解　説

第五世代コンピュータのソフトウェア†

古川　康一††

1. はじめに

第五世代コンピュータ・プロジェクトは，1990年代の汎用コンピュータの開発を目指して，1982年から研究開発がスタートした．その目標は知識情報処理に向いた並列超高速コンピュータの開発である．この目標に含まれている「**知識情報処理**」と「**並列**」の二つの技術は，両立させることが非常に困難な問題としてよく知られていたが，当プロジェクトでは，その大問題に挑戦したのである．われわれは，プロジェクトをスタートさせる3年前からその計画作りを進めていたが，その段階で**論理プログラミング**が，上に述べた二つの技術の橋わたしを可能にする重要な技術であることが分かってきた．その結果，論理プログラミングを中心理念とするプロジェクト案が作られたわけである．

現在，プロジェクトは10年計画の折返し地点を通過したところである．その間，ほぼ計画どおり研究が進められてきた．なかでも特筆すべき成果は，並列論理型プログラミング言語 **Guarded Horn Clauses**（以下，**GHC** と略す）の開発である[10),24)]．論理プログラミングというと，すぐに Prolog を思い起こすが（事実当プロジェクトにおいて Prolog も重要な役割を演じているが），われわれのプロジェクトでの中心となるプログラミング言語は，いまや GHC である．

GHC は，どちらかというとアーキテクチャ寄りの低水準プログラミング言語であるが，当プロジェクトでは，それと並んで，より水準の高い言語が自然言語理解の研究から生まれた．**Complex Indeterminates Language**（以後 **CIL** と略す）がそれである[18)]．

第五世代コンピュータ・プロジェクトのソフトウェアの研究は，Prolog を土台にしてスタートしたが，現在までに，GHC と CIL の二つのプログラミング言語を生んだ．一般にプログラミング言語は，諸々の概念を純化し，結晶にしたようなものであり，その意味でこれら二つの言語を創り出したことの価値は計り知れない．

第三の特筆すべき成果は，諸々の**プログラム変換技術**である．われわれは現在 GHC と CIL の二つのプログラミング言語を有しているが，それらは，容易に統合しえない．そこで期待されているのが，プログラム変換技術による両言語の結合である．もし CIL で書かれたプログラムをうまく GHC プログラムに変換できれば，両言語の結合がなされたことになる．そのような可能性をも含めて，プログラム変換の諸技術が開発されている．

本稿では，以上の三つの話題を中心にして，第五世代コンピュータのソフトウェア技術の一側面を紹介したい．

2. 並列コンピュータのための核言語

2.1 GHC の誕生

本プロジェクトが開始された当初，論理プログラミング言語で実用化されていたのは Prolog のみであった．ところで，Prolog は静的な世界の記述には大変優れたプログラミング言語であることが知られていたが，動的な世界の記述には大きな問題点をかかえていた．それは，並行処理の基本概念に欠けていたからである．具体的には，入出力機器のような状態をもつオブジェクトの記述が困難で，そのため，オペレーティング・システムが Prolog で記述できない，ということになった．

この点を解決する方法はいくつかあるが，Prolog のデータベース管理機能（assert, retract）を強化，改良する方法がその一つである．それは，**ESP**[1),2)]によって採られた方法である．より抜本的な方法は，以下で述べる並列論理型プログラミング言語の導入である．

Keith Clark らは，Prolog を拡張して **IC-Prolog**

† Software Research of the Fifth Generation Computer Project by Koichi FURUKAWA (Research Center, Institute for New Generation Computer Technology).

†† (財)新世代コンピュータ技術開発機構

と呼ばれる言語を設計した[4]．その拡張の主な点は，並行処理の導入である．たとえば二つのゴールがあるとき，それを Prolog のように左から右へと逐次的に実行するのではなく，二つのゴールを少しずつ交互に実行できるようにした．こうすると，たとえば二つの木の末端が同じシンボルの並びをもっているかどうかを効率良く調べることができる．それは，木を左端から辿りながらシンボルの並びが一致しているかどうかを漸次調べることができるからである．

ところが，IC-Prolog でオペレーティング・システムを書こうと思ってもうまくいかない．たしかに並行プログラミングによって独立したオブジェクトを記述することが可能となったが，入出力機器などを一つのオブジェクトとして扱おうとすると，Prolog の有するバックトラック機能がそれを妨げる．というのは，現実の物理的なオブジェクトはバックトラックできないからである．言い換えると，現実世界はバックトラックしないからである．

その点を考慮して開発したのが **Relational Language** である[3]．Keith Clark と Steve Gregory は，Comunicating Sequential Process や Guarded Command などのアイデアを IC-Prolog に持ち込んで，この言語を設計した．その結果，バックトラック機能のない，より表現力の弱い言語ができたわけである．

その後，Ehud Shapiro は，Relational Language より表現力のある言語 **Concurrent Prolog**（以下 **CP** と略記する）を開発した[22]．その表現力を示す端的な例は，自分自身のインタプリタが簡単に書ける点である．Relational Language は，引数の入出力モードを指定して，その指定に基づいてコンパイルするコンパイラ・ベースの言語であるが，CP は実行時に各変数の出現ごとにデータを書き込んでもよいかどうかを調べるようになっている（そのための余分な情報がプログラマによって与えられる）ので，解釈実行が容易である．

ところで，CP は，その計算実行メカニズムが大変複雑になることが分かってきた．その一番の理由は or 並列 Prolog と同様，**多環境*** を必要とするからである．

Clark と Gregory は，CP の出現に対抗して，Relational Language の表現力を増強した言語 PARLOG を発表した[5]．PARLOG は依然としてモード宣言を必要とするコンパイル・ベースの言語であるが，並列性のほかに逐次性も共存させることによって，その表現力強化を図った．

上田は，CP と PARLOG を詳細に比較検討して，その長所，短所の原因を明らかにし，その結果としてその両者の長所を併せもつ言語 **GHC** を開発した[10),25]．すなわち，GHC は CP と同様インタプリタ・ベースであり，自分自身のインタプリタが容易に書けると同時に，PARLOG と同様実装が容易で，かつ効率の良い言語になっている．さらに，構文的にも，GHC は前2者に比べて最も単純で，すっきりしている．

次節では，GHC についての簡単な紹介をしよう．

2.2 GHC の構文と実行規則

GHC プログラムは，その名前が示すとおり，**ガード付節**の集合で定義される．そして，その一般形は，

$$H :- G1, G2, \cdots, Gm \mid B1, B2, \cdots, Bm.$$

の形をしている．ここで，縦棒 "|" は，**コミットメント・オペレータ**と呼ばれ，節を2分する記号として用いられる．コミットメント・オペレータの左は，頭部Hも含めて**ガード部**と呼ばれ，その右は**ボディ部**と呼ばれる．大雑把にいえば，ガード部はその節が選択されるための条件を示し，ボディ部は，その節による実際の計算を示す．

GHC の実行規則は，つぎの二つの**実行中断規則**で与えられる．

（a） 節のガード部は，その節の呼出し元のゴールを具体化 (instantiate) してはいけない．

（b） 節のボディ部は，その節が選択されるまでは，ガード部を具体化してはいけない．

ルール（a）は同期の方法を定め，ルール（b）はボディの実行方法を定める．ルール（a）の条件は，ガード部でゴールに影響を及ぼすような計算をしてはいけないことを表明している．もし，計算を続けると呼出し元のゴールが具体化されてしまうとき，その計算は中断される（このため，多環境を必要としない）．この中断によって，**同期**がとられるわけである．中断が解かれるのは，呼出し元のゴールと and 関係にある他のゴールの計算が進んで，それによって呼出し元のゴールがより具体化されたときである．

たとえば，二つの待行列のあるサービス窓口について考えてみよう．サービス窓口は一つであるので，二つの待行列は最終的に一本化したい．この一本化を行う手続き merge を GHC で記述してみよう．ここ

* 環境とは，変数とその束縛された値との対の集合である．or 並列 Prolog では，各 or 分岐で同じ変数が異なる値を取りうるので，各分岐ごとに異なる環境（多環境）が必要となる．

で述語 "merge (Xs, Ys, Zs)" は，"待行列 Xs と Ys を一本化したものが Zs である" ことを表すものとする．

```
(m1)  merge ([X|Xs], Ys, Zs) :- true |
            Zs=[X|Us], merge (Xs, Ys, Us).
(m2)  merge (Xs, [Y|Ys], Zs) :- true |
            Zs=[Y|Us], merge (Xs, Ys, Us).
(m3)  merge ([ ], Ys, Zs) :- true | Zs=Ys.
(m4)  merge (Xs, [ ], Zs) :- true | Zs=Xs.
```

節(m1)の第1引数は [X|Xs] となっているが，これは，人が第1の待行列に到着するのを待っていることを表している．(m2)も同様である．(m3)は，第1の待行列がなくなった場合（もうこれ以上人がこない場合）であり，(m4)も同様である．merge プログラムを用いるつぎのような単純な例を考えよう．

```
? - queue 1 (As), queue 2 (Bs),
    merge (As, Bs, Cs), serve (Cs).
```

すなわち，二つのゴール queue 1 (As), queue 2 (Bs) がそれぞれ待行列 As および Bs を作り，その二つを一本化して Cs を作り，Cs に対してサービスを行う問題を考える．

いま，ゴール queue 1, queue 2 の両者とも人の並びを作り出していない時点を考える．すると As, Bs がともに変数のままである．そのとき，(m1)の呼出しを考えると，そこで**同一化 (unification)** As=[X|Xs] が必要となるが，この同一化は，変数 As を具体化しようとするので，この時点で上のルール(a)により実行が中断される．この中断を，もう少し詳しく調べてみよう．節(m1)の merge の第1引数 [X|Xs] は，呼出しゴールの第1引数が少なくとも要素を一つ以上持ったリストでなければならないことを示している．ところが，変数 As は，それが queue 1 によって具体化されるまで，果たして要素が一つ以上あるのか，あるいは空なのか，分からない．そのため，(m1)のガード条件が判定できないわけである．すなわち，(m1)が選ばれるか(m3)が選ばれるかが決まらないことになる．同様に，Bs が具体化されないかぎり(m2)が選ばれるか(m4)が選ばれるか決まらない．

ここで，queue 1 ゴールが As を値 [john|Rest] に具体化したとする．すると，(m1)の呼出しに伴う同一化 As=[X|Xs] は，[john|Rest]=[X|Xs] となり，これによって X=john, Xs=Rest となるが，この同一化は As=[john|Rest] をさらに具体化することはないので，中断は起こらない．そのため，(m1)のガード部の条件は満足され，節(m1)が選択可能となる．節(m1)が選ばれると，そのボディで，同一化 Zs=[X|Us] が実行され，Cs=Zs から，Cs=[X|Us] となる．すなわち，待行列 Cs の第一要素が X に決まったことになる．節(m1)のボディには，もう一つのゴール merge (Xs, Ys, Us) があるが，このゴールを実行すると，再び queue 1 と queue 2 を一本化する動作が開始される．

もし，両方の待行列にともに人が並んだ場合，節(m1)および節(m2)の両方のガード部が満足される．このとき，いずれかの節が非決定的に選択される（処理系の都合によって片方に決まってしまうこともありうる）．

以上で，GHC の構文および実行規則の説明を終えるが，ここで指摘したいのは，実行の中断がルール(a)のみによって決められ，同期を取るためにコミットメント・オペレータ以外の余分な記述を必要としない点である．このことにより，GHC は構文的にも意味的にもすっきりした分かりやすい言語となっている．

2.3 GHC によるプログラミング

GHC のプログラミングでは，**プロセス**が中心的な役割を果たす．プロセスは，動的なオブジェクトと考えてよい．GHC でゴールを実行して節が呼び出されるとき，一つのプロセスが生まれたことになる．たとえば，前節の例では，queue 1, queue 2, merge, serve の4つのプロセスが作られる．merge プロセスは，それが実行されると，(m1)あるいは(m2)が呼ばれている間は，それらのボディ部で再び merge プロセスが作られるので，結果としてそのプロセスは生き続ける．もし(m3)あるいは(m4)が呼ばれると，その実行が終了するとともに，merge プロセスは消滅する．

プロセスの**生成・消滅機能**を利用すると，構造が動的に変化する柔軟な組立てラインを作ることができる．例として，リストの重複を除去するプログラム compact を考えよう．述語 "compact (Xs, Ys)" を，"リスト Xs の重複を除去したものがリスト Ys である" と定義する．compact の GHC プログラムは以下のとおりである．

```
(c1)  compact ([ ], Ys) :- true | Ys=[ ].
(c2)  compact ([X|Xs], Ys) :- true |
                Ys=[X|Ys1],
                remove (X, Xs, Xs1),
                compact (Xs1, Ys1).
```

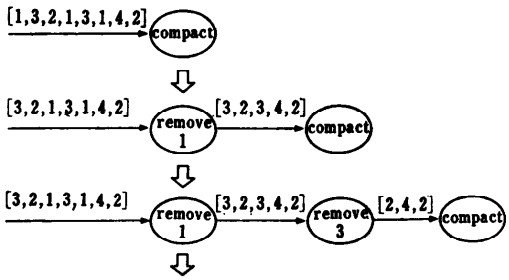

図-1 compact プログラムの実行にともなうプロセス構造の増殖過程

```
(r 1)  remove (X, [ ], Us) :- true | Us=[ ],
(r 2)  remove (X, [X|Xs], Us) :- true |
                remove (X, Xs, Us).
(r 3)  remove (X, [X1|Xs], Us) :- X≠X1 |
                Us=[X1|Vs],
                remove (X, Xs, Vs).
```

いま，ゴール

```
? - compact ([1, 3, 2, 1, 3, 1, 4, 2], Ys).
```

を与えたとする．このゴールを実行すると，まず一つの compact プロセスが生成される．このゴールは，節(c 2)のガード部を満たすので，そこで，Ys=[1|Ys1] を実行するとともに，remove および compact プロセスを生成する．remove プロセス自身，再帰的に定義されているので，入力データがあるかぎり生き続ける．compact が再帰的に1回呼ばれるたびに，新たな remove プロセスが作られる．その結果，図-1 に示すように，プロセスが増殖していく．すなわち，プロセス構造が動的に変化するわけである．

プロセス生成が容易なことは，並列プログラムにとって，非常に重要である．それは，プロセスが並列実行の単位だからであり，そのため，問題の処理過程に応じて並列性を自由に取り出すことが可能となるからである．

GHC のプログラミングにおけるプロセスの重要性は，上で指摘したとおりであるが，それを逐次プログラムと対応づけてみると，プロセスは，逐次プログラムにおけるデータに相当し，プロセス構造はデータ構造に相当する．順序2進木のプログラムを例にとって考えてみると，Prolog プログラムでは，図-2 に示すように，search あるいは update の引数に，順序2進木を表すデータ構造が現れる．すなわち，search ((Key, Value), T) は，順序2進木 T を探索して，Key に付随する値 Value を見つけ出すことを示している．update も同様である．

ところで，GHC プログラムでは，図-3 に示すように，順序2進木の各節がプロセスとして表現され，それらの間の構造は，通信変数によって作られるプロセス構造によって表現される．すなわち，このプログラムでは図-2 における順序2進木の表現 nt_node (Key, Value, Left, Right) がない代わりに，各プロセス（末尾再帰プログラム）は，内部状態として各節のキーと値をもち，その他，左右の部分木プロセスに対する通信変数 Left と Right をもつ．順序2進木をデータ構造として表現している Prolog プログラムの場合，Left および Right はそのデータ構造を辿るためのポインタと考えられるが，それをプロセス構造として表現している GHC の場合，Left/Right は，プロセス間を結合する通信路となる．プログラムの詳細の説明は省略するが，これら二つのプログラムの最も大きな相違は，update のし方に現れている．すなわち，Prolog プログラムでは，関数性を保つために $\log n$ の手間をかけてルートから update すべき節までのすべての節のコピーを作っているが，GHC プログラムでは，当該節のプロセスを再帰的に一度コールして，その内部状態を変えることによって，update を行っている．

GHC の search プログラムでは，探索木に対する操作は，nt_node (非終端節) あるいは t_node (終端節) プロセスの第1引数である通信変数にコマン

```
search((Key,Value),nt_node(Key,Value,Left,Right)) :- !.
search((Key,Value),nt_node(Key1,Value1,Left,Right)) :-
        Key<Key1,!,search((Key,Value),Left).
search((Key,Value),nt_node(Key1,Value1,Left,Right)) :-
        Key>Key1,!,search((Key,Value),Right).
search((Key,Value),t_node) :- Value=undefined.

update((Key,Value),nt_node(Key,Value1,Left,Right),
        nt_node(Key,Value,Left,Right)) :- !.
update((Key,Value),nt_node(Key1,Value1,Left,Right),
        nt_node(Key1,Value1,Left1,Right)) :-
        Key<Key1,!,update((Key,Value),Left,Left1).
update((Key,Value),nt_node(Key1,Value1,Left,Right),
        nt_node(Key1,Value1,Left,Right1)) :-
        Key>Key1,!,update((Key,Value),Right,Right1).
update((Key,Value),t_node,nt_node(Key,Value,t_node,t_node)).
```

図-2 Prolog による順序2進木プログラム

```
nt_node([],_,_,Left,Right) :- true | Left=[],Right=[].
nt_node([search(Key,Value)|Cs],Key,Value1,Left,Right) :-
        true | Value=Value1,nt_node(Cs,Key,Value1,Left,Right).
nt_node([search(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key<Key1 | Left=[search(Key,Value)|Left1],
        nt_node(Cs,Key1,Value1,Left1,Right).
nt_node([search(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key>Key1 | Right=[search(Key,Value)|Right1],
        nt_node(Cs,Key1,Value1,Left,Right1).
nt_node([update(Key,Value)|Cs],Key,Value1,Left,Right) :-
        true | nt_node(Cs,Key,Value,Left,Right).
nt_node([update(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key<Key1 | Left=[update(Key,Value)|Left1],
        nt_node(Cs,Key1,Value1,Left1,Right).
nt_node([update(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key>Key1 | Right=[update(Key,Value)|Right1],
        nt_node(Cs,Key1,Value1,Left,Right1).

t_node([]) :- true | true.
t_node([search(Key,Value)|Cs]) :- true |
        Value=undefined,t_node(Cs).
t_node([update(Key,Value)|Cs]) :- true |
        nt_node(Cs,Key,Value,Left,Right),
        t_node(Left),t_node(Right).
```

図-3 GHC による順序2進木プログラム

ド列を与えることによって指定される．この意味で，この GHC プログラムは，nt_node あるいは t_node をオブジェクトとするオブジェクト指向プログラミングの形態を取っていると考えることができる．

2.4 FGHC と KL1

これまで，GHC について説明してきたが，第五世代コンピュータ・プロジェクトで実際に採用されている言語は，GHC のサブセットである **Flat GHC**（以下 **FGHC** と略記する）を核とする KL1-c と呼ばれる言語である．FGHC は，ガード部でユーザ定義の述語を許さないように制限された言語である．この制限によって，ガード部が入れ子になることがなくなり，平坦 (flat) になる．それが，この名前の由来である．

KL1-c は，FGHC にシステム・プログラムを記述するためのメタ・コール機能を追加したものである．核言語族 KL1 は，このほか，1) KL1-c とともに用いて並列プロセッサへのゴールの配分を指示する KL1-p，2) {KL1-c, KL1-p} の上位に位置し，一般のユーザにとっての使いやすさを追求した KL1-u，3) {KL1-c, KL1-p} の下位に位置し，実際のハードウェアとのインタフェースとなる KL1-b，の3層から成る．

KL1-u としては，応用領域ごとにそれぞれ適した言語を想定できるが，現時点では，オペレーティング・システムを記述するために，とくにオブジェクト指向性を強調した言語 **A'UM**[26] を開発している．

3. CIL

Complex Indeterminates Language (CIL) は，**状況意味論**☆ に基づく談話理解システムの研究から生まれたプログラミング言語である．**Complex Indeterminate**（**複合不確定項**）は関係句 "x such that p" を表現する（x：p のように書く）ための道具であり，状況意味論の基本要素の一つである．CIL は，この複合不確定項を容易に扱えるようにするために，Prolog に二つの拡張を施したものである．その一つは，**部分項 (Partially Specified Term)** と呼ばれるものであり，他の一つは**制約 (Constraint)** と呼ばれるものである．"x such that p" において，x が部分項に対応し，p が制約に対応する．

部分項を用いると，動的に属性が決まっていくようなデータ構造が容易に扱える．部分項は，{family_name/'Furukawa', age/45} のように，属性名と値の対の集合で表される．いま，二つの部分項 X= {family_name/'Furukawa', age/45} と Y= {first_name/'Koi-

☆ 自然言語のためのモデル論的意味論で，文の意味を真理条件として決めるのではなく，状況の間の関係と考える新しい関係論的意味論．談話理解などの文脈処理のよいモデルとなると期待されている．情報の部分性に着目している点が特徴．Barwise と Perry によって開発された．

chi', age/45, affiliation/ICOT} が等しいとすると，この二つの項の同一化 (unification) が可能となり，X=Y= {family_name/'Furukawa', age/45, first_name/'Koichi', affiliation/ICOT} となる．もし，あらかじめ属性集合が分かっていたら，部分項は，Prolog の通常の項で表現が可能となるが，それが実行時に初めて決まる場合には，新たな仕掛けが必要となる．CIL では，部分項同士の同一化アルゴリズムが用意されている．

制約は，複数の部分項の属性間に成り立つべき関係を述べるのに用いられる．たとえば，構文解析において，主語と述語の人称 (person) は一致しなければならない．すなわち，次のような文法情報をもつ主語Xと述語Yの組合せは許されない．X= {exp/she, case/subject, person/third}, Y= {exp/read, case/verb, tense/present, person/first}．このような制約を CIL では，constr (X ! person=Y ! person) と記述することができる．ここで，X ! person は，部分項Xの属性 person の値を示す記法であり，constr (…) は，…が制約条件であることを示す組込み述語である．

制約機能は，Prolog II[6] や ESP などと同様，変数の具体化を待って制約条件を調べる**遅延評価機構**を用いて実現されている．制約機能は，また，生成-検査型プログラムの有力な効率化手法でもある．通常 Prolog では，バックトラック機能を用いて生成-検査型プログラムを作り上げているが，一つの候補の生成に手間取る場合，そのコストは大変大きくなる．一方，もし候補解の一部分のみを生成した時点で検査が可能となる場合，もしそれが不合格ならば，それ以降の生成の手間は省くことができる．それは，制約条件を検査と考えると，検査は，候補解の一部分が生成されて条件が判定できるようになった途端，実行可能となるからである．このように，制約機能は，Prolog の探索機能と組み合わさって，問題解決のための強力な道具立てになっている．

部分項と制約を併せもつことによって，知識表現の分野でよく知られているフレーム，スクリプトなどの機能の大部分を容易に実現することができる．すなわち，CIL 自身，汎用の知識表現言語，あるいは知識プログラミング言語と考えることができるであろう．

4. プログラム変換技術

プログラム変換は，これまでも，関数型言語を対象として，多くの研究がなされてきたが，本プロジェクトでは，それらの研究を論理型言語の上に展開し，発展させてきた．ここでは，その中から，Prolog プログラムの部分計算，同じく Prolog で書かれた生成-検査型プログラムの変換，および FGHC プログラムの変換について述べる．

4.1 Prolog プログラムの部分計算

部分計算は，入力データの一部分が与えられたときに，その情報だけで計算できるところを計算してしまって，残りのデータが来ないと処理できない部分を新たなプログラムとして出力する方法である．すなわち，初めに与えられたプログラムを与えられた入力データに対して特殊化したものが結果として得られる．

部分計算は，言語処理系への応用が大変興味深い．それは，1971年に Futamura によって理論的な考察がなされている[12]．その考えを Prolog 上で実現する研究が竹内らによってなされた[23]．Prolog におけるプログラミング・スタイルの一つに**メタ・プログラミング**と呼ばれるものがある．それは，言語のインタプリタを実現する有効な技術であり，純 Prolog のインタプリタは，わずか3行で記述が可能である．それはつぎのようなプログラムである．

```
(s 1)  solve (true).
(s 2)  solve ((A, B)) :- solve (A), solve (B).
(s 3)  solve (A) :- clause (A, B), solve (B).
```

このプログラムは，よく知られているが，その動きの概略を以下に説明しよう．solve (X) は，ゴール(列) Xを解釈実行することを意味する．節(s 1)は，停止条件である．すなわち，ゴールが "true" になれば，それは解けたことになるので，そのまま成功する．節(s 2)は，二つ以上のゴールの and 結合を解くルールである．ここで，(A, B) は，先頭がAで残りがBであるようなゴール列を表す．そのようなゴール列が与えられたとき，ゴールAとゴール(列) Bをともに解釈実行し，それらがともに成功すればよい．節(2)の右辺がそれを表している．節(3)は，単一ゴールのためのルールである．単一ゴールAが与えられたら，組込み述語 clause (A, B) によって，Aを解く可能性のある節 "A :- B". を取り出して，つぎにその節のゴール(列) Bを実行する（このとき，同一化が行われるが，ここでは省略した）．**部分計算器 (partial evaluator)** は，上記インタプリタを拡張して得られる．それはつぎのようなプログラムである．

```
(p 1)  psolve (true, true).
```

```
(p2)  psolve ((A, B), (RA, RB)) :-
          psolve (A, RA),
          psolve (B, RB).
(p3)  psolve (A, R) :- clause (A, B),
                       psolve (B, R).
(p4)  psolve (A, A) :- residual (A).
(pa)  psolveAll (A, NewCls :-
          bagof ((A :- R), psolve (A, R),
          NewCls).
```

節(pa)が，部分計算の主プログラムで，ゴールAを与えると，それを部分計算し，計算し切れなかった残りを節集合の形で求める (NewCls). その節集合は "A :- R." の形の節の集合であり，それは bagof 以下で求められる．この bagof 以下の意味は，集合表現を用いると

$$NewCls = \{A :- R \mid psolve\ (A, R)\}$$

である (NewCls の具体表現はリストである)．さて，ここで，上記のRを求めるプログラムが psolve (A, R) である．psolve の定義を見れば分かるように，このプログラムは，Prolog インタプリタ solve と大変よく似ている．節(p1)，(p2)，(p3)はそれぞれ節(s1)，(s2)，(s3)に対応している．それらの違いは，psolve では，計算し残したゴール(列)を第2引数で求めている点である．節(p4)は，実際に計算し残したゴールを生成する節である．ゴール residual (A) は，ゴールAが計算できないことを示す述語である．residual の定義は，ここでは与えていないが，たとえば，append ゴールで，第1および第3引数が未知変数であればそれ以上展開できない，といった事柄が定義されていると思えばよい．そして，それらの定義は，プログラムを解析することによって自動的に導き出せることが知られている[9]．

部分計算の応用例としては，確信度つきの Prolog インタプリタ，多世界表現を扱える Prolog インタプリタなどを対象としたものが知られている[17),23)]．また，ボトム・アップ型構文解析プログラムの例も報告されている．これらのプログラムは，すべて Prolog の上で他の言語を定義するインタプリタになっている．そのインタプリタを，与えられた特定のプログラムに対して部分計算し，特殊化することによって，解釈実行の高速化が図れる．見方を変えれば，この過程は，与えられたプログラムをコンパイルすることに相当している．このコンパイルによって，効率が3〜100倍向上したことが報告されている[17),23)]．さらに，部分計算プログラム自身を与えられたインタプリタに対して特殊化して，そのインタプリタに対応するコンパイラを作る試みも成功している[8]．

4.2 生成-検査型プログラムの変換

生成-検査型プログラムは，Prolog プログラムで容易に書くことができる．生成部は，解の候補をつぎつぎと生成し，検査部でそれを検査するわけであるが，バックトラックの仕組みを利用してこのようなプログラムを簡単に作ることができる．もし検査部での検査に合格しなかったら，そこで「失敗」が起こるが，この「失敗」が生成部にバックトラックにより伝えられ，生成部はつぎの候補解を生成することになる．

ところで，生成-検査型プログラムは，効率がよくないことが多い．その原因の一つは，生成部と検査部が離れていることによる．生成部が解候補の一部分を作った時点で，検査部が不合格であると判定できるような場合でも，この分離のために，解候補を完成させることが必要となる．そして，極端な場合には，生成部は無限ループに陥ってしまうこともある．

つぎの例は，そのような事態が起こる例である．問題は，与えられた地図上で，地点Xから地点 goal に至るサイクルのない経路を求める問題である．この問題は，Xから goal に至る任意の経路 Path を生成する生成部 path (X, Path) と，得られた Path がサイクルを含まないことを検査する検査部 good_list (Path) の二つの部分から作られる．そのプログラムを**図-4**に示す．このプログラムにおいて，生成部が無限ループに陥るのは，自明であろう．たとえばゴール path

```
good_path(X,Path) :-
        path(X,Path),good_list(Path).

path(goal,[goal]).
path(N,[N|Path]) :-
        arc(N,Next),path(Next,Path).

good_list([]).
good_list([X|L]) :-
        not_member(X,L),good_list(L).

not_member(X,[]).
not_member(X,[A|L]) :-
        X=/=A,not_member(X,L).

arc(X,Y) :- arc1(X,Y).
arc(X,Y) :- arc1(Y,X).

arc1(a,b).
arc1(a,d).
...
```

図-4 無限ループに陥る経路探索プログラム

(a, Path) の実行を考えると，まず path の第2節が呼び出され N=a, Path=[a|Path1] となった後に arc (a, Next) の実行に移り，Next=b となる．つぎに path (b, Path1) が実行されるが，arc の定義から，arc (b, a) も成り立ち，再び path (a, Path') の形のゴールが現れ，この計算が再び繰り返されることになる．

この無限ループを回避する方法の一つは，3. で述べたように，生成-検査型プログラムを，制約-生成型プログラムにする方法である．すなわち，検査部 good_list を，生成部に対する制約 (constraint) として扱う方法である．

上述の無限ループを避けるもう一つの方法は，**プログラム変換**によって，生成部に検査部を融合させてしまう方法である．これは，プログラムの**展開** (**unfolding**) あるいは**たたみ込み** (**folding**) を応用して達成できる．Seki, Furukawa[21]は，とくに不等号の実行順序を構造的に入れ替える方法を用いて，このプログラム変換を導いた．それは，$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} = \sum_{j=1}^{n}\sum_{i=1}^{j} a_{ij}$ と類似の，実行順序の変更である．この変換によって導いたプログラムを，図-6 に示す．改善されたプログラムでは，解がサイクルをもってはならないことが，新たな地点を辿るたびに調べられているのが分かるであろう．

さて，このプログラム変換を施すと，制約をもったプログラムが，制約のない通常の Prolog プログラムになる．上田[25]は，通常の Prolog プログラムを FGHC に変換する方法を開発したが，その方法とここでのプログラム変換をともに用いると，制約をもった Prolog プログラムを FGHC プログラムへ変換することが可能となる．

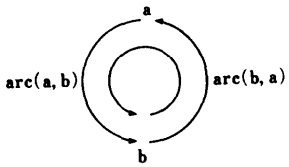

図-5 path プログラムが辿る無限ループ

```
good_path(X,Path) :- good_path1(X,[],Path).

good_path1(goal,Path,[goal]) :-
        not_member(goal,Path).
good_path1(N,History,[N|Path]) :-
        not_member(N,History),arc(N,Next),
        good_path1(Next,[N|History],Path).
```

図-6 変換後の経路探索プログラム (not_member 以下は，元のプログラムと同じ)

4.3 FGHC プログラムの展開

展開は，部分計算やプログラム変換を行うための基本操作の一つである．Prolog の上での展開は，ほとんど自明であり，なんの問題もないが，FGHC ではそう簡単ではない．それは，展開によってゴール間の同期の関係が変化してしまう可能性があるからである．

たとえば，つぎのようなプログラムを考えてみよう．

(c1) firstSecond ([A|I*n*], O) :- true |
O=[A|Out], first (I*n*, Out).

(c2) first ([B|In1], O1) :- true | O1=[B].

firstSecond (I, O) は，入力リスト I の第1および第2要素のリストを出力Oの値とするプログラムで，first (I, O) は，I の第1要素のみから成るリストをOの値とするプログラムである．ここで，(c1) の first を (c2) によって展開すると，つぎのような節 (c1)' を得る．

(c1)' firstSecond ([A, B|I*n*1], O) :- true |
O=[A|Out], Out=[B].

プログラム {(c1), (c2)} とプログラム {(c1)'} は，ともに，入力リストの第1および第2要素を出力するので，等しいように思えるが，第1要素を出力するタイミングまで考えると，それらは異なってくる．すなわち，{(c1), (c2)} では，第1要素が firstSecond の第1引数に具体化されると同時に出力が可能となるが，{(c1)'} では，第1および第2要素が具体化されて初めて，その出力が可能となる．この違いは，見逃すことができない．いま，これらのプログラムのほかに，つぎのような節があったとしよう．

(c3) waitFirst ([X|X*s*], I*n*) :-
true|I*n*=[b|I*n*1].

この節は，第1引数の第1要素が具体化されるのを待って，第2引数の第1要素をbとする．そして，つぎのようなゴールを考えよう．

(G) ?:- firstSecond ([a|I*n*], O),
waitFirst (O, I*n*).

firstSecond の中の I*n* は，waitFirst が実行されて初めて具体化されるが，waitFirst が実行されるためには，変数Oの第1要素が具体化されるのを待たなければならない．ところで，プログラム {(c1), (c2), (c3), (G)} では，変数Oは，firstSecond

ゴールの実行でただちに [a|Out] が具体化されるが，プログラム {(c1)′, (c3), (G)} では，そのためには，In の第1要素の具体化まで，待たなければならない．結局，(In の具体化)←(Oの具体化)←(In の具体化) のような因果律となり，ここで**デッドロック**が生じてしまう．

このデッドロックを回避する展開規則が Furukawa ら[11]によって開発された．その基本的なアイデアは，ガード部を作りかえるような展開の場合には，出力変数への同一化ゴールをあらかじめ分離しておくことである．出力変数のみが，展開する節を呼び出すゴールと and 関係にある他のゴールとの同期に関与しているので，出力変数に対する因果関係さえ保存すれば，全体の因果関係は保たれるわけである．そして，この出力変数への同一化ゴールの分離は，展開の逆操作である**たたみ込み**を用いて達成された．

現在，この展開規則を基にした FGHC の部分計算プログラムを開発中である．さらに，展開規則の改良，展開規則の正当性を議論するための FGHC の意味論などの関連研究も進められている．

5. おわりに

本稿では，GHC および CIL の二つのプログラミング言語を中心に，第五世代コンピュータ・プロジェクトにおけるソフトウェア技術について概説した．ここで述べた研究活動は，本プロジェクトにおけるソフトウェア研究を網羅していない．詳細な紹介ができなかったテーマのうち，特に重要なものを三つあげて，その概要を紹介しよう．

第1は，**CAL** と呼ばれる**制約論理プログラミング言語**である[19]．CIL の項で述べたように，「制約」は強力なプログラミング・パラダイムを与えるが，CAL は**任意の多項式**を制約として記述できる言語である．現在，制約論理プログラミングは，世界的な流行のきざしを見せている[6),13)]が，その多くは，1次式のみを扱うものであるのに対して，CAL は高次の多項式の扱いが可能である．その欠点は，不等式が扱えない点である．

この種の代数を扱える制約論理プログラミングは，論理プログラミングの応用範囲をオペレーションズ・リサーチの分野にまで拡大する点で，興味深い．

第2は，**Computer Aided Proof (CAP)** と呼ばれるシステムの研究開発である[20]．このシステムは，数学の証明記述を支援するものであり，その証明が正しいかどうかを調べてくれる．本システムは，証明を記述するための言語 (**Proof Description Language, PDL**) で書かれた証明を理解し，その各ステップの論理的な正しさを検証する．現在，主として行列に関する問題の証明支援が可能になっている．本システムは，また，仕様からプログラムを抽出するシステムへの拡張も計画されている．

第3は，**Argus/V** と呼ばれるプログラムの検証システムである[14]．Argus/V は，Lisp プログラムの検証を行う **Boyer-Moore Theorem Prover (BMTP)** を手本にして開発されたもので，対象言語は Prolog である．対象言語が関数型言語から論理型言語へ変わったことにより，その検証技術も，論理に基づくものになっている．その変更にともない，BMTP よりも優れた点がいくつか得られた．その第1は，証明したい関係式の中に，存在限定子がある現れ方の範囲内で許される点である．これにより，証明できる性質の範囲が広がったことになる．第2は，証明手続きとプログラムの実行手続きがほぼ同じ枠組になっているので，システムが簡潔になった点である．第3は，帰納法の戦略がより単純になった点である．BMTP では**一般化**と**相互交配**と呼ばれている二つの発見的戦略規則が，Argus/V では**単純化**と呼ばれる規則で統一された．

Argus/V のほかに，関連してプログラムの変換・合成[16]，プログラムの解析[15]，プログラムの修正を行う各サブ・システムが開発されている．

最後に，ソフトウェア研究の今後の展開について，述べてみたい．1. でも述べたとおり，われわれのプロジェクトは，並列論理型言語 GHC と制約論理型言語 CIL を生んだ．そして，その間のギャップの一部は，4.2 で述べたプログラム変換法および上田の方法を用いて解消可能であることも述べた．しかしながら，その方法は万能ではない．とくに，連立方程式を解くような，より強化された制約（**能動的制約**と呼ばれている）は，その方法では扱えない．また，GHC は，データベースの探索問題の記述に向いていないことが，最近明らかにされてきた．このように，制約と探索の両機能を，いかに GHC に代表される並列論理型言語に取り込むか，が大変重要な問題として認識されてきつつある．この問題は，論理を基にした，より表現力のある知識表現言語の設計問題と見なすこともできる．

今後のソフトウェア研究で見逃せないもう一つの

テーマは，並列プログラミング技術の開発である．この分野は，未踏の分野であるといえるであろう．それは，汎用の並列言語 GHC をいかに使いこなすか，という問題だけに止まらない．実際の並列コンピュータの上で，最大限の並列性をどうやって引き出し，実質的なスピードの向上をどうやって達成するか，という問題である．

本プロジェクトの残りは，あと4年であるが，これらの問題が，これからの課題であろう．

参 考 文 献

1) Chikayama, T.: ESP Reference Manual, ICOT Technical Report TR-044 (1984).
2) Chikayama, T: Unique Features of ESP, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, pp. 292-306 (1984).
3) Clark, K. L. and Gregory, S.: A Relational Language for Parallel Programming, Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, ACM, pp. 171-178 (1981).
4) Clark, K. L., McCabe, F. and Gregory, S.: IC-Prolog Language Features, in Logic Programming (K. L. Clark and S.-A. Tarnlund eds.), Academic Press, pp. 253-266 (1982).
5) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London (1984).
6) Colmerauer, A.: Theoretical Model of Prolog II, In Logic Programming and Its Applications, Michel Van Caneghem and David H. D. Warren (eds.), Ablex Publishing Corp., pp. 3-31 (1986).
7) Colmerauer, A.: Opening the Prolog III Universe, Byte, pp. 177-195 (Aug. 1987).
8) Fujita, H. and Furukawa, K.: A Self-applicable Partial Evaluator and Its Use Incremental Compilation, to appear in New Generation Computing.
9) 藤田，古川：Prolog プログラムの部分計算の自動化，ICOT Technical Memo TM-25 (1987).
10) 古川，溝口(編)：並列論理型言語 GHC とその応用，知識情報シリーズ，共立出版 (1987).
11) Furukawa, K., Okumura, A. and Murakami, M.: Unfolding Rules for GHC Programs, to appear in New Generation Computing.
12) Futamura, Y.: Partial Evaluation of Computation Process: An Appraoch to a Compiler-Compiler, Systems, Computers, Controls, 2, pp. 721 -748 (1971).
13) Jaffar, J. and Lassez, J-L.: Constraint Logic Programming, IBM Watson RC Internal Memo (1986).
14) Kanamori, T., Fujita, H., Horiuchi, K. and Maeji, M.: Argus/V: A System for Verification of Prolog Programs, 1986 Proceedings FJCC, Dallas, Texas, IEEE Computer Society Press (1986).
15) Kanamori, T. and Horiuchi, K.: Type Inference in Prolog and Its Applications, ICOT Technical Report TR-095 (1984).
16) Kanamori, T. and Horiuchi, K.: Construction of Logic Programs Based on Generalized Unfold/Fold Rules, ICOT Technical Report TR-177 (1986).
17) Levi, G. and Sardu, G.: Partical Evaluation of Metaprograms in a "multiple worlds" Logic Language, to appear in New Generation Computing.
18) Mukai, K. and Yasukawa, H.: Complex Indeterminates in Prolog and Its Application to Discourse Models, New Generation Computing, 3, pp. 441-466 (1985).
19) 坂井，相場：CAL：制約論理プログラミングの理論と実例，電子情報通信学会研究資料，SS 87-28 (1987).
20) Sakai, K.: Toward Mechanization of Mathematics—Proof Checker and Term Rewriting System—, ICOT Technical Report TR-348 (1988).
21) Seki, H. and Furukawa, K.: Notes on Transformation Techniques for Generate and Test Logic Programs, Proc. 1987 Symposium on Logic Programming, IEEE Computer Society Press (1987).
22) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, Tech. Report TR-003, ICOT (1983).
23) Takeuchi, A. and Furukawa, K.: Partial Evaluation of Prolog Programs and Its Application to Meta Programming, Proc. IFIP '86 (1986).
24) Ueda, K.: Guarded Horn Clauses, Logic Programming '85 (E. Wada ed.), Lecture Notes in Computer Science, 221, Springer-Verlag (1986).
25) Ueda, K.: Making Exhaustive Search Programs Deterministic, Proc. Third Int. Conf. on Logic Programming, Springer-Verlag (1986).
26) 吉田，近山：A'UM—並列オブジェクト指向言語—，情報処理学会研究会資料プログラミング言語，14-4 (1987).

(昭和63年2月25日受付)