(*3) (compiler without mode)	(compiler with mode)	(interpreter)
Append	B	502	0	---	---	2313 / 217
(500+0	BD100	502	0	88.7/ 5660	54.8/ 9160	---
elements)	D	502	0	79.0/ 6350	43.0/11700	---
	P			15.8/31600	11.9/42200	188 / 2670
Merge	B	202	0	---	---	1005 / 201
(100+100	BD100	202	0	42.9/ 4710	28.7/ 7040	---
elements)	D	202	0	38.4/ 5260	23.6/ 8560	---
	P			8.3/24300	6.0/25300	73.7/ 2740
Bounded buffer	B	204	0	---	---	1473 / 138
(size=1)(*4)	BD100	204	200	147 / 1390	121 / 1690	---
	D	204	200	143 / 1430	119 / 1710	---
Bounded buffer	B	204	0	---	---	1470 / 139
(size=10)(*4)	BD100	204	20	60.2/ 3390	47.6/ 4290	---
	D	204	20	56.3/ 3620	43.3/ 4710	---
Primes	B	2778	8445	---	---	80521 / 35
(2 to 300)	BD100	2778	73	966 / 2880	769 / 3610	---
(without output)	D	2778	0	886 / 3140	689 / 4030	---
	P			216 /12900	188 /14800	2969 / 936
Quicksort	B	378	2225	---	---	20233 / 19
(50 elms)	BD100	378	0	125 / 3020	96.5/ 3920	---
	D	378	0	119 / 3180	91.3/ 4140	---
	P			21.3/17700	17.3/21800	246 / 1540

*1 B = CP breadth-first mode;
BD100 = CP bounded depth-first mode (bound=100);
D = CP depth-first mode;
P = Prolog-10 compiler ('fastcode' mode) and interpreter.
*2 In milliseconds. Overhead for timing has been excluded.
*3 RPS = number of reductions per second. An RPS value does not count reductions in guards. RPS values of Prolog programs were calculated using the number of reductions of CP programs.
*4 A Prolog counterpart does not exist.

低速なのは，bounded bufferのためにプロセス切替えが頻繁に起こるからである．実際，バッファの大きさを10にすれば，効率は最大75% 向上している．

6．まとめ

軽くて高速なＣＰのコンパイラを作成した．Prologで書かれていて，Prologをターゲット言語としているので，保守が容易で，しかも実行時にPrologとインターフェースをとることができる．移植性がよいので，すぐれたProlog処理系さえあれば，直ちにＣＰを使って並列プログラミングの実験を始めることができる．

参考文献

[Clark and Gregory 84] Clark, K.L. and Gregory, S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College, London, 1984.

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

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

[梅村84] 梅村恭司，Lisp を Pascal に翻訳し実行する，移植性の高い記号処理系，情報処理学会記号処理研究会資料29-1，1984.

ストリームAND 並列型言語の一拡張について

竹内彰一
（財）新世代コンピュータ技術開発機構

1. 序

Prologに代表される論理型プログラミング言語の特徴は従来より次のように言われている。

① 論理変数を扱えること
② バックトラッキングがあること
③ 計算と証明が同一の過程で行なわれること

①はユニフィケーションがあることと言っても良いが、この特徴により論理型言語では部分的にしか定義されていないデータ構造を自然に扱えるようになっている。この特徴はしばしば d-list などを例にとり、論理型言語が強力なリスト処理言語であるという主張の根拠となってきた。②もまた Prolog に特有な特徴であり、この特徴により探索型アルゴリズムが非常に容易に書けるようになっており、これがしばしば Prolog が人工知能向き言語であるという主張の根拠となっている。③は論理型言語の理論的基盤ならびに、実行のセマンティクスを与えるものとなっており、これを用いたメタ・プログラミングなどが可能となっている。

一方、一般に並列プログラミング言語の特徴は次のように考えられる。

① 計算の順序を半順序的に指定できること
② 複数の計算の間の同期を表現できること

並列論理型言語は論理型言語を基に並列プログラミングが可能なように拡張された言語とここではとらえる。並列論理型言語は一般に2つの流れに分けて考えることができる。1つの流れは Relational Language[1], Concurrent Prolog [2] (以後 CP と呼ぶ),PARLOG[3],KL1[4] などのいわゆるストリームAND 並列型言語族であり、他の流れは T-Prolog,Delta-Prologなどの逐次 Prolog (Concurrent Prolog, Delta-Prolog と混同しないように文脈に応じて従来の Prolog を逐次 Prolog と呼ぶことがある) をベースとする言語族である。本論文ではストリームAND 並列型言語について、論理型言語、並列型言語両者の特徴を保存したままの拡張について考える。

2. ストリームAND 並列論理型言語

2.1 概要

ストリームAND 並列論理型言語の特徴は論理AND で結合された複数のリテラルのリゾリューションを並列に行なうこと、および、並列に解かれている個々のリテラルはそれらの間で共有されている論理変数を通じて情報を交換し合うことができるという点にある（図1）。図1を図2に示すようなネットワークで表現すると分り易い。

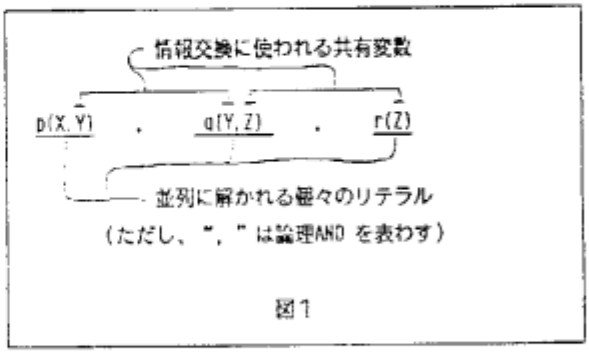


図1

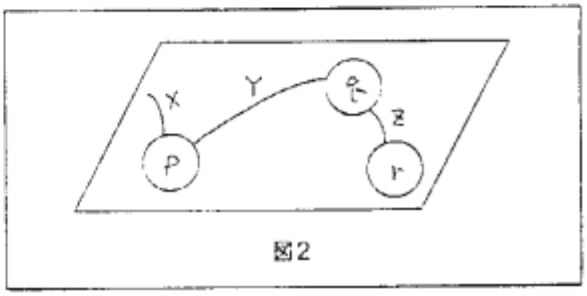


図2

図2において、円で囲まれたものはリテラル、円をつなぐアークはリテラル間で共有されている論理変数を表わしている。平面は1つの論理世界を表わしており、同じ平面上にあるすべてのリテラル（円）は互いに論理的にAND で結合されていることを意味する。この図は平面を1つの世界、円をその世界に属するアクティビティ、アークをアクティビティ間の通信チャネルとして読むこともできる。

個々のリテラルのリゾリューションは次のシンタックスをもつガード付き節で行なわれる。

H :- < ガード部 > | < ボディ部 >.

ここでHはヘッド、"|"はコミット・オペレータと呼ばれ、<ガード部> 、<ボディ部> はそれぞれリテラルの並びである。

ゴール・リテラルが与えられたとき、それのリゾリューションは次のようにして行なわれる。今、ゴール・リテラルをAとすると、このAとユニフィケーションが可能なヘッドをもつガード付き節の各々について、そのガード部の真偽が調べられ、ガード部が真になった節が1つだけ選ばれて、ゴール・リテラルはその節のボディ部に置換えられる。これを図2のリテラル p(X,Y) を例に示すと図3のようになる。図3には、リテラル Pがリゾルブされる前と後の平面を示した。このリテラル置換操作においてコミット・オペレータは、一度節が選択されると他の節による置換の可能性をすべて放棄する（このことを committed choice という）という役割を果す。

以上、ストリームAND 並列論理型言語に共通な計算過程の概略を図式的に説明した。ここで説明したレベルより下の計算メカニズム、

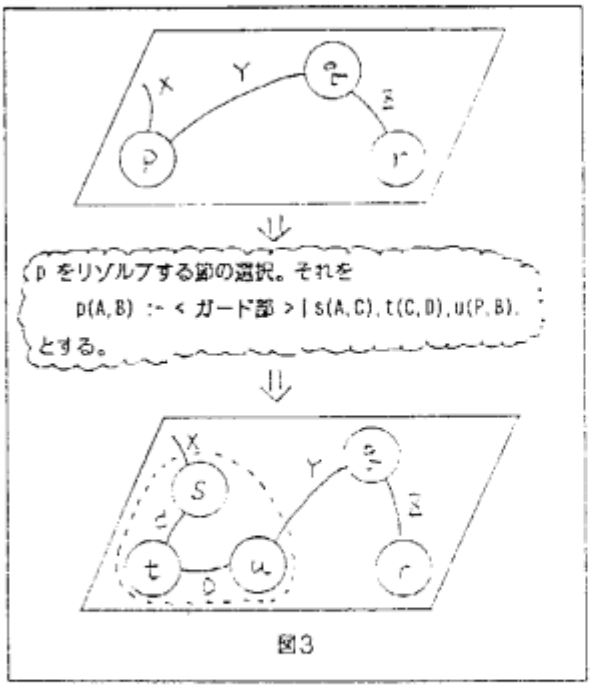


図3

特にガード部の計算と同期についてはストリームAND 並列性に基づく言語でも異なる。次節では、2つの代表的言語、 CP と PARLOG との比較を種々の観点から行なう。

2．2　同期機構とプログラミング・スタイル

CP の同期機構は読出専用標記（read only annotation）により実現される。読出専用標記は変数に対して付加されるものである。読出専用標記の付加された変数（読出専用変数と呼ぶ）の値を読むときに、その値が未定義であれば少なくともその値の principal functorが決まるまで待つことを意味する。一方、PARLOGの同期機構はモード宣言で実現される。モード宣言は述語の各引数の使われ方を指定するものである。この指定には入力と出力の2つのモードがある。ゴール・リテラル中の入力モードと指定されている引数は CP の読出専用と同じように扱われ（厳密には若干異なる）、同期を実現する。ここでは細かい相違には注目しないで、両者のプログラミング・スタイルに対する影響を考える。両者の基本的相違は、 CP の同期機構が変数に対して付加されるのに対し、PARLOGの同期機構は述語の引数に付加されることである。序に述べたように、論理型言語では内部に未定義変数を含んだデータ構造が扱えるが、 CP ではこのようなデータ構造中の変数に対しても読出専用標記を付加することができる。このことを一般化して述べれば、 CP では同期機構はデータの側に埋め込まれていると考えられる。一方、述語を手続きとみなせば、PARLOGでは同期機構は手続きの側に埋め込まれていると考えられる。このように CP と PARLOG では同期機構を仕掛ける対象がデータと手続きというプログラミング上相対するものに実はなっている。この大きな相違は両者のプログラミング・スタイルの相違として当然顕著に現れてくると思われる。 CP の特異なプログラミング・テクニックとして報告されている『 protected date 』[5] はこのプログラミング・スタイルの差を示す好例と言える。

2．3　ガード部の計算とその実行効率

一般に CP のプログラムではゴール・リテラルに対しある節がそのゴールをリゾルブできるかどうか調べる過程で、ヘッド・ユニフィケーションやガード部の計算のときに、ゴール・リテラル中の変数をインスタンシエートすることがある。このようなインスタンシエーションはその節が選択されるまで（その節のコミット・オペレータの処理が成功して終了するまで）その節の処理の中だけでのみ見え、それ以外からは見えないようにしなくてはならない。これを実現する手法はいくつか提案されている[6,7,8] 。

一方、PARLOGのプログラムはコンパイル時に徹底したモード解析によりヘッド・ユニフィケーションやガード部の処理でゴール中の変数をインスタンシエートしないようなプログラムに変換される（変換できないプログラムはコンパイル時にハネられる）。従って PARLOG には上のような問題は実行時には生じず、同じプログラムを走らせた場合、実行効率は CP よりも良くなる。

2．4　メタ・プログラミング

プログラムをデータとして扱ったり、データとしてもっているプログラムを評価したりするプログラミングをここではメタ・プログラミングと呼ぶことにする。メタ・プログラミングは言語のインタプリタを作成したり、デバッガ／トレーサを作成したり、また複数の仮説世界を管理してある定理を各世界で証明してみるといった人工知能的応用プログラムを書いたりするときにしばしば現れる。このようなプログラミングがある言語で可能かどうかは基本的に自分自身のインタプリタを自分で書けるかどうかに掛かっている。 CP で書かれた CP のインタプリタはすでに存在し、また CP で書かれた Prolog インタプリタも存在する。一方 PARLOG については、PARLOGで書かれた Prolog インタプリタは存在するが、 PARLOG で書かれた PARLOG のインタプリタはまだない。実行時にモード解析に相当することを行なうようなインタプリタが PARLOG で容易に書けるかどうか問題である。また、プログラムをデータとして扱い、それをインタラクティブに変更し、かつ評価するというような応用を考えるとモード解析というプログラム全体に及ぶ解析は応用上大きな制約になると思われる。

以上、3つの観点から CP と PARLOG の比較を行なったが、それらをまとめるならば、データという手続き間でやりとりされる受動的なものに同期機構を埋め込むとか、インタラクティブにプログラムを修正したりするようなメタ・プログラミングのような根本的に動的なものを対象にしたプログラミングは CP で行なう方が容易であり、一方、データ・フロー解析が事前に行なえ、実行時にそれが変化しないようなプログラムについては PARLOG で書いた方が実行効率が良いと言える。

3．　拡張

問題解決システム等への応用を考えると、ストリームAND 並列論理型言語は、問題解決の対象が複雑な動作系（例えば、原子炉制御系、プラント・システムなど）であるときに、相互に作用を及ぼし合いながら並列に動作する系の構成要素をモデル化し易い点、また複数の推論を並列に進めることができるという点で並列問題解決シ

ステムを構築する上で有力な言語と言える。しかし一方で今まで提案されているストリームAND 並列論理型言語はすべて『 commited choice』という思想のもとに、 Prolog に代表される論理型言語の問題解決などへの適性である序で述べた特徴②、すなわち自動的な探索機能を失っている。この結果、動作系の可能な状態で構成される状態空間の探索を必要とするようなプラン生成問題の記述がストリームAND 並列論理型言語では容易ではなくなっている。本章では以上の考察をもとに CP を例に、ストリームAND 並列論理型言語に探索機能を付加する方向への拡張を考える。

3．1　探索機能

Prologにおける探索機能はゴール・リテラルに対し１つの節を適用しても、他の節の適用の可能性を保持し、失敗の場合に備えるというようにして実現されている。一方、ストリームAND 並列論理型言語では、ゴール・リテラルに対し適用する節が１つ選択されると必ず他の節の適用の可能性を捨てている（ committed choice ）。ストリームAND 並列論理型言語に探索機能を付加するためにはまずこの commited choice という概念を弱める必要がある。

本論文ではこのために述語を２種類に分類することを提案する。第１の種類は AND関係と呼ばれ、これに属する述語はガード付き節だけから定義されてなくてはならない。第２の種類は OR 関係と呼ばれ、これに属する述語はコミット・オペレータをもたない節（無ガード節と以下呼ぶ）だけから定義されている。１つの述語がガード付き節、無ガード節の両者を用いて定義されることはない。しかしガード付き節および無ガード節ともに右辺に並ぶリテラルは AND 関係に属するリテラル、 OR 関係に属するリテラルの両者が混じってかまわない。

AND 関係に属するゴール・リテラル、 OR 関係に属するゴール・リテラルのリゾリューションはそれぞれ次のようになる。AND 関係に属するゴール・リテラルのリゾリューションは必ずガード付き節を用いて CP の通常のリゾリューションと同じ様に行なわれ、選択された節はコミット・オペレータにより他の節を選ぶ可能性を完全に放棄する。一方 OR 関係に属するゴール・リテラルのリゾリューションは必ず無ガード節を用いて行なわれる。無ガード節によるゴール・リテラルのリゾリューションはヘッドとゴールとがユニフィケーション可能な節の右辺のリテラル列でゴール・リテラルを置換えることにより行なわれる。このとき無ガード節はコミット・オペレータを持たないから他の節によるゴール・リテラルのリゾリューションの可能性を捨てない。従って、例えばゴール・リテラルAとｎ個の無ガード節、 Hi :- <Bi>.　i=1,...,n （ただし <Bi> はリテラルの並び）の各ヘッド Hi とのユニフィケーションがすべて可能な場合には、リゾリューションの結果ゴール・リテラルAは、 <B1> or ... or <Bn>　で置換えられる。今ゴール・リテラルAは元の世界（平面）では、 P1,....,Pm　と論理的にAND で結合されていた、すなわち、元の論理世界が A and P1 and ... and Pm　であったとすると、このAのリゾリューションにより、新しい世界は、(<B1> or ... or <Bn>) and P1 and ... and Pm となる。ところで、

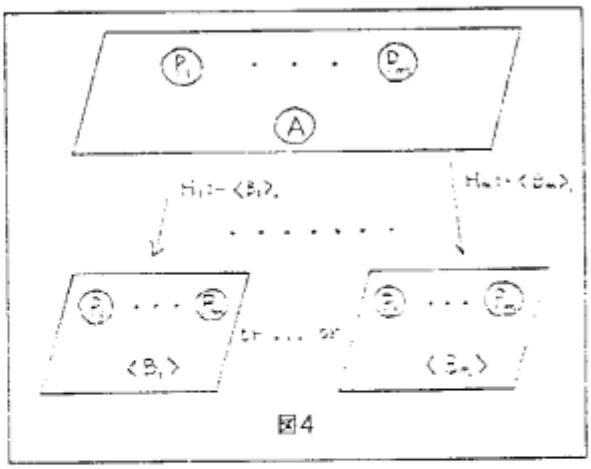


図4

A and P1 and ... and Pm

⇓

(<B1> or ... or <Bn>) and P1 and ... and Pm

= (<B1> and P1 and ... and Pm)
or (<B2> and P1 and ... and Pm)
⋮
or (<Bn> and P1 and ... and Pm)

であるから、一般に、 OR 関係に属するゴール・リテラルのリゾリューションでは適用可能な無ガード節が複数個あった場合には結果として元の世界が節の数だけ枝別れし、互いに論理 OR の関係にある複数の世界が生じることがわかる。以上のことを図式的に示すと図４のようになる。このように世界が枝分れした場合については、それぞれの世界のそれぞれのリテラルについてそれが AND関係に属するか、 OR 関係に属するかに応じて今まで述べたリゾリューションを続けていくことになる。

以上述べたように、この拡張においては述語を AND関係と OR 関係との２種類に分類し、 OR 関係に属するリテラルのリゾリューションにおいてはあらゆる可能性を保持する計算機構を導入した。この結果ユーザはこの OR 関係を用いて探索型アルゴリズムを容易に書けるようになる。

OR関係を用いることにより探索機能を導入することが可能になったが探索の戦略については並列探索や逐次探索等のいろいろな可能性がある。 CP をこのように拡張した言語処理系を DEC-10 Prolog 上に試作したが、この処理系では逐次探索を採用している。どのような探索戦略をとるかはインプリメンテーションとも深くかかわっており今後検討をして行く予定である。

CPに逐次 ANDを加えたものをこのように拡張し、 OR 関係の探索戦略を逐次探索（上に書かれた節が優先、深さ優先）としたものは CP と Prolog の両方の Superset になっていることに注目すべきである。実際 Prolog はこの拡張された言語の中で論理 ANDおよび節を逐次AND および無ガード節に限定したサブ言語に一致する。

3．2　プログラム例

拡張された言語のプログラム例を２つ示す。シンタックスはほとんど CP と同じである。ただし各述語定義の先頭に太字で示されて

いる、:- and_relation(X). や:- or_relation(Y). は X, Y がそれぞれ AND関係、OR 関係であることを示している。また AND関係を定義する節においてガード部が空のときにコミット・オペレータを書くのを省略してもよいという GP プログラムの慣習を用いている。

プログラム例1はこの言語で書かれたこの言語自身のインタプリタである。プログラム中の and_relation(X),or_relation(X) はそれぞれリテラル Xが AND関係か OR 関係かを判定する述語（AND関係）である。また、clauses(X,Cls) はリテラル Xの定義節のリストを第2引数 Clsに返すシステム述語である。

プログラム例2はブロックの積替問題を解くプログラムである（紙面の都合で一部の述語の定義は省略されている）。このプログラムではブロックなどを AND関係によりメッセージを受取ったときに行動を起こすようなオブジェクトとしてモデル化し、積替えアクション（ブロックを別のブロックの上に移すアクションあるいはブロックを床に下すアクションの2種類がある）を選ぶ部分をOR関係で記述している。この問題は可能なアクションが構成する状態遷移空間を探索することが必要であるが、このプログラムでは OR 関係で記述された節がそれを自動的に実現している。

4. 謝辞

日頃御指導をいただく ICOT 第2研究室古川室長にここで感謝の意を表する。また ICOT 研究所の諸氏並びに KL1処理系試作検討会のメンバーの方々との議論は非常に有益であった。ここに感謝する。

5. 文献

[1] K.Clark, S.Gregory: A Relational Language for Parallel Programming,Proc. of ACH Conf. on Functional Programming Languages and Computer Architecture, 1981.

[2] E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT TR-003, 1983.

[3] K.Clark, S.Gregory: PARLOG: Parallel Programming in Logic, Research Report DOC84/4, 1984.

[4] K.Furukawa et al.: The Conceptual Specification of the Kernel Language version 1, ICOT TR-054, 1984.

[5] L.Hellerstein, E.Shapiro: Implementing Parallel Algorithm in Concurrent Prolog: The MAXFLOW Experience, International Symposium on Logic Programming, Atlantic City, 1984.

[6] 宮崎他：Concurrent Prolog のシーケンシャル・インプリメンテーション(Shallow Bindingによる多環境の実現)，日本ソフトウェア科学会第1回全国大会

[7] 市吉他：Concurrent Prolog のシーケンシャル・インプリメンテーション(Deep Binding による多環境の実現)，日本ソフトウェア科学会第1回全国大会

[8] 田中他：Concurrent Prolog のシーケンシャル・インプリメンテーション(copy 方式による多環境の実現)，日本ソフトウェア科学会第1回全国大会

```
$$$$  Program example 1  $$$$

:- and_relation(solve(_)).
  solve(true).
  solve((X,Y)) :- solve(X),solve(Y).
  solve(X) :- and_clauses(X,Cls) | and_resolve(X,Cls).
  solve(X) :- or_clauses(X,Cls) | or_resolve(X,Cls).
:- and_relation(and_resolve(_,_)).
  and_resolve(X,[C|Cls]) :- and_unify(X,C,B) | solve(B).
  and_resolve(X,[_|Cls]) :- and_resolve(X,Cls).
:- and_relation(and_unify(_,_,_)).
  and_unify(X,(X :- G|B),B) :- solve(G) | true.
  and_unify(X, (X :- B), B).
:- or_relation(or_resolve(_,_)).
  or_resolve(X,[(X :- B)|Cls]) :- solve(B).
  or_resolve(X,[_|Cls]) :- or_resolve(X,Cls).
:- and_relation(and_clauses(_,_)).
  and_clauses(X,C1) :- and_relation(X) | clauses(X,C1).
:- or_relation(or_clauses(_,_)).
  or_clauses(X,C1) :- or_relation(X) | clauses(X,C1).


$$$$  Program example 2  $$$$

:- and_relation(plan_form(_)).
  plan_form(Ans) :- pf([],Ans).
:- or_relation(pf(_,_)).
  pf(Actions,Actions) :-
     state_trans(S,[],Actions), object_world(S7).
  pf(Actions,Ans) :- pf([A|Actions],Ans).

:- and_relation(state_trans(_,_,_,_)).
  state_trans([],Sofar,[]) :- write_actions(Sofar).
  state_trans(S1,Sofar,[A|Actions]) :-
     update(S1,A,Sofar,S2), state_trans(S2,[A|Sofar],Actions?).
:- and_relation(update(_,_,_,_)).
  update(S1,A,Sofar,S3) :- action(A,S1,S2),
     not_same_block(A?,Sofar?), transform(A?,S2,S3).
:- or_relation(action(_,_,_)).
  action(to_block(block(X),Y,block(Z)),
                  [(block(X),movable(on(Y))),(block(Z),clear)|S],S).
  action(to_place(block(X),Y,place(Z)),
                  [(block(X),movable(on(Y))),(place(Z),clear)|S],S).
:- and_relation(transform(_,_,_)).
  transform(to_block(block(X),Y,block(Z)),
                [(block(X),c_on(block(Z))),(Y,c_clear),
                 (block(Z),c_under(block(X)))|S],S) :-
     X? \= Z? | true.
  transform(to_place(block(X),Y,place(Z)),
                [(block(X),c_on(place(Z))),(Y,c_clear),
                 (place(Z),c_under(block(X)))|S],S) :-
     X? \= Z? | true.

:- and_relation(object_world(_)).
  object_world(S) :-
     distribute(S,[(a,A),(b,B),(c,C)],[(p,P),(q,Q),(r,R)]),
     object(a,A?,clear,block(b)), object(b,B?,block(c),place(p)),
     object(p,P?,block(t),ground), object(q,Q?,clear,ground),
     object(c,C?,clear,place(r)), object(r,R?,block(c),ground).
:- and_relation(object(_,_,_,_)).
  object(Id,[movable(on(Under))|S],clear,Under) :-
     object(Id,S?,clear,Under).
  object(Id,[on(Under)|S],On,Under) :- object(Id,S?,On,Under).
  object(Id,[under(On)|S],On,Under) :- object(Id,S?,On,Under).
  object(Id,[clear|S],clear,Under) :- object(Id,S?,clear,Under).
  object(Id,[c_on(Under)|S],On,_) :- object(Id,S?,On,Under).
  object(Id,[c_under(On)|S],_,Under) :- object(Id,S?,On,Under).
  object(Id,[c_clear|S],_,Under) :- object(Id,S?,clear,Under).
  object(Id,[],_,_).

:- and_relation(distribute(_,_,_)).
  distribute([(T,M)|S],Bs,Ps) :-
     var_target(T) |
     or_send(T,M,Bs?,Ps1,Bs2,Ps2), distribute(S?,Bs2?,Ps2?).
  distribute([(T,M)|S],Bs,Ps) :-
     send(T,M,Bs?,Ps1,Ps2,Ps2) | distribute(S?,Bs2,Ps2).
  distribute([],[(a,[on(block(b))]),
                 (b,[under(block(a)),on(block(c))]),
                 (c,[under(block(b)),on(place(r))])],
                [(p,[]),(q,[]),(r,[under(block(c))])]).
:- and_relation(send(_,_,_,_,_,_)).
  send(block(T),M,Bs,Ps,Bs2,Ps) :- send(T,M,Bs,Bs2).
  send(place(T),M,Bs,Ps,Bs,Ps2) :- send(T,M,Ps,Ps2).
:- and_relation(send(_,_,_,_)).
  send(T,M,[(T,[M|Ts])|Assoc],[(T,Ts)|Assoc]).
  send(T,M,[(U,V)|Assoc],[(U,V)|Assoc2]) :-
     send(T,M,Assoc?,Assoc2).
:- and_relation(or_send(_,_,_,_,_,_)).
  or_send(block(T),M,Bs,Ps,Bs2,Ps) :- or_send(T,M,Bs,Bs2).
  or_send(place(T),M,Bs,Ps,Bs,Ps2) :- or_send(T,M,Ps,Ps2).
:- or_relation(or_send(_,_,_,_)).
  or_send(T,M,[(T,[M|Ts])|Assoc],[(T,Ts)|Assoc]).
  or_send(T,M,[(U,V)|Assoc],[(U,V)|Assoc2]) :-
     or_send(T,M,Assoc?,Assoc2).
```

Concurrent Prolog のシーケンシャル インプリメンテーション

-- Shallow binding方式による多環境の実現 --

宮崎敏彦　竹内彰一　古川康一
（財団法人　新世代コンピュータ技術開発機構）

（1）はじめに

Concurrent Prolog （以下CP）はストリーム and 並列を基礎とした論理型言語である。CPでは各プロセス間のコミュニケーションは共有変数によって実現される。また、非決定的プログラミングをcommit operator によって実現している。 CP処理系の効率的な実現を考える場合の主な問題点は、

(a) suspend 時のスケジューリング
(b) 候補節間の並列実行

であろう。

CP処理系はすでに幾つかのものが存在している。例えば、Shapiro はDEC-10 Prolog 上にインタプリタを書いており[Shapiro 83]、上田．近山らはPrologにコンパイルするコンパイラをDEC-10 Prolog で記述している [上田 84]。しかし、これらの処理系は、まず (a)については busy-wait 方式を用いており、効率の点で問題があった。また (b)は並列実行ではなく逐次実行であり、その意味で完全なCPの処理系ではなかった。

本稿では、2章で上記 (a)、(b) についてもう少し細かい議論をし、3章でCPの一般的な計算モデルを述べ、4章では (b)に対して我々が採用した shallow binding方式について具体的に説明を加える。また5章では shallow binding方式と他の方式を比較し、その長所と短所について議論する。

（2）効率的実現の為の問題点

1章で述べた2つの問題点の内 (a)に関しては Shapiro がすでにそのアイディアを示している。それは、「suspend した計算はsuspend の原因となった変数と関係付けられ、他の計算によってその変数が具体化された時、関係付けられている計算は活性化される。」というものであり、本インタプリタでは、suspend queue と呼ぶqueue を持つデータ・タイプ（以下suspと呼ぶ）を導入することで具体的に実現している。

(b) の候補節間の並列実行については以下の様な問題点がある。下の例を考えよう。

```
goal:    p(X,X)        /*  X は未定義  */
clause1: p(1,Y) :- guard1(Y) | body1.
clause2: p(2,Y) :- guard2(Y) | body2.
          program 1
```

上の例で、head unificationはどちらのclauseも成功する。そのとき Xの値はclause1 においては1、clause2 では2となり、かつこれらは互いに矛盾するものである。このため各clauseの実行を並列に行なう為には。各clauseがそれぞれ異なる束縛環境を持たなければならない（このような個々の束縛環境をlocal 環境と呼ぶ）。しかし問題はもう少し複雑である。下の2つの例を考えよう。

```
goal: p(f(A))      /* Aは未定義 */
clause1: p(X):-q(X),… | ….
clause2: q(f(1)):-… | ….
      program 2

goal1: q(X) & p(X), … | …,    /* Xは未定義 */
clause3: q(f(A)).
clause4: p(f(1)):-… | ….
      program 3
```

program 2 においてclause2 がcommitすると変数 Aの束縛情報はclause1 に公開される。しかしclause1 はまだcommitしていないのでgoalの世界では Aはまだ未定義でなければならない。つまり Aに関する束縛情報はclause1 のlocal 環境として保存されなければならないのである。一方program 3 においては、clause4 がcommitすると Aに関する束縛情報はgoalの世界に公開されlocal 環境として保存する必要はない。このように、束縛情報を公開する際、その変数がどの世界に属すかによってlocal 環境に保存するか否かが変わる為、各local 環境を管理できなければならない。

（3）計算モデル

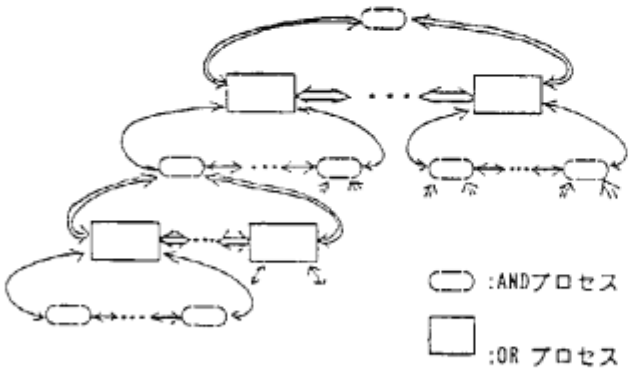
Concurrent Prolog （以下単にCPと呼ぶ）の計算過程はPrologと同様、and-or木で表現することができる。

CPにおける各goalはand nodeに対応し、あるgoalに対応する幾つかのclauseはそれぞれor node で表される。ここではこのand nodeとor node をAND プロセス、ORプロセスとよぶ。

図．1は、あるgoalを解く実行過程の一場面を表したものである。←→でつながれるループをand ループと呼び ⟸⟹ でつながったループをorループと呼ぶ。

and ループは一群のAND プロセスの並びとそれらの親であるORプロセスよりなる。and ループに含まれるAND プロ

セスがひとつでも失敗した場合はそのand ループ全体の失敗である。orループはそのorループに含まれるORプロセスのひとつでも成功すれば全体が成功し親のAND プロセスが成功となる。逆に、失敗したORプロセスはループから除去される。 and ループ上にあるすべてのAND プロセスが成功するとcommitされて、親のORプロセスは、そのbodyパートをAND プロセスとして展開する。



図．1　AND-OR木

（4）shallow binding による多環境の実現

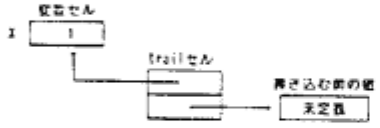
4．1） shallow bindingとenvironment-switching

2章で述べたように、候補節間を並列に実行する為には多環境を実現する必要がある。ここではそのためにshallow binding 方式を採用した。shallow binding 方式とは、祖先の変数に対する束縛情報をそのまま祖先の変数セルに書き込み、元の値を自分の guard 内に保存しておく。そして並列に実行されるべき他の環境の異なるプロセスに移る際、値を元に戻すというものである。

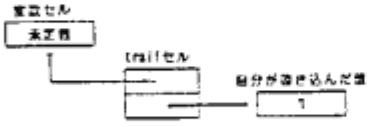
例を示そう。

```
goal:  p(X),q(X)      /* X は未定義 */
clause1:  p(1) :- guard | body.
```

program 4



図．2a

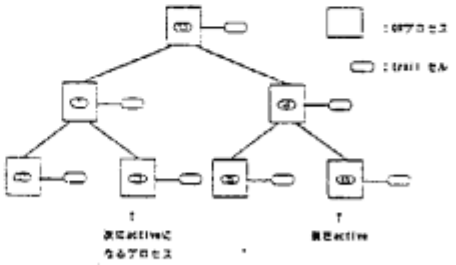


図．2b

program 4 で goal p とclause1 のhead unification では図．2aの様なtrail セルと呼ぶペアが作られる。しかし、clause1 はまだcommitを越えていないので、goal p にとってX は未定義でなければならない。このため、goal q を実行する際には上記のtrail セルを使って図．2bの様な値の入れ換えを行なわなければならない。

そしてふたたびclause1 のguard を実行する際には上記trail セルを用いて、自分の束縛環境に戻すわけである。この様な作業をここではenvironment-switching と呼ぶ。一般にtrail セルは各guard ごとにあり、ORプロセスがこれを管理する。

図．3の様なOR木を考えよう。（各ORプロセスはそれぞれtrail セルを幾つかずつ持っているものとする。）



図．3　OR 木

ORプロセス⑥が現在activeなプロセスとし、次にactiveになるプロセスが③であった場合、environment-switching によって保存された③の束縛環境を復元する為には、両者を結ぶパス上にある⑥，④，①，③のtrail セルを用いれば良い。（⑤のtrail セルは他のORプロセスがすべて⑨プロセスの子孫であるので使われない。）

ここで注意しなければならない事は、environment-switching を行なうためには現在activeなプロセスと次にactiveになるプロセスのAND-OR木上の関係を高速に認識できなければならない事である。そこで、ここでは各ORプロセス間にenvironment ポインタと呼ぶポインタを張り、どのプロセスからも常に現在activeなORプロセスに辿り着ける様にする。environment ポインタとは自分の直接の祖先か直接の子孫を指すポインタであり、子孫を指す場合はそのいずれかの子孫がactiveである場合である。図．4に例を示す。

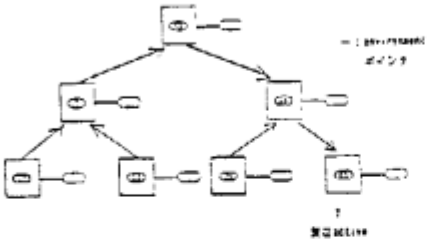
これによって、environment-switching は以下の様に表わせる。

① 次にactiveとなるORプロセスからenvironment ポインタを辿り、現在activeなプロセスまで行く。このとき辿ったenvironment ポインタの向きを逆にする。

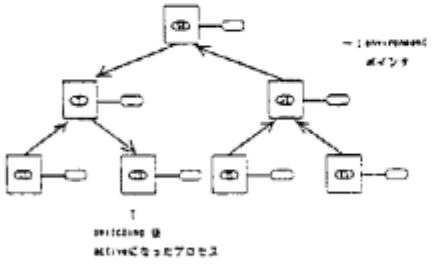
② 今辿ったポインタを逆に辿りながら、そのパス上にあるORプロセスのtrail セルを用いて値を入れ換える。

③ 自分自身に辿り着けば自分自身のenvironment ポインタをnil にする。（activeなプロセスのenvironment ポインタはnil とする。）

図．4の例でORプロセス③へのenvironment-switching 終了後の状態を図．5に示す。



図．4　environment ポインタのあるOR木



図．5　environment-switch後

4．2）local 環境の管理

program 2 の例では、clause1 がcommitした時、A が 1 であるという束縛環境はclause1 のguard 内にのみ公開され、goalの世界には公開されない。つまり変数A に対応するtrail セルは、clause1 に対応するORプロセスに登録される。これに対して、program 3 の例の場合clause4 がすでにcommitを越えていれば、clause3 のcommitにおいて、A に対応するtrail セルはexport時に消えてしまう。なんとなれば、変数A はgoal1 のlocal 環境に属しているからである。

一般には変数セル領域はそのclauseがcommitすると親の変数セル領域に属すようになる。上の二つの例で、変数A に対するtrail セルの取り扱いを区別するためには、その変数セルが所属するlocal 環境を識別できなければならない。この為には各local 環境にA という変数セルがあるか否かを調べれば良いのであるが、それでは非常に無駄が多いことは明らかであろう。そこでここでは各local 環境にそれぞれ一意に決まる番号を付け識別することにする。そしてcommitの際には、そのlocal 環境の番号は親のlocal 環境の番号への参照ポインタに変える。（local 環境番号にアクセスする為に、本インタプリタではundefined というデータタイプを未定義であり、かつその値部がlocal 環境番号を指すポインタである、と解釈する。local 環境番号は変数セルのallocation時に初期設定される。）

図．6において、①、②、③、④、⑤の変数セル領域は何回かのcommitによって同じlocal 環境に属している。

しかしこのままだとlocal 環境番号を得るためには毎回最悪 n回の参照ポインタを辿らなければならない。このようなオーバーヘッドを避けるために、一度辿った参照パスはその最後の参照パスで付け変えるようにする。

head unificationやexportの際の、“ある変数が、当該 and プロセスに属すか？”という判断は、実は“local 環境番号が同じか？”というチェックをすれば良いことになる。このように、このlocal 環境番号を用いることによって、head unificationやexportの際にtrail セルを作る（あるいは登録する）か否かが決定されるのである。

図．6　local 環境の管理

4．3）Unification

head unificationの処理の概要を表．1に示す。

表．1　unification

C \ P	wait	susp	wait(?)	susp(?)	term
wait	C.→REF(P) or P.→REF(C)	C.→REF(P) or P.→REF(C)	C.→HOME(P)	C.→HOME(P)	C.→P
susp	C.→REF(P) or P.→REF(C)	C.→REF(P) or P.→REF(C)	C.→HOME(P)	C.→HOME(P)	C.→P
wait(?)	P.→HOME(C)	P.→HOME(C)	SUSPEND	SUSPEND	SUSPEND
susp(?)	P.→HOME(C)	P.→HOME(C)	SUSPEND	SUSPEND	SUSPEND
term	P:=C	P:=C	SUSPEND	SUSPEND	unify(P,C)

ここで、 A or B は基本的にはどちらでも良いことを表わす。（local 変数同士のunification はゴミ集めを考えた場合注意する必要あり。）

SUSPEND はこのunification がsuspend することを意味する。この時suspend の原因となった変数が未定義ならば、それをsuspというデータタイプに変え、suspend recordを作る。suspend したgoalはそのsuspend recordが指すsuspend queue に登録される。変数がsuspであれば、すでにあるsuspend queue の最後にそのgoalをつなげる。

```
goal: p(X?)          /*  X は未定義  */
clause: p(1) :- guard | body.
```

program 5

program 5 の例では、goal p はsuspend し、変数X の持つsuspend queue に登録される。登録されたプロセスはその変数に値が入ることのよってactivateされる。

変数とのunification は、shallow binding では図．7

の手順で表わすことができる。 read only 変数と通常の変数とのunification は、通常の変数の変数セルから read only変数の変数セルへ、読み出し専用を意味するread-only-reference ポインタを張る。このように、読み出し専用とはその変数セルへの参照ポインタの属性である。

X ：変数セル
X↑：変数セルの番地

```
if instantiated(X) then
    Xとunify;
elseif X は自分のlocal 環境内の変数セル
  then
    X↑:=unify する相手;
  else （祖先の変数セルのはず）
    trail セルをallocate;
    X↑と Xの値をtrail セルに登録;
    trail セルをtrail フレームに登録;
endif;
```

図. 7 変数とのunification

4．5） Commitment

commit操作は、commitしようとしているORプロセスがおこなう。操作の手順は以下のとおり。

①commitしようとしているORプロセスの兄弟のORプロセスを殺す（これをabort と呼ぶ）。

②guard 内での束縛情報を公開する（これをexportと呼ぶ）。

③commitしたORプロセスに対応するclauseのbody部をAND プロセスとして展開し、各プロセスをactive queueに登録する。もしbody部が空なら、祖父のORプロセスがcommit。

exportは当該ORプロセスが管理しているtrail セル全てに対して、図．8を行なう。

```
X↑: trail セル中の変数X の変数セルへのポインタ
V  : trail セル中のセーブした値。
     ( 変数X の現在のglobalな値)

if  X↑が指す変数セルが親のAND プロセスに属す
  then
    現在のX の値とV をunify ;
  else （もっと祖先の変数セル）
    祖父であるORプロセスの管理するtrail セルとして登録;
endif ;
```

図. 8 export

例を示そう。

```
goal: p(f(A))            /* Aは未定義*/
clause1: p(X):-g(X),…|….
clause2: q(f(1)):-…|….
        program 6
```

program 6 の場合clause2 におけるcommitでは変数A の値が 1であるという情報をclause1 のguard 内に対してのみ公開でき、clause1 のcommitで始めてgoalの世界に対して公開される。つまりclause2 のcommit操作ではA に関するtrail セルを持つclause1に対応するORプロセスに登録する。 commitにおけるunify の仕事はhead unificationと以下の点で異なる。

①suspend queue を持つ変数セルに値が入ったならば、そのエントリをすべてactive queueに登録する。

②commitのunify でsuspend した場合は、当該ORプロセスを対応する変数セルのsuspend queue に登録する。

（5）議論

本稿では候補節の並列実行の為の多環境の実現方式としてshallow binding 方式について述べた。多環境の実現手段としては、その他eager copying 方式，deep binding方式等が考えられる。そこでここではそれらの方式とshallow binding 方式とを比較する。

まず長所としては、

a)eager copying に比べ、goalのcopyが必要ない分unification やexportの処理は速い。

b)deep bindingに比べ、dereference のアルゴリズムが単純であり、guard のネストに依存せず速い。

また短所は、次のgoalを実行するためにはenvironment switching と呼ぶ処理が必要であり、このコストは現在のgoalと次にscheduleされるgoalとのOR木上の距離に依存する。しかし、例えばbounded-depth-first といった、scheduleを工夫することによってenvironment switchを減らす（あるいはコストを安くする）ことは可能である。

謝辞

多くの示唆を与えていただきました ICOT 近山氏に深謝いたします。

参考文献

[Shapiro 83] Shapiro.E.Y A Subset of Concurrent Prolog and Its Interpreter. ICOT TR-003

[Ueda 84] K.Ueda,T.Chikayama Efficient Stream/Array Processing in Logic Programming Language FGCS'84

[西百 84] 西百，他 Concurrent Prolog のシーケンシャル・インプリメンテーション(Deep Binding による多環境の実現)
日本ソフトウェア科学会 第1会大会 3D-3 1984

[田中 84] 田中，他 Concurrent Prolog のシーケンシャル・インインプリメンテーション(Copy 方式による多環境の実現)
日本ソフトウェア科学会 第1会大会 3D-4 1984

[上田 84] 上田，近山 並列論理型言語の実用処理系
日本ソフトウェア科学会 第1会大会 3D-5 1984

Concurrent Prolog ウィンドウ・トレーサについて

藤田 悦之*、 竹内 彰一**、 近藤 浩康**

* 日本ビジネスオートメーション株式会社、 **（財）新世代コンピュータ技術開発機構

1．はじめに

Concurrent Prolog(以下 CP) は並列性を持った論理型プログラミング言語である。CPのプログラムでは、並列に動作する複数個のプロセス（ゴール）が共有変数によってストリーム通信を行ないながら実行される。

逐次動作するシステムの動きを観察するには逐次トレースが有効な手段であるが，並列動作するものに対しては動作過程を正確に把握することが難しい。CPでは複数個のゴールが同時に実行されるため，なんらかの支援機能の必要性が要請されてきた。

我々はこうしたCPプログラムの動きを解析するtoolとして、CPプログラムの実行過程を複数のウィンドウによってモニタするHulti-Window-Tracer System，および実行の履歴を検索する Computation-Tree-Viewerを開発した。本稿ではこのHulti-Window-Tracer Systemおよび Computation-Tree-Viewerについて簡単に解説を行なう。

2．Hulti-Window-Tracer System（以下 MWiS）

MWISはプログラムの動作状態のモニタを目的としたシステムである。本システムは論理変数のモニタ機能やゴールのモニタ機能を提供する。さらに，ユーザインタフェースとしてマルチウィンドウシステム[Shapiro Takeuchi 83]を採用し，CPプログラムの対話的な実行機能を提供している。

2－1．システムの機能概要

MWTSの機能は以下のようにまとめられる。

①論理変数をモニタする機能

トップレベルの質問（CPプログラムを起動させるゴール）が含む論理変数（多くの場合プログラムが求めた解がインスタンシエートされる）にインスタンシエートされるtermをCPプログラム実行中にウィンドウに表示できる。

②プロセスをモニタする機能

トレースの指定とはトレースするterm，ウィンドウのディスプレイ上の位置と大きさをキーボードから入力する事である。CPプログラムの実行中に発出されたゴールとユーザが前もってトレース指定したtermとユニファイ可能な場合、このゴールを所定のトレース・ウィンドウに表示することができる。

③論理変数にインスタンシエートする機能

トップレベルのゴールが含む論理変数に対して、キーボードから入力したtermを有限長あるいは無定長リストの雛形としてインスタンシエートできる。

④マルチウィンドウの管理

すべてのウィンドウに対してその位置と大きさをいつでも変更することができる。ウィンドウは、表示しきれなくなると自動的にスクロールアップするが，コマンドによってユーザが指定したウィンドウ内にあるテキスト行を自由にスクロールアップ、スクロールダウンさせることで，隠れて見えなくなった行を再び見ることができる。[Shapiro Takeuchi 83]

2－2．MWTSの実例による機能説明

本章では”エラトステネスのふるい”による素数を求めるプログラム（プログラム１）を実行例として，本システムの機能の詳細な説明をする。

プログラム－１．エラトステネスの素数計算プログラム

```
primes(J) :- integers(2,I) ,sift(I?,J).

integers(N,[N | S]) :- K := N + 1 | integers(K,S).

sift([P | I],[P | R1])
    :- filter(P,I?,R)  ,sift(R?,R1).

filter( _,[],[]).
filter(P,[Q | L],K)
    :- 0 =:= Q mod P | filter(P,L?,K).
filter(P,[Q | L],[Q | K])
    :- 0 < Q mod P | filter(P,L?,K).
```

ここで、primes(J) は論理変数Jに素数のストリームをインスタンシエートする。 integers(K,I) は論理変数Iに K以上のストリームをインスタンシエートする。 sift(I?,J)は論理変数Iに２以上の自然数のストリームを受け取

り、論理変数Ｊに素数のストリームをインスタンシエートする。　またfilter(P,X,Y) は論理変数Ｘのストリームからｐの倍数を除いたストリームをＹにインスタンシエートする。

プログラム１によって，質問 ?- cp primes(P) を実行すると，変数Ｐに素数からなるリストがインスタンシエートされる。siftが展開されるごとに，新しく見つかった素数Ｐを第一引数とするプロセスfilterが起動されていく。

実行例－１．ＣＰシステムの実行例

```
?-cp primes(S) ,outstream(S?).
*** outstream: 2
*** outstream: 3
*** outstream: 5
*** outstream: 7
```

実行例１はCPシステム上でprimesを実行したものである。outstream は，primes(S) と並列に動作する。そして読出し専用表記付き論理変数Ｓ？がプロセスprimesによってインスタンシエートされると，その値を印刷して次にインスタンシエートされるまで待ち状態になるプロセスである。

ゴールprimesを本システムMWTSを使って実行すると論理変数Ｓのインスタンシエーションをウィンドウの中に見ることができる（実行例２）。ウィンドウのラベルprimesはこのウィンドウの識別名で、この種のウィンドウ（システムの機能①）のラベルはユーザが自由な名前を与えることができる。この場合、実行例１のoutstream の役割を，識別名primesのウィンドウが果たしている。このウィンドウが素数を計算するプログラムのゴール primes(S)にある論理変数Ｓをモニタしている。

実行例２は素数が論理変数にインスタンシエートされる過程での他のプロセスのインスタンシエーション状況を示している。実行例２のウィンドウfilterは異なるプロセスを区別していない。同じ述語名で同じ個数の引数（同じarity ）を持つ別のプロセスのトレース情報をプロセスごとに分離する方法としてマルチ・ウィンドウ・トレーサ実行中での動的トレース設定がある。

次ぎに静的トレース設定と動的トレース設定の機能について説明する。

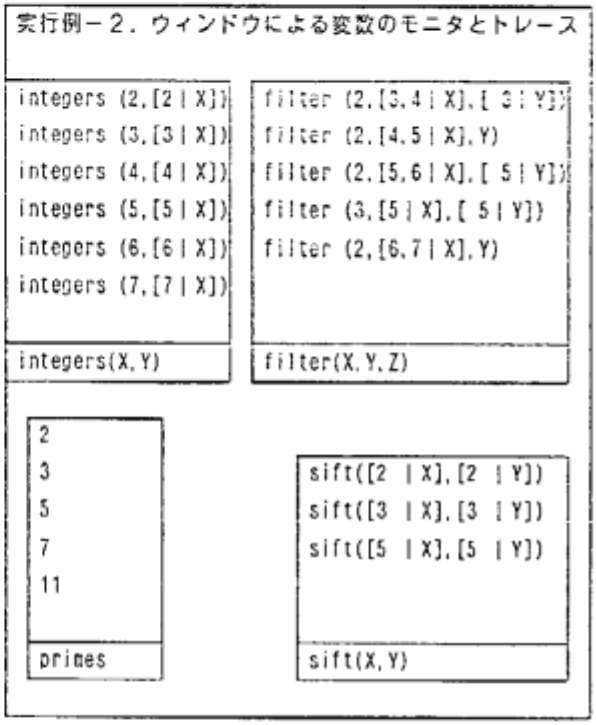
i)静的トレース設定

プログラム１に対してはintegersとsiftに適用される。ユーザが指定するtermとユニファイ可能なゴールを予め設定した一つのウィンドウに表示する。静的トレース指定の例としてMWTS起動時に次のコマンドを入力する。

```
trace(integers(X,Y),(2,2,10,5)) .
```

（意味）　ディスプレイ画面上の座標(2,2) を左上隅とする，幅１０，高さ５のウィンドウを開き，そこにゴールintegers(X,Y) とユニファイできるゴールを表示する。

実行例－２．ウィンドウによる変数のモニタとトレース



ii) 動的トレース設定

プログラム１に対してはfilterに適用される。動的トレースの静的トレースとの違いは，MWTSを起動して始めてのトレース情報が検出されるまで，ウィンドウが開かれない点である。また，動的トレースを指定するとき，ウィンドウを区別する為の引数位置を添える。

```
trace(filter(X,Y,Z),(14,2,15,6),1).
```

（意味）　ゴールfilter(X,Y,Z) とユニファイできるゴールが検出されたら，trace 述語の第三引き数で指定された数に対応するfilterの引数の位置にあるtermごとにウィンドウを自動的に開いてそこに表示する。ディスプレイ画面上の座標(14,2)を左上隅とする，幅15，高さ6 のウィンドウを一番最初のウィンドウの位置とする。

すなわち動的トレースは、あらかじめユーザが指定したリテラルとユニファイ可能なトレース情報がMWTSの動作中に現れると、既に開かれているウィンドウに表示されてい

る，ウィンドウを区別するための引数のteraとユニファイ可能かどうか調べて次の二つの場合に分ける。

a)もしユニファイ可能ならば、そのウィンドウに表示する。
b)もしユニファイ不可能ならばこの時点で新しいウィンドウを開いてそこに表示する（実行例３）。

動的トレース・ウィンドウの生成によって、プロセスfilterごとのウィンドウを実現することができた。

実行例－３．動的トレースウィンドウの生成

```
filter(2,[3,4 | X],[ 3 | Y])
filter(2,[4,5 | X],Y)
filter(2,[5,6 | X],[ 5 | Y])
filter(2,[6,7 | X],Y)
filter(2,[7,8 | X],[ 7 | Y])
filter(2,X,Y)
```

```
filter(3,[5 | X],[ 5 | Y])
filter(3,[7 | X],[ 7 | Y])
filter(3,X,Y)
```

システムの機能③を利用すると実行中のプロセスの論理変数に対して直接インスタンシエートすることができる。
プログラム２のsiftの論理変数にキー入力した自然数を直接インスタンシエートできる。CPプログラムを作成する場合、全体が完成していない場合でもこの機能によって、部分的な動作確認をすることができる。これにシステムの機能②（トレース）と併用することで、キーボードからのインスタンシエーションによるプロセスの進行を観察することができる。

３．Computation Tree Viewer for Concurrent Prolog

CPプログラムの実行過程はAND-木として表現できる。Computation-Tree-Viewer （以下CTV)はAND-木を実際に生成しながらCPプログラムを実行するCPインタプリタを内蔵している。CPプログラムの実行が成功して終了した後に，AND-木の任意のノードを観察する。

AND-木は、メタＣＰインタプリタ（CPで記述されたCPインタプリタ）にAND-木の生成機能を追加して実現した。AND-木の構造はCPの言語仕様を反映した形式を持っている。AND-木の構造は以下のようになっている。各ノードはひとつのホーン節を持っていて頭部とガード部及び本体を含んでいる。そしてガード部及び本体のゴールがさらに別のホーン節を構成する。AND-木の構造はホーン節の再帰的な連鎖として展開される。

CTV はAND-木のノードを指すポインタをひとつ持っている。最初ポインタはAND-木のrootを指していて，ユーザ・コマンドによってこのポインタを木構造に沿って移動させることができる

AND-木の，あるノードから別のノードへの移動は簡単なコマンドによって実行される。各ノードでは、AND-木の最上位にあるノード（質問ゴール）からの最短経路を表示すると共にそのノードにあるインスタンシエートされたHorn節を表示する。コマンドによって親，兄弟，子供の関係にあるノードへ移動してHorn節を観察することができるから導出の前後関係を理解することに役立つ。

サーチ命令は、ユーザの指定したゴールを現在ユーザに見えているノードから下にあるノードへの道筋に沿って検索する。そして発見された全てのノードにポインタを張る。後に visitコマンドによってそれらのノードを巡回する事ができる。ノードを検索して Bug（プログラムの虫）の可能性のある導出を行っているHorn節を敏速に捜し出す。

主なＣＴＶのコマンドは以下の通りである。

ｂｏｄｙ，<番号>．	ボディ部にある<番号>で指定されるノードへ移動する。
ｇｕａｒｄ，<番号>．	本体にある<番号>で指定されるノードへ移動する。
ｐａｒｅｎｔ．	親ノードへポインタを移動する。
ｓｅａｒｃｈ，<項>．	現在ポインタが指しているノードから葉にむかって全て捜して，発見されたノードにポインタを張る。
ｖｉｓｉｔ．	ｓｅａｒｃｈによって張られたポインタの指すノードを順に見る。

実行例－４．ＣＴＶによる素数プログラムの観察

```
-------------------------------------------------------------------------------
primes([2,3,5,7]) :- integers(2,[2,3,4,5,6,7]),sift([2,3,4,5,6,7],[2,3,5,7]).
|: body,1.
Pass> root~b1~me.
integers(2,[2,3,4,5,6,7]) :- 3:=2+1|integers(3,[3,4,5,6,7]).
|: parent.
Pass> root~me.
primes([2,3,5,7]) :- integers(2,[2,3,4,5,6,7]),sift([2,3,4,5,6,7],[2,3,5,7]).
|: body,2.
Pass> root~b2~me.
sift([2,3,4,5,6,7],[2,3,5,7]) :- filter(2,[3,4,5,6,7],[3,5,7]),sift([3,5,7],[3,
5,7]).
|: search,filter(_,_,[5|_]).
<search> * found the object that is unifiable to filter(X,Y,[5|Z]) *
<vscnedule> * visiting scnedule *
-------------------------------------------------------------------------------
Pass> root~b2~b1~b1~b1~me.
Pass> root~b2~b2~b1~me.
Pass> root~b2~b1~b1~me.
-------------------------------------------------------------------------------
Pass> root~b2~me.
sift([2,3,4,5,6,7],[2,3,5,7]) :- filter(2,[3,4,5,6,7],[3,5,7]),sift([3,5,7],[3,
5,7]).
|: visit.
Pass> root~b2~b1~b1~b1~me.
filter(2,[5,6,7],[5,7]) :- 0<5 mod 2|filter(2,[6,7],[7]).
|: visit.
Pass> root~b2~b2~b1~me.
filter(3,[5,7],[5,7]) :- 0<5 mod 3|filter(3,[7],[7]).
|: visit.
Pass> root~b2~b1~b1~me.
filter(2,[4,5,6,7],[5,7]) :- 0=:=4 mod 2|filter(2,[5,6,7],[5,7]).
|: visit.
Pass> root~b2~me.
sift([2,3,4,5,6,7],[2,3,5,7]) :- filter(2,[3,4,5,6,7],[3,5,7]),sift([3,5,7],[3,
5,7]).
|: end.
* epilogue Computation-Tree-Viewer *
```

４．今後の問題

本稿で報告した２つのシステムは，早急に機能確認をするためのプロトタイピングシステムであるため，メタＣＰインタプリタ（CPで記述されたCPインタプリタ）にトレース機能およびAND-木の生成機能を追加して実現した。しかしながらメタCPインタプリタはCPの言語仕様上，コミットされなかったHorn節の動作についてはモニタすることができないため，プログラムの動作を把握するには未だ不十分である。今後は，この点も含めてシステムを拡張していく予定である。

謝　辞

多くの示唆を与えていただきました古川康一ICOT第二研究室長を初めとするICOT第二研究室の諸氏，Computation-Tree-Viewer に関する有益な助言を与えていただいたICOT第三研究室の黒川利明氏，近山隆氏，そして富士通国際情報社会科学研究所の田中二郎氏，日本電気 C&Cシステム研究所の上田和紀氏に深謝いたします。

参考文献

[Shapiro 83] Ehud Y.Shapiro,"A Subset of Concurrent Prolog and Its Interpreter,"ICOT TR-003 (1983).

[Shapiro Takeuchi 83] Ehud Y.Shapiro & Akikazu Takeuchi, "Object Oriented Programming in Concurrent Prolog, " ICOT TR-004 (1983).

[Takeuchi 82] Takeuchi, "Let's Talk Concurrent Prolog", ICOT TM-0003 (1982).

[Shapiro 82] Ehud Y.Shapiro,Algorithmic Program Debugging" ,The MIT Press.Cambridge,Massachusetts.

Mandala IIの拡張機能
— ユーザ定義推論エンジンの実現方式 —

大木 優、古川 哲一、竹内 彰一、国藤 進、安川 秀樹、宮崎 敏彦
(新世代コンピュータ技術開発機構)

1. 緒言

人間は問題に直面すると、持っている豊大な知識および種々の戦略や推論法を使って問題を解決しようとする。例えば、数学の証明問題を解く場合と最適の予想をする場合とでは、人間の推論法は異なっているだろう。そのため、人間の知的活動を計算機上に実現する場合には、人間と同様にはなくの推論法を自由に実現することが望ましいと考えられる。Mandala は人間の知的活動を計算機上に実現するために開発された言語で、そのために様々な特徴ある機能を持ち、さらに拡張されつつある。本報告では、新しく拡張した機能の一つであるユーザ定義推論エンジンの組み込み機能について述べる。この機能は、人間が様々な推論法を持つことを反映している。

ユーザ定義推論エンジンの組み込み機能とは、ユーザ自身で記述した推論エンジンをMandala の推論エンジンとして組み込むことを可能とする機能である。ユーザ定義推論エンジンの組み込み機能の利点は次の通りであると考えられる。

(1) ユーザ自身で、ユーザ専用の推論エンジンを組み込むことが可能となる。
(2) システムとして複数の推論エンジンを容易に提供することが可能となる。
(3) 推論エンジン自体の機能拡張が容易になる。

2. Mandala の概要

Mandala は、stream-and並列論理型言語KL1 (Kernel language 1) のユーザ言語および問題解決システムや知識表現システムを構築するためのシステム記述言語として位置づけられる。Mandala では、複雑な知識を表現し操作するために、複数のプログラミング技法を可能としている。

(1) 論理プログラミング
KL1 が論理型言語であるため、論理型言語の特徴を生かしたプログラミングが可能である。
(2) オブジェクト指向プログラミング
問題の対象領域に登場するものの機能や知識をモデル化し、プログラミングできる。
(3) データ指向プログラミング
データ変数への参照、更新時に特定のプログラムを実行できる。
(4) ルール指向プログラミング
プロダクション・システム形式のルール・プログラミングが可能である。

なお、KL1 は、CP(Concurrent Prolog) とPP(Pure Prolog) を融合した言語である。

2.1 Mandala の基本構成要素

Mandala の基本構成要素は、KL1 の節から作られる単位世界(Unit World)と、CPのプロセスから作られる実体(Instance)の２つである。それぞれを図１のようなディスクおよび円を用いて図示する。

(1) 単位世界　　(2) 実体
図１ Mandala の基本構成要素

(1) 単位世界
プログラミング・システムとしては、管理のための最小単位であるモジュールに対応し、オブジェクト指向プログラミングでは実体を作り出す鋳型として扱われる。単位世界の節は次のように記述される。

〈単位世界名〉(〈節〉).

(2) 実体
動的な個体を表わすのに用いられる。

2.2 Mandala のリンク

Mandala では、図２に示すように単位世界や実体の関係を示すために４つの基本リンクがある。

(1) instance-of リンク
単位世界と実体間を関係づける。オブジェクト指向プログラミングでは、実体の動作手続（メソッド）が単位世界に記述されているとみなす。
(2) is-aリンク
単位世界間の概念の階層関係を表わす。概念階層間の性質の継承 (inheritance)は自動的に行なわれる。
(3) part-of リンク
より小さい実体を部品として持つような複合実体を

表現する。

(4) manager-ofリンク

単位世界とそれを表現する実体との関係を示す。特に、この実体を管理者(manager) と呼ぶ。基本的に単位世界には管理者が必要である。

3. Mandala の実体の表現

3.1 実体の表現

一般に、オブジェクト指向プログラミングのオブジェクトは内部状態を持ち、メッセージによって活性化される。Mandala の実体は、実体への入力列をゴールとみなして単にCPのプロセスと考えることができる。それは次の形で表わされる。

instance (<名前>, <入力列>, <世界>).

ここで、<名前>は実体の識別子であり、<入力列>は実体が受け取るメッセージ列である。<世界>は、実体と関係する単位世界名と実体固有の内部状態を保持する。このinstance自身はCPで次のように定義される。

```
instance(Name,[Message | Input],World):-
    simulate(Name,Message,World,NewWorld),
    instance(Name,Input?, NewWorld?).
instance(Name,[],World).
```

第1節のsimulateはMessage をWorld の世界で解き、新しい世界NewWorldを返す述語である。Message の解き方はsimulate述語の定義に従っており、この意味でsimulate述語を推論エンジンと見ることができる。simulate述語の定義を変えることにより、種々の推論を行なう実体を作ることが可能となる。

3.2 CP用simulate述語

simulate述語の例として、CPの節を実行するsimulate述語を説明する。図3にそれを示す。

①：ゴールがA，B又はA&Bならば、それらを2つのsimulateに分割する。

②：ゴールがシステム述語ならば、それを実行する。

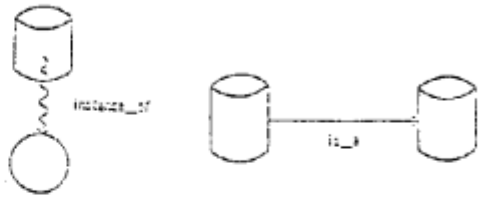
③：最新のWorld を見つけ、実体に関係している単位世界より、ゴールと同じ関数子を持つ節およびis_a リンクでつながっている上位の単位世界を見つける。見つかった節の中でガード部が解けた（simulate-resolve述語）節のボディ部を解く（simulate述語）。

④：見つかった節の中からヘッドとunify するものを選び、ガード部を解く。

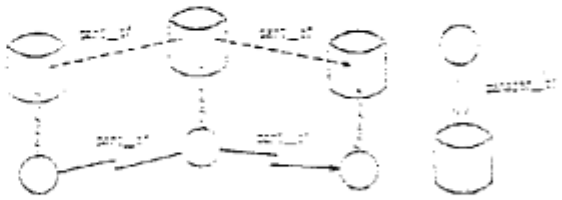
⑤：どの節でもガード部が解けなければ、上位の単位世界の節を捜す。

⑥：節をヘッド、ガード部、ボディ部の3つに分割する。

ここで、世界は変更歴をすべて保持するhistory 方式で実現している。図3では、Allworldが世界のすべての履歴をリストとして保持している。



(1) instance_ofリンク　(2) is_a リンク



(3) part_ofリンク　(4) Manager_ofリンク

図2　Mandala の基本リンク

4. ユーザ定義推論エンジンの組み込み機能

前章で述べたように、simulate述語を取り替えることによって、種々の推論が可能になることがわかった。ユーザ定義推論エンジンの組み込み機能は、ユーザ自身がsimula

```
①  simulate(Name,true,Allworld).
   simulate(Name,(X,Y),Allworld) :-
           simulate(Name,X,Allworld),
           simulate(Name,Y,Allworld).
   simulate(Name,(X&Y),Allworld) :-
           simulate(Name,X,Allworld) &
           simulate(Name,Y,Allworld).
②  simulate(Name,A,Allworld) :- system(A) | call(A).
③  simulate(Name,A,Allworld) :-
           search_newest_world(Allworld,[Newworld|_]),
           select_clauses_from_parent(A,Cs,Newworld?,Super_world) |
           simulate_resolve(Name,A,Cs,B,Allworld,Super_world),
           simulate(Name,B?,Allworld).

④  simulate_resolve(Name,A,[C|Cs],B,World,Super_world) :-
           simulate_unify(A,C,C,B),
           simulate(Name,C?,World) | true.
   simulate_resolve(Name,A,[C|Cs],B,World,Super_world) :-
           simulate_resolve(Name,A,Cs,B,World,Super_world) | true.
⑤  simulate_resolve(Name,A,[],B,World,Super_world) :-
           select_clauses(A,Cs,Super_world,Super_world_1),
           simulate_resolve(Name,A,Cs?,B,World,Super_world_1) | true.

⑥  simulate_unify(A,(A,(C;B)),C,B).
   simulate_unify(A,(A,(B)),true,B).
   simulate_unify(A,(A),true,true).
```

図3　CP用Simulate述語

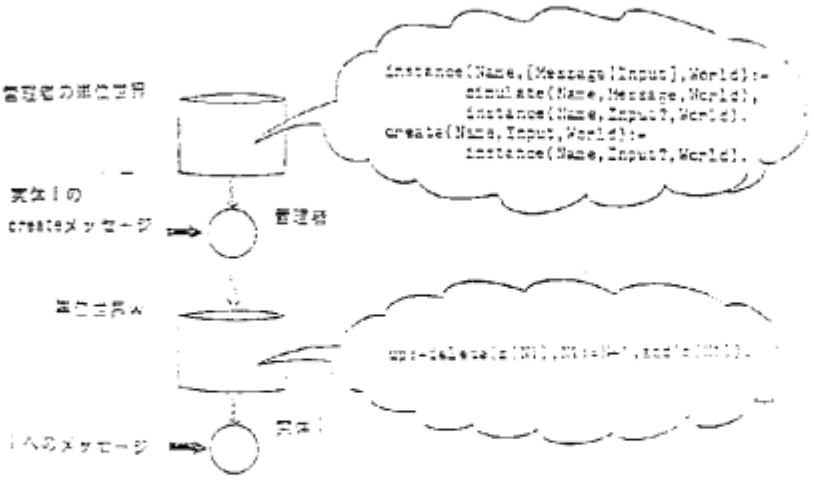


図4　実体の生成

te述語を定義することを可能にすることによって実現する。

4.1　実体の生成

まず、図4を用いて、実体の生成方法について説明する。実体を生成するためには、ある単位世界の管理者にcreateメッセージを送る。管理者の単位世界には、create述語の定義以外にinstance述語やsimulate述語などの定義が記述されている。create述語の定義は次の通りであり、

```
create(Name,Input,World):-
    instance(Name,Input?,World).
```

instance述語を解くプロセスを生成する。すなわち、生成された実体はこのinstance述語のプロセスに相当し、管理者の単位世界に記述されているsimulate述語に従ってゴールを解く。ユーザが、管理者の単位世界にsimulate述語を記述することが出来れば、ユーザ定義推論エンジンの組み込みが可能となる。

4.2　Mandala のWorld Map

管理者の単位世界は、実体から見ると、メタ・レベルになる。counter のupなどの動作を記述している応用プログラム・レベルはメタ・レベルに対してオブジェクト・レベルと呼ぶ。Mandala のWorld Map は、オブジェクト・レベルやメタ・レベルの単位世界の構成や実体の構成を記述したものである。図5にMandala のWorld Map の一部を示す。

図5に示すように、メタ・レベルはいくつかの単位世界から構成されている。それらは、次の3つの分類に分けることが出来る。

(1) instance述語やcreate述語を定義している単位世界（ 'Meta_Class' ）

(2) 節を選ぶselect_clauses 述語を記述している単位世界('select_Clauses')

(3) simulate述語を定義している単位世界（ 'CP_Simulate' やprover）

ここでproverはユーザが記述したsimulate述語を保持している単位世界である。メタ・レベルの単位世界をis_a でリンクする利点は、ユーザ定義推論エンジンを新しく組み込む際、例えばproverのように、節を選ぶ述語やinstance述語などの述語を定義する必要がなくなることである。これは、is_a リンクで単位世界をつなぎ、上位の単位世界を相続することができるからである。ユーザ定義推論エンジンproverは次のような CF(certainty factor: 確信因子) 付のルールを処理するsimulate述語を記述している単位世界である。

ヘッド :- (ボディ, CF) .

'CP_Simulate' には、前章で示したsimulate述語が記述されている。proverについては、次節で示す。なお、メタ・レベルの上のレベルは、メタ・レベルのis_a リンクを

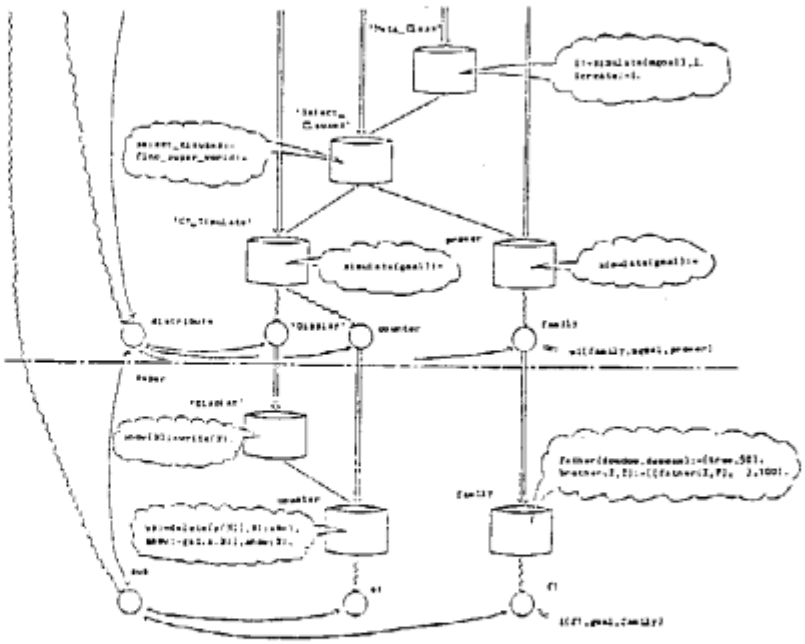


図5　Mandala のWorld Map(一部分)

管理したり、節を解く推論エンジンが定義されている。

4.3 prover用simulate述語

ユーザ定義推論エンジンの例として、prover用simulate述語を簡単に説明する。図6にその一部を示す。このsimulate述語は前項で述べたような節を持つ知識を使って、ゴールを満足するすべての解とそのCFを求める。

①：システム述語でないならば、prove 述語を使って、ゴールを求める。

②：prove 述語のゴールがシステム述語ならば、それを実行する。

③：複合のゴールが"，"でつながっていれば、分割して一つづつ解く。

④：最新のWorld を見つけ、実体に関係している単位世界からゴールとunify 可能なヘッドを持つ節を選ぶ。

⑤：見つかったすべての節を、CFを計算しながら一つづつ解く。

この述語を含む単位世界proverはMandala の初期処理時にメタ・レベルに組み込まれる。

```
simulate(Name,Goal,Allworld) :-
①       prove(Name,Goal,[],Tree,Tree,Proof,Allworld) |
        show_stream(Proof?).

prove(Name,A,[],((A<e)/100)<<[100],Tree,[Tree],Allworld) :-
②       prolog(cpsystem(A)) & call(A) | true.
prove(Name,A,[(C,TC<<CT)|D],((A<e)/100)<<[100],Tree,Chan,Allworld) :-
        prolog(cpsystem(A)) & call(A) |
        prove(Name,C,D,TC<<CT,Tree,Chan,Allworld).
prove(Name,(A,B,C),D,(TA & TB)<<[X|Y],Tree,Chan,Allworld) :-
③       prove(Name,A,[(B,C),TB<<[Y]|D],TA<<[X],Tree,Chan,Allworld).
prove(Name,(A;B),D,(TA & TB)<<[X,Y],Tree,Chan,Allworld) :-
        prove(Name,A,[(B,TB<<[Y])|D],TA<<[X],Tree,Chan,Allworld).
prove(Name,A,C,STree,Tree,Chan,Allworld) :-
        search_newest_world(Allworld,[Newworld|_]),
④       select_clauses_from_parent(A,Cs,Newworld?,Super_world),
        make_goal_with_cf(Goal,[],Mcs)),
        try_each(Name,Mcs,A,C,STree,Tree,Chan,Allworld) | true.
try_each(Name,[],_,_,[],Allworld).
try_each(Name,[(AD:-B)<CF|R],A,C,ST,T,Chan,Allworld) :-
        copy((A-C-ST-T,A1-C1-ST1-T1), AD = A1,
⑤       ST1 = ((AD <= TB)/CF<<[CF]) |
        prove(Name,B,[(cf(F,CF1,CF),TCF<<CCF)|CT],
                                    TB<<CF1,T1,Chan1,Allworld),
        try_each(Name,R,A,C,ST,T,Chan2,Allworld),
        merge(Chan1?,Chan2?,Chan).
try_each(Name,[_|Clauses],A,C,S,T,Chan,Allworld) :-
        otherwise | try_each(Name,Clauses,A,C,S,T,Chan,Allworld).
```

図6　prover用Simulate述語

4.4 demon の実現

あるゴールを解く前や解いた後に、特定の節を処理したいことがある。この様な節をdemon と呼ぶ。この様な場合、図7のように、demon のみ処理するsimulate述語を 'CP_Simulate' の下位にis_a でつなぐことにより、簡単にdemon を実現できる。before_simulateと after_simulateは、それぞれゴールを解く前と解いた後のdemon を処理する述語である。ユーザは、before# ゴール又はafter#ゴールというヘッドをもつ節をdemon として記述することが可能となる。

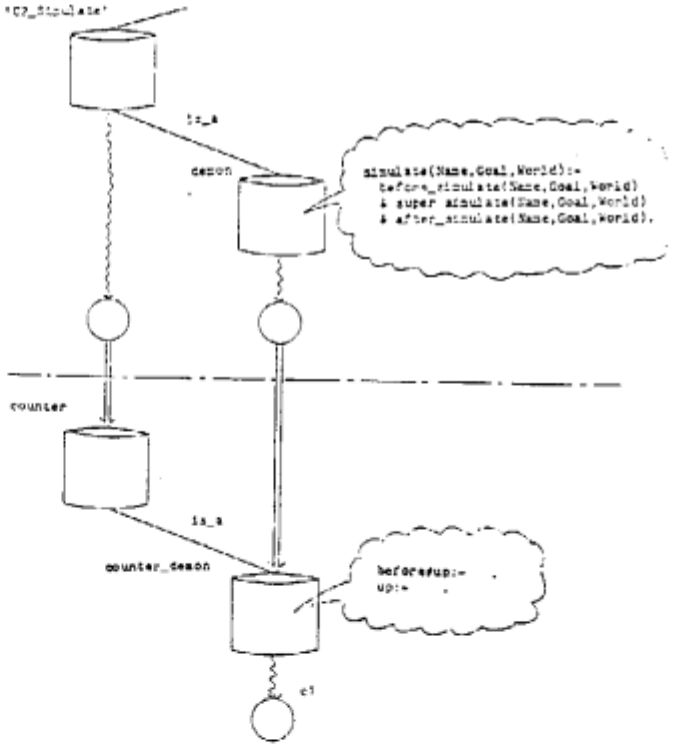


図7　demon の実現

5. 結言

ユーザ定義推論エンジンの組み込み機能により、メタ・レベルのsimulate述語をユーザが組み込むことが可能となった。メタ・レベルでis_a リンクが使えるので、ユーザはsimulate述語だけを記述すれば良い。節を選ぶための述語はシステム提供の述語を使うことができる。また、is_a リンクによりsimulate述語の継承の記述も容易となった。

6. 謝辞

本研究に関して有益な討論をしていただいたICOT第2研究室の近藤、北上の各氏、および日本電気・上田和紀氏に謝辞致します。

7. 参考文献

1) Furukawa,K, et al. ; MANDALA:A LOGIC BASED KNOWLEDGE PROGRAMMING SYSTEM, FGCS'84,1984,TOKYO

2) 竹内彰一 ; Concurrent Prologによるオブジェクト指向プログラミング, bit Vol.15 1388-1396

Prologによる
数式処理システム試作の構想

武脇 敏晃 ，宮地 泰造 ，國藤 進 ，古川 康一
（（財）新世代コンピュータ技術開発機構）

1 はじめに

既存の数式処理システムとしては、天体力学の計算を行うCANAL や、汎用システムとしてREDUCEやMACSYMA 、マイコンで動くものとしてはmu-Math などが有名である。これらのシステムの多くは、Lispによって開発されているが、Prologを用いると、「書換え規則」が容易にプログラムできるなどの長所が指摘されている[1]。Prologで開発された数式処理システムとしては、エジンバラ大学のBundy らによるPRESS [2),3)] がある。PRESS は、処理の手法として「メタレベル推論」と「オブジェクトレベル推論」という2つの推論を用いており、これによって複雑な処理の簡単化が行なわれている。しかし、人間と同様な方法で数式を解く場合には、PRESS が持っている単なる書換え機能等よりも、より高度なメタレベル推論である「未知数の置換え」などが必要となってくる。本稿では、「未知数の置換え」などのいくつかの手法を取入れると共に、処理制御にメタレベル推論を導入することによって、複数種類の複雑な数式をも処理できるシステムを試作したので報告する。また、数式処理に用いた推論および、その相対的な最適度を知る一つの機能として、ある問題に対する複数個の推論過程の適用規則を難易度とともに説明する機能を組込んだ。これより人間が数式を解く為の様々な方法とその経験的な難しさを表現するシステムの実現を試みた。

Prologと数式処理の関係については、文献4)に詳細な考察が行なわれている。

2 システムの特徴

本システムの主目的は、人間と同様な方法で数式を解くシステムの構築である。すなわち、数式を単に高速に解くことよりも、数式を解くために人間が使っている手法や推論過程を利用して数式を解き、その推論過程を難易度と共に明らかにすることを目的としている。

なお、本システムが対象とした数式は、高校教科書〜大学受験程度の一つの未知数を持った方程式である。

2.1 メタレベル推論

一般に、方程式を解くために使用可能な各種の規則を無制限に使用すると、探索空間が大きくなりすぎるため、探索空間を絞り込み、無駄な探索を押える必要がある。そこで我々は、数式処理の制御にメタレベル推論を導入した。メタレベル推論の主な特徴は、次の三つである。

1)複雑な処理制御の決定を容易に行なえる。
2)各数式処理規則のモジュール化が図れ、規則の追加変更が容易になる。
3)探索空間の絞り込みにより、無駄な探索を押える。

このメタレベル推論を実現するために、メタ述語であるdemo述語[4),5)] を導入した。これにより、上記の特徴の他に、メタレベル推論部と、オブジェクトレベル推論部が明確に分離し、拡張性に富むプログラムの実現が可能になった。

2.2 推論過程と難易度

推論過程に難易度を付与することによって、人間が経験的に活用している難しさの程度を表現することを試みた。

本システムが対象としている方程式は、受験問題という性質上、多くの場合に、各種の解法テクニックを駆使すれば解くことができる。また、一つの問題に対して複数の解法が存在する場合には、同じ問題でも、使用する規則により、問題の難易度に差が出てくる。そこで各種の解法規則にコストを与え、推論過程において使用された規則のコストを計算することにより、問題の相対的な難易度を考察することができる。また、異なる種類の問題に対して、相対的難易度の明確化も試みることができる。

問題の難易度を求めるために、次の3種類の重み付けを行った。

1)解法の手法の発見の難易度による重み付け。
2)解法手数による重み付け。
3)規則の連続使用時の動的な重み付け。

このような推論過程をたぐることによる問題の難易度付けは、CAI の教材の配列作成にも応用できると考えられる。

2.3 Prologによる実現

本システムは、インプリメント言語としてPrologを用いた。Prologは、

1)メタレベルの制御が容易に行なえる。
2)複数の解法が存在する時に、Prologのバックトラック機能により、自動的に他の解法を求められる。
3)数式処理の手法である書換え規則をPrologの節で表現できる。
4)書換え規則を適用するときのパターンマッチングが容易に行なえる。

など、数式処理システム実現に大変適している。

3．システムの実現法

3.1 処理の制御方法

本システムは、解決すべき数式を分析して、適用すべき適切な規則の推論を、メタレベル推論によって行なっている。メタレベル推論では、大別して、図 3-1のような２種類の制御を行なっている。型制御部は、方程式の型による分類、整形を行なう。規則制御部は、方程式に適用可能な規則により処理を行なう。

3.1.1 型制御

型制御部とは、方程式に存在する未知数の位置によるもので、以下の６つの型に分類する。ただし、一つの方程式に複数の型が存在する場合があり、複数個の解法も存在する可能性がある。なお、①でいう代数方程式とは、１変数の多項式を両辺に持つ方程式のことである。

①代数方程式型　　②無理方程式型
③分数方程式型　　④指数方程式型
⑤対数方程式型　　⑥三角方程式型

図 3-2の述語'solve' は、全体および型制御部の制御を行なうtop level のプログラムで、引数は順に、入力方程式、解こうとしている未知数、方程式の解、問題の難易度を示している。述語 'equation-type'は方程式の型を判断するもので、引数は順に、入力方程式、未知数、解こうとする未知数や型による条件等の制御情報、方程式の型を示している。述語'eq-type-solve' は型の整形処理を行うもので、引数は順に、方程式の型、方程式、制御情報、整形処理後の方程式、整形処理後の制御情報を示している。また、述語'method-demo' は、後述の規則制御を行うものである。

3.1.2 規則制御

規則制御部とは、入力された方程式の特徴と制御情報から、適用する規則を選択し適用を行うもので、以下の８種類の規則からなる。複数の規則が適用可能な時は、下記の1)から8)の順で適用される。ここでいう特徴とは、２次方程式や、相反方程式などである。

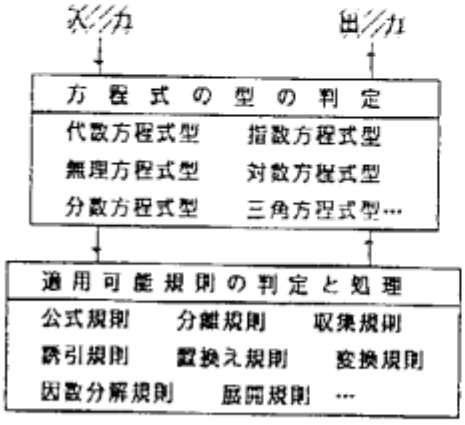


図 3-1　制御の流れ

1)公式規則

これは、ある特別の処理方法で、簡単に解くことができる場合に適用できる。そして、以下に述べる分離規則や収集規則などを用いても処理可能なものである。

例 1.　２次方程式の解の公式。

2)分離規則

これは、方程式に含まれる未知数の数が１個だけの時に適用される。これは、未知数の周囲にある関数の逆関数を両辺に適用して、未知数が一方の辺に分離するまで繰返す。

例 2.　$2^x = a \longrightarrow x = \log_2 a$

3)収集規則

これは、２つ以上存在する未知数の数を減少させる時に適用される。これは、分離規則が適用できる方向に持っていく効果がある。

例 3.　$(x+a)\cdot(x-a) \longrightarrow x^2 - a^2$

4)誘引規則

これは、収集規則のように、未知数の数は減少させないが、未知数を含む空間を小さくできる時に適用される。これは、収集規則が適用できる方向に持っていく効果がある。

例 4.　$\log_2(x+1) + \log_2(x-1) \longrightarrow \log_2(x+1)(x-1)$

5)置換え規則

これは、方程式が　h(f(X))=g(f(X)) である時に適用され、f(X)を他の未知数に置換える。

例 5. ４次方程式　$x^4 + 2x^2 - 8 = 0$を
$(x^2)^2 + 2x^2 - 8 = 0$とすると、x^2 を他の未知数（たとえば、y ）に置換えことができる。置換えると、二次方程式 $y^2 + 2y - 8 = 0$ になるので、これを解く。

6)変換規則

これは、未知数の数を減少させたり、未知数を含む空間を小さくはしないが、他の規則で処理できる形に

```
solve(Eqn,X,Ans,Cost):-
        equation_type(Eqn,X,Ctrl,Type),
        eq_type_solve(Type,Eqn,Ctrl,N_Eqn,N_Ctrl),
        method_demo(N_Eqn,N_Ctrl,Ans,[],His),
        cost(His,Cost).

eq_type_solve([algebraic],Eqn,Ctrl,Eqn,Ctrl):-!.
eq_type_solve(Type,Eqn,Ctrl,N_Eqn,N_Ctrl):-
        member(fraction,Type),!,
        fraction(Eqn,Ctrl,N_Eqn,N_Ctrl).
eq_type_solve(Type,Eqn,Ctrl,N_Eqn,N_Ctrl):-
        member(irration,Type),!,
        irration(Eqn,Ctrl,N_Eqn,N_Ctrl).
                :
```

図 3.2　述語solve

変換できる時に適用する。

例 6. 偶数次の相反方程式を x^2 で割ることにより、置換え規則が使えるようになる。

$ax^4 + bx^3 + cx^2 + bx + a = 0$

$\rightarrow \quad a(x+1/x)^2 + b(x+1/x) + c - 2a = 0$

7)因数分解規則

これは、方程式の持っている因数の１つを容易に見つけることができる時に適用する。

例 7. 奇数次の相反方程式は、x + 1という因数を必ず持っている。

8)展開規則

これは、方程式が展開可能な時に適用する。

例 8. $x(x+1)+x^2 = 4$

$\longrightarrow 2x^2 + x - 4 = 0$

この規則制御部の処理を行うプログラムを図 3-3に示す。述語'method-demo' の引数は順に、方程式、制御情報、方程式の解、これまでの規則使用履歴、処理終了時の規則使用履歴を示す。MD1)は、方程式が解けた時に処理を終える。MD2)は、方程式が分割可能な時、分割して処理を行う。MD3)は、MD1),MD2) 以外の時に規則適用処理を行う。述語 'apply' は、実際に規則の適用を行う部分であり、第１引数の世界に於いて、第２引数の述語を実行する。なお、第２引数の述語は、第１引数の世界に於ける述語である。（ここでの第２引数の述語solve は、前述のsolve とは別のものである。）

3.2 問題の難易度決定処理

人間は、一般に方程式をみると、およその難易度を推定できる。その難易度は式の特別の型や、形の複雑さで判断することが多い。そこである特別の数式（たとえば、２次方程式、複２次方程式、相反方程式など）を解くのにどの程度のコストがかかるかを、経験的に算出してみる。これにより、同じ方程式で複数の解法が存在するものについて、解法を知るとともにコストの比較もできる。

重み付けの方法は、次の３種類で行った。

1)一般に異なる規則は、異なる難易度を持つ。たとえば、x+a=b → x=b-a と x^2 =a → x=$\sqrt{a}$, x= -$\sqrt{a}$ では、後者の方が難しい規則である。そこで、難しい規則ほどコストが掛かるように重み付けを行った。

2)手数が掛かる規則の適用は、それだけ計算量が増し、難しいといえる。そこで、解法手数が増せばコストが掛かるように重み付けを行った。ただし、適用規則に対して独立にした。

3)規則の適用回数が増せば、一般に、難しさが増す。そこで、規則の適用回数が多くなればコストが掛かるように重み付けを行った。ただし、適用規則の種類に対して独立にした。

図 3-4の使用規則によるコストはそれぞれ 2と 6となり、

```
MD1)
method_demo(Eqn,Ctrl,Ans,_,[]):-
        end_check(Crtl,Eqn,Ans),!.
MD2)
method_demo(E1 && E2,Ctrl,Ans1 && Ans2,His,H3):-!,
        method_demo(E1,Ctrl,Ans1,His,H1),
        method_demo(E2,Ctrl,Ans2,His,H2),
        append(H1,H2,H3).
MD3)
method_demo(Eqn,Ctrl,Ans,His,N_His):-
        strategy(Eqn,His,World,Ctrl,Ctrl1),
        apply(World,solve(Eqn,Ctrl1,Int,N_Ctrl,H1),_),
        append(H1,His,H2),
        method_demo(Int,N_Ctrl,Ans,H2,H3),
        append(H1,H3,N_His).
```

図 3-3 述語method-demo

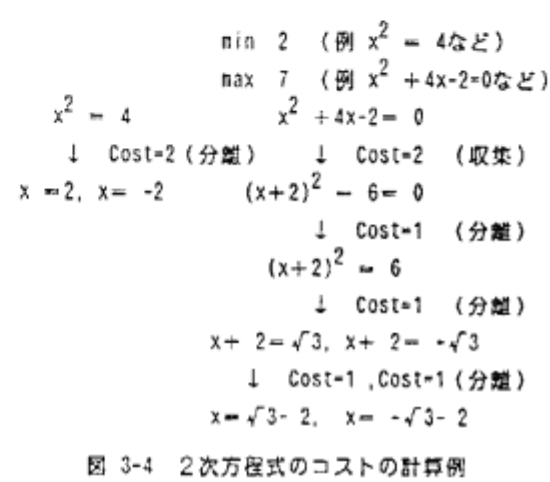


図 3-4 ２次方程式のコストの計算例

表 3-1 特別な方程式の難易度の幅

方程式名	難易度
２次方程式	2 7
３次方程式（因数定理）	14 →
４次方程式（複２次）	10 26
（相反）	25 40
（因数定理）	26 →
５次方程式（相反）	30 45
（因数定理）	41 →

これに規則適用回数による重みつけを行うと、それぞれ 2と 7になる。同様に、他の特別な方程式についてもコストの幅が付く。それを表 3-1に示す。

また、このコストを学習することによって、より最小コストになるように規則適用順序の制御を行えば、数式処理の最適アルゴリズムを実現することも可能である。

4. 実行例

以下の、複２次方程式でも、４次の相反方程式でもある、$4x^4 - 17x^2 + 4 = 0$ の解法例を示す。

推論過程において、適用規則番号や規則適用のコストに手数を考慮したコストなどの解法の説明を出力している。解法例を見ると、複２次方程式として解いた場合、使用規則数は、４回でコストが16であるのに対して、相反方程式として解いた場合には、使用規則数が９回でコストが36である。この場合、推論過程より複２次方程式として解くほうが易しいことがわかる。またこれは、人間の問題解決時の経験的な問題解決の難易度にほぼ一致している。

（方程式の入力）

```
| ?- test(4*x^4-17*x^2+4=0,x,A).

<<<Input Equation>>>
4*x^4-17*x^2+4=0

<<<Equation Type>>>
[algebraic] equation

Unknowns: x
```

◎複２次方程式として解いた場合

```
Rule No.  subst3  ( sub1 )  , Cost --->   4
substitute unknown1 for x^2
4*unknown1^2+ -17*unknown1+4=0

Rule No. for01 , Cost  --->  5
unknown1=4 , unknown1=1/4

replace( sub1 )
substitute x^2 for unknown1
x^2=4

Rule No. iso08 , Cost  --->  3
x=2 , x= -2

replace( sub1 )
substitute x^2 for unknown1
x^2=1/4

Rule No. iso08 , Cost  --->  3
x=1/2 , x= -1/2

<<< Answers>>>
x=2 , x= -2 , x=1/2 , x= -1/2

Total Cost = 16
```

◎相反方程式として解いた場合

```
Rule No. con12 , Cost  --->  5
divide equation by x^2
4*(x+x^ -1)^2-25=0

Rule No.  subst0  ( sub2 )  , Cost --->   3
substitute unknown4 for x+x^ -1
4*unknown4^2-25=0

Rule No. iso02 , Cost  --->  1
4*unknown4^2=25

Rule No. iso07 , Cost  --->  1
unknown4^2=25/4

Rule No. iso08 , Cost  --->  3
unknown4=5/2 , unknown4= -5/2

replace( sub2 )
substitute x+x^ -1 for unknown4
x+x^ -1=5/2

Rule No. con02 , Cost  --->  4
multiply equation by x
x^2+ -5/2*x+1=0

Rule No. for01 , Cost  --->  5
x=2 , x=1/2

replace( sub2 )
substitute x+x^ -1 for unknown4
x+x^ -1= -5/2

Rule No. con02 , Cost  --->  4
multiply equation by x
x^2+5/2*x+1=0

Rule No. for01 , Cost  --->  5
x= -1/2 , x= -2

<<< Answers>>>
x=2 , x=1/2 , x= -1/2 , x= -2

Total Cost = 36
```

5．おわりに

この論文では、数式処理システムの制御にメタレベル推論を導入することによる複雑な処理制御の容易化、置換えなどのメタレベル推論の導入、数式処理の推論の過程と問題の難易度を調べることについて述べてきた。

解法規則の分類として公式規則、分離規則などの８種と方程式の型による６種により分類して処理を行なっているが、この枠組の決定方法によっては、処理効率などが大きく異なり、システムの最適処理を実行する構成において重要である。そして、高次の代数方程式を解くために整係数の代数方程式に関する因数分解アルゴリズムなどをより有効に利用する方法の検討も重要である。

また、数式を解く推論過程から得られた難易度の結果を学習し、これより難易度が最小になるよう規則適用順序を制御することも可能である。

Prologによる数式処理において、その容易な拡張性を確認できた。特に、CAI などへの拡張が、今後、有益であると思われる。

《謝辞》 なお、本研究の機会を与えて下さったICOT研究所長 渕一博所長および有益な討論をいただいたICOT第２研究室、第３研究室の各氏、慶応義塾大学の永田守男氏に深謝いたします。

《参考文献》

1)永田: "数式処理におけるProlog",情報処理 Vol.25, No.10 (1984).

2)Bundy,A. and Welham,B.;"Using Meta-level Inference for Selective Application of Multiple Rewrite Rules Sets in Algebraic Manipulation," Artificial Intelligence No.16,pp.189-212 (1981).

3)Borning,A. and Bundy,A.;"Solving Using Matching in Algebraic Equation," Reseach Report 158, Dept. of A.I. Univ. of Edinburgh (1981).

4)Bowen,K. and Kowalski,R.; "Amalgamating Language and Meta-language in Logic Programming,"June(1981).

5)Hiyachi,T. Kunifuji,S. Kitakami,H. Furukawa,K. Takeuchi,A. and Yokota,H.;"A Knowledge Assimilation Method for Logic Databases,"IEEE 1984 International Simposium on Logic Programming,pp.118-125 (1984).