ICOT Technical Memorandom: TM-0970

TM-0970

制約プログラミングについて
— 制約ロジック・プログラミング を中心として—

相場　亮、古川　康一

November, 1990

© 1990, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

制約プログラミングについて
制約ロジック・プログラミングを中心として –

相場 亮、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

近年、「制約プログラミング(Constraint Programming)」が注目を集めている。ロジック・プログラミングの会議の多くに「制約(Constraint)」というセッションが作られている。この理由は、「制約」に基づく問題解決に多くの関心が集まってきていることに他ならないと思うが、本稿においては、この「制約プログラミング」について、制約プログラミングの中でも特に最近研究が盛んになりつつある「制約ロジック・プログラミング(Constraint Logic Programming)」を中心に紹介する。

なお、本稿を超える内容については、制約プログラミングについては、Wm Leler の著作 [Lel-88] に、また制約ロジック・プログラミングについては、Jacques Cohen の稿 [Coh-90] に詳しい。

「制約」は、知識処理をはじめとする、いわゆる人工知能の分野にとって、非常に広い役割を果たす重要なパラダイムである。その特徴は、「宣言的である」という点であり、利用者は、解くべき目標を設定することのみが要求され、その目標に到達する、解法については、原則的には触れることなしに、問題を解くことが出来る。

計算機を用いて問題を解く場合、たとえばソートの問題のように、その問題を解く手法が充分に解析され、標準的で効率的なアルゴリズムが得られている場合は別として、問題の領域と問題を構成する要素とを設定し、その要素間の関係を明確化することが必要である。通常のプログラミング言語を用いる場合、さらに問題を解く手法を発見しなければならない。すなわち、問題の解決は、

1. 問題の解析
2. アルゴリズムの発見

の2つのフェーズが必要である。

そして、この発見されたアルゴリズムを、プログラムの形で記述するというのが、従来の問題解決の手法である。

これに対して、「制約」とは、問題領域における問題の構成要素間の関係を宣言的に記述したものである。したがって、「制約」による問題解決は、次の2つのフェーズによって行われる。

1. 問題の解析と制約による記述
2. 制約充足による問題解決

これらのうち、2.の「制約充足による問題解決」は基本的にはシステムが行うが、1.「問題の解析と制約による記述」についてはユーザの責任である。問題の解析については、問題をその構成要素およびそれらの間の関係という形で捉えるということであるから良いとして、そのようにして捉えた問題をどのように記述するか、すなわちどのような言語を用いて問題を記述するかということが次の課題となる。

上にもあるように、制約とは「問題を構成する対象間の関係」という、非常に一般的なものである。これをどのように表現すればよいのか。この関係が、たとえば「整数の大小関係」といった具合

いに固定されているものであれば事は容易で、整数を表す定数と大小関係を表す記号を組み込みで用意しておけば記述することは可能である。しかし、一般的にはそうではない。

ここに、関係を記述することで問題を記述する言語が一つある。ロジックプログラミング言語である。ロジックプログラミング言語においては、すべては述語という形の関係で記述され、しかもこの述語によって表される関係はユーザが自由に定義できる。すなわち、制約プログラミングはロジックプログラミングと結びついて「制約ロジックプログラミング」となって、十二分な記述能力を得ることが出来るのである。

制約ロジック・プログラミングでは、「制約充足による問題解決」は、言語処理系に中に組み込まれた「制約充足器」によって行なわれる。それぞれ異なる種類の「制約充足器」を持った各種の制約言語が提案されている。

以下、**2.**では、制約とロジック・プログラミングについての、より詳しい説明を行なう。**3.**では、制約ロジック・プログラミング言語の例を4つ与える。**4.**では、制約ロジック・プログラミング言語についての理論的な考察を行なう。**5.**では、今後の展開について述べる。

2 制約とロジック・プログラミング

ここで、近年特に盛んに研究されているロジック・プログラミングに制約を取り込んだシステムについて概観してみる[Miz-89]。すなわち、制約ロジック・プログラミングである。制約ロジック・プログラミングは、制約に基づく問題解決をロジック・プログラミングと組み合わせることによって、問題の記述能力の向上と柔軟な表現能力とを目指したものである。

前にも書いたように、制約は非常に一般的なものであって、これをアルゴリズミックに扱うことは困難である。しかし、この一般的な「対象間の関係」も、分解して行くと、かなり具体的な、たとえば「数値の大小関係」であるとか、「ある式とある式とが等しい」という関係であるとかをその基礎に持つことが多い。すなわち、トップレベルにおける関係は一般的なものでも、その関係を構成するものは具体的な数値であるとか、リストであるとか、ブール値であるとかに分解することが可能である。たとえば、簡単な例では、3次元空間中での同一位置にあるという関係は、x、y、z座標の一致という、3組の数値の一致という形に分解することが可能である。

このような基本的な領域における制約の解の発見は、アルゴリズミックに行える。したがって、制約ロジック・プログラミング言語においては、このような基本的領域の制約の解法を「制約評価系」という形で言語機能に取り込んでいるのである。

制約充足問題は、次のように形式化することが可能である。すなわち、この問題は、変数の集合 $V = \{v_1, v_2, \ldots\}$、各変数に対応する値域の集合 $D = d_1, d_2, \ldots$、および制約の集合 $C = \{c_1, c_2, \ldots\}$ から定義される[Fox-90]。ただし、各値域の集合 d_i は、有限集合でも無限集合でも良い。また、各 d_i は離散値の集合でも、連続値の集合でも良い。

各制約 $C_j \subseteq d_1 \times d_2 \times \ldots \times d_m$ は m- 組で、m 個の変数への無矛盾な値の割り当てを決定する。制約充足問題とは、これに対して、すべての変数 v_k に対して、すべての制約 c_j を満たすような値を値域 d_i から見つけ出す問題である。

この解を求める方法としては、たとえば次のような方法が考えられる。

1. ある変数をひとつ選ぶ。
2. その変数の値を、その値域からひとつ選ぶ。
3. その値が、それぞれの制約に対して、無矛盾であるかどうかをチェックする。もし矛盾している場合にはバックトラックにより別の値を選び、無矛盾であれば、次の変数について同様の処理を行なう。

これは、明らかに探索問題として制約充足問題を解いていることになる。したがって、変数や値の選択において多くのヒューリスティクスが考慮される余地があり、これを単純に行なってしまうと、

極めて効率の悪いシステムが得られることになる。しかし、このようなバックトラックに基づく探索は、Prolog等のロジック・プログラミング言語の得意とするところである。さらに、無矛盾性の判定をする際に、基本的な制約を解くシステムが存在して、自動的に無矛盾性の判定をすることが出来れば、制約問題への適用性は極めて高いということが出来る。すなわち、これが制約ロジック・プログラミング言語である。

良く知られているように、代表的なロジック・プログラミングであるPrologは1972年にA. Colmerauerによって提唱された。制約ロジック・プログラミングについても、Prolog同様、彼に負うところが大きい。

最初の制約ロジック・プログラミングが何であるかという問いに明確に答えることは難しいが、1984年に発表されたA. ColmerauerのProlog II [Col-84]が、おそらくその最初のものであると言ってさしつかえないと思う。

ロジック・プログラミングの立場から見ると、制約の記述を許すことにより、制約ロジック・プログラミングにおいては、より宣言的な記述が可能となる。すなわち、基本的な制約の解放については、その解決手法を利用者に意識させることなしに、制約を宣言的に記述することが出来るのである。そのため、制約ロジック・プログラミングにおいては、基本的な制約から解を得るシステム、すなわち制約評価系が必要となる。

Prologにおける計算領域がHerbrand空間であることに留意すれば、このことはとりもなおさず、制約ロジック・プログラミングはPrologの計算領域を拡張したということを意味する。たとえば実数上の線形方程式、線形不等式のための制約評価系があれば、これらを制約として扱えることになり、計算領域がこれらを含むように拡張されるということになる。

ここで、制約ロジック・プログラミングの基本メカニズムの説明のため、単純化された操作モデルを与える[Yok-89]。

1. 拡張ホーン節

   $P :- P_1, P_2, \ldots, P_n; C_1, C_2, \ldots, C_m$ を拡張ホーン節と呼ぶ。ここで、$P_1, P_2, \ldots, P_n$ をリテラル部、$C_1, C_2, \ldots, C_m$ を制約部と呼ぶ。

2. 拡張レゾルベント

   拡張ホーン節に対応する拡張レゾルベントを $R =< RL : RC >$ によって定義する。ここで、RL はリテラルに対する通常のレゾルベントであり、リテラルレゾルベントと呼ぶ。また、RC は制約に対するレゾルベントであり、制約レゾルベントと呼ぶ。リテラルレゾルベントはロジック・プログラミングのレゾルベントと同じで、解かれるべきリテラルの論理積である。一方、制約レゾルベントは解かれるべき制約の集合のある種の標準形である。

3. 拡張評価

   評価のある時点における拡張レゾルベントを $R_n =< L_1, L_2, \ldots, L_m; RC >$ とする。このとき、ある拡張ホーン節 $P :- P_1, P_2, \ldots, P_k; C_1, C_2, \ldots, C_l$ に対して、$\Theta(P) = \Theta(L_1)$ なる代入 Θ が存在するとするとき、新しいレゾルベント R_{n+1} は $R_{n+1} =< \Theta(P_1, P_2, \ldots, P_k, L_2, \ldots, L_m) : RC' >$ によって定義される。ここで、$RC' = simplify(\Theta(C_1 \wedge C_2 \wedge \ldots \wedge C_l \wedge RC))$ である。ただし、$simplify$ は `true`、`false`、`undefined` のいずれかと、簡約化された制約レゾルベントを値としてとる関数で、制約の集合(論理積)の標準形(解)を求めるためのものである。ただし、制約が互いに矛盾する場合には、$RC' =$`false` の場合にはこの評価のパスは fail となる。このような評価によって、リテラルレゾルベントが $\emptyset$ となった時、評価は終了し、その時の制約レゾルベントがリテラル部の解に付け加えられる。

上で述べたスキーマにおける関数 $simplify$ は、制約評価系に対応しているが、一般にこの制約評価系は「遅延評価機構」と「簡約評価機構」とに分けられる。

簡約とは、たとえば連立方程式における消去法や、数値の同値性のチェックのような、制約式を利用者にとってより簡単な表現へと変換する処理を言う。

一方、記述可能な制約でありながら、簡約評価機構の能力が原因で、ただちに簡約することが出来ない場合がある。たとえば消去法を簡約評価機構として持つ場合、非線形方程式制約を扱うことは出来ない。このような場合には、制約中のある変数が具体的な値を持つのを待って、簡約評価機構が扱える形式になってから簡約を行なうことが必要となる。これを制御するのが「遅延評価機構」である。

たとえばProlog II や CIL [Muk-90] において見ることの出来る不等号制約や非等号制約は、制約中のすべての変数が具体的な値を持つようになってから、評価される。これを「受動的な制約処理」と呼ぶが、これは制約評価のほとんどが遅延評価機構によるものであるとして位置付けられる。このような受動的制約においては、それぞれの制約は個別に解かれるため、複数の制約にまたがるような関係を扱うことができない。

一方、たとえば同一変数に対して2つ以上の制約が存在する場合、これらを組み合わせて新しい制約とすることができるような能力を簡約評価機構が持っている場合、これを「能動的な制約処理」と言う。

遅延評価機構を必要とするということは、記述出来る制約の範囲と簡約評価機構の扱える範囲との間に差異が存在するということを意味する。しかし、遅延しても最終的に簡約評価機構で扱える形式にならない場合もありえ。この場合、最終的な解は undefined ということになってしまう。このような遅延評価機構を必要とするような制約評価系のことを、「不完全な制約評価系」と呼ぶ。

一方、記述可能な制約を、遅延させることなしにすべて扱うことができるような簡約評価機構があれば、遅延評価機構は必要ではない。このような制約評価系においては、記述可能な制約に対しては、必ず true、あるいは false が得られる。このような制約評価系のことを、「完全な制約評価系」と呼ぶ。

これまで見てきたように、制約ロジック・プログラミングにおいては、制約評価系が重要な役割を果たす。制約評価系において、我々は、以下の条件が極めて重要なものであるということを主張する [Aib-88]。

以下において、「制約の集合の充足可能性の判定」という用語を用いる。これは、その制約集合をすべて満たすような解が存在するか否かの判定という意味で用いる。ここで注意しなければならないことは、一見簡単な問題ですら、充足可能性の判定が出来ない場合があるということである。たとえば整数係数方程式の整数解を求めるような場合、この問題は決定不能であるから、充足可能性の判定は出来ないことになる。

1. 制約の集合の充足可能性の判定が出来ること。

2. 制約の集合が充足可能であれば、解の標準形がただ1通り存在し、それを求めるアルゴリズムが存在すること。

3. このアルゴリズムは「現実的な意味」で実現可能であること[1]。

さらに、このアルゴリズムは「漸増性 (incrementability)」を持っていた方が、制約評価アルゴリズムとしては望ましい。

この「漸増性」については、[Miz-89] に分かり易く述べられている。今、制約 C_1、C_2、C_3 がこの順に得られたとする。C_1 を標準形 S_1 に変換するのに要する計算時間を t_1、C_1, C_2 を標準形 S_2 に変換するのに要する計算時間を t_2、さらに C_1, C_2, C_3 を標準形 S_3 に変換するのに要する計算時間を t_3 とする。このとき、厳密な定義ではないが、漸増性とは、S_1 と C_2 とから S_2 が計算時間 u_1 で、また S_2 と C_3 とから S_3 が計算時間 u_3 で求められ、t_2 と $t_1 + u_2$、および t_3 と $t_2 + u_3$ との間に大き

[1] たとえばアルゴリズムは存在しても、極めて効率が悪いようでは、実際に用いることは出来ない。

な差がないことを言う。より直観的に言い換えると、ある制約の標準形にあらたに制約を付け加えた場合、これまでに得られている標準形については再計算の必要がないことをいう。

3 制約ロジック・プログラミングの例

この節においては、制約ロジック・プログラミング言語の例として、Prolog III [Col-87]、CLP($\mathcal{R}$) [Hei-87]、CHIP [Din-88]、およびCAL [Aib-88] について簡単に述べる。

3.1 Prolog III について

Prolog III [Col-87] は、上にも書いたように、A. Colmerauer によって提唱された、最も初期の制約ロジック・プログラミング言語のひとつである。文献に登場したのは1987年のことであるが、1984年には、ほぼ現状と同じ構想を持っていたようである。また、この言語は、Prolog(いわゆるマルセイユProlog)、Prolog II に引き続く、一連の言語の中で最新のものである。

Prolog III の特徴は、計算領域がProlog II 同様、無限木であるということである。ただひとつの節からなる木を葉と呼ぶ。葉と、その葉に付けられているidentifier とは区別されない。したがって、ブール値や有理数は、木の特殊なものとなる。木を表現するために、変数、定数、演算子が用意してある。したがって、これらからなる項は、木を表現しているわけである。

Prolog III における各節は次の形をしている。

$$t_0 \rightarrow t_1, t_2, \ldots, t_n, S;$$

ここで、$t_0, \ldots, t_n$ は項であり、S は制約系である。むろん $t_1, \ldots, t_n$ が存在しなかったり、S がない(その場合には空の制約系とみなされる)場合もある。制約系は { と } とで囲んで記述する。

Prolog III で扱うことの出来る制約は次の4種類である。

1. 定数の型に関する制限
2. 線形方程式、不等式系
3. ブール値に関する関係式系
4. リストに関する関係式系

これらにおいて、2. の線形方程式、線形不等式については、次のような構文上の制限を設けて、線形のもの以外は記述できないようになっている[2]。

a. 乗算においては、変数が、その2つの被演算子の片方にしか変数は含まれていてはならない。

b. 除算においては、その2番目の被演算子には変数が含まれていてはならない。

また、4. については、リストの接続の際、その最初の引数の長さを陽に与えることが必要であり、これによって効率化を計っている。

制約については、上の制約の種類の内、1. の定数の型に関する制限は単項の関係子によって記述され、その木のラベルを認識することによって扱われる。2. の線形方程式、不等式系は記号的シンプレックス法で扱われ、3. のブール制約は [Coh-90] によれば SL レゾリューションによって扱われる。4. に関しては特に記述がない。

以下にこれらのうち、いくつかのカテゴリについて制約系の例を示す。

次の例は線形方程式、不等式の例である。鳩(p)と鼠(r)とで、合わせて12の頭と34の足を持つとき、それぞれの数を尋ねるような問い合わせをProlog III では次のような書く。

[2]実際の処理系においては、次に述べるCLP($\mathcal{R}$) と同様、非線形制約を記述することが可能で、遅延評価によって扱われる。

$$\{p \geq 0,\ r \geq 0,\ p+r=12,\ 2p+4r=34\}?$$

この場合、Prolog III では乗算の "*" を省略出来る。これに対する答えは次のようになる。

$$\{p=7,\ r=5\}$$

次の例は、長さ 10 のリストに対して、リスト <1, 2, 3> を左から連結した結果とリスト <2, 3, 1> を右から連結した結果が等しいものを求めるための問い合わせである。ここで、・はリストの連結を意味する演算子である。

$$\{z:10,\ <1,\ 2,\ 3> \cdot z = z \cdot <2,\ 3,\ 1>\}?$$

これに対する答えは次のようになる。

$$\{z = <1,\ 2,\ 3,\ 1,\ 2,\ 3,\ 1,\ 2,\ 3,\ 1>\}$$

このように、Prolog III の制約評価系においては、制約の記述に制限を設けて、必ず yes、または no を返すことが出来るようになっている。これを制約評価系のブラックボックス化という。

上で述べたように、各変数が一意に定まる場合だけでなく、次のように複数の値が制約系を満たす場合がある。

$$\{\ 0 \leq x,\ x \geq 3/4,\ x \neq 1/2\}$$

さらに、これらの値が制限を全く持たない場合、すなわち任意の値について制約系が満たされるような場合には、制約評価アルゴリズムによって単純化された系は { } で表わされる、空の制約系となる。

次に示す例は、前菜、メインディッシュ、デザートからなる食事で、それぞれにいくつかの選択枝が与えられており、合計のカロリーがある値を下回るような組み合わせを求めるようなプログラムである。

```
LightMeal(a,m,d) →
        Apptizer(a,i), Main(m,j), Dessert(d,k),
        {i >= 0, j >= 0, k >= 0, i+j+k <= 10};
Main(m,i) → Meat(m,i);
Main(m,i) → Fish(m,i);
Appetizer(radishes,1) →;
Appetizer(salad,6) →;
Meat(beef,5) →;
Meat(pork,7) →;
Fish(sole,2) →;
Fish(tuna,4) →;
Dessert(fruit,2) →;
Dessert(icecream,6) →;
```

Prolog III は、Prolog、Prolog II と続く、いわゆるマルセイユ系の Prolog の流れをくんで、アルファベットの大文字、小文字の区別は意味を持たず、変数はアルファベット 1 文字か、あるいはアルファベット 1 文字を先頭に持つ文字列で表わされる。また、与えられたゴール列を満す解を次々に表示するようになっている。

したがって、問い合わせ

```
LightMeal(a,m,d)?
```

を行なうと、これを満す結果の組み合わせが得られる。

```
{a=radishes, m=beef, d=fruit}
{a=radishes, m=pork, d=fruit}
{a=radishes, m=sole, d=fruit}
⋮
```

次に示す例は、銀行からの借り入れ金の分割返済の計算である。返済は、一定の期間をおいて行なうものとし、その期間に対して 10% の利息が付くものとする。この計算を行なうプログラムは次のようになる。

```
Payment(< >, 0) →;
Payment(<i> · x, c) →
        Payment(x,(1+10/100)c-9);
```

最初の節は、もし借り入れ金が 0 であれば、返済の必要はないということを表わしており、2 番目の節は、1 回の返済金額を i とすると、借り入れ金額 c を返済するのに、c に対する 10% の利息と、残りの金額を返済しなければならないということを述べている。

このとき、次のような問い合わせを行なってみる。

```
Payment(<i, 2i, 3i>,1000)?
```

この問い合わせは、1000 ドルを借り入れ、これを 3 回の分割で返済する場合である。ただし、各回の返済金額については、2 回目の返済では 1 回目の返済金額の 2 倍の金額を、3 回目の返済では 1 回目の返済金額の 3 倍の金額を返済するものとする。

この問い合わせに対する答えは、次の通りである。

```
{i = 207+413/641}
```

3.2 CLP($\mathcal{R}$) について

CLP($\mathcal{R}$) [Hei-87] は、制約ロジック・プログラミング・スキーマ CLP(**X**) のインスタンスとして開発された言語である。CLP($\mathcal{R}$) の研究は、J-L. Lassez, J. Jaffar, M. Maher 等 IBM Thomas J. Watson Research Center のグループと、オーストラリアのモナシュ大学のグループとの共同で行なわれている。

CLP(**X**) の特徴は、この種の言語として多ソート一階論理に基づき、論理的枠組みの中で初めて意味論を与えたものであることと、satisfaction complete、solution compact といった、この言語スキーマに適合するための条件を明らかにした点にある。これらについては、次節で述べる。

CLP($\mathcal{R}$) は、計算領域として実数 (実際には浮動小数点数) を持ち、この上の線形方程式、線形不等式を制約として記述、評価することができる言語である。

Prolog III とは異なり、制約としては、非線形の方程式、あるいは非線形の不等式を記述することが可能で、したがって、演算子としては、浮動小数点数上の単項のマイナス (−)、加算 (+)、減算 (−)、乗算 (*)、除算 (/)、巾乗 (**) がある。

定数、変数および演算子によって構成された式は、次のような二項関係子によって結び付けられ、制約を形成する。$a = b$、$a < b$、$a > b$、$a \leq b$、$a \geq b$

CLP($\mathcal{R}$) の処理系は、次の 3 つの構成要素から成る。

1. 推論エンジン

   プログラム評価の全体の制御と、変数束縛の保守を行なう。

2. インターフェース

算術式を評価し、制約を標準形に変換する。

3. 制約評価系

線形方程式、および線形不等式を解く。また、非線形方程式を線形になるまで遅延させる機能を持つ。

たとえば、次のような簡単なCLP($\mathcal{R}$)のプログラムについて考える。

```
p(S, T) :- S + T = 8.
q(U, V) :- U - V = 3.
```

このプログラムのもとで、次のような問い合わせを評価する。

```
?- p(X, Y), q(X, Y).
```

すると、節の本体中の制約(等式)、およびユニフィケーションの際に得られる等式を合わせて、次の6本の方程式が得られる。

```
X = S          ユニフィケーションにより得られる
Y = T          ユニフィケーションにより得られる
S + T = 8          pの節の制約
X = U          ユニフィケーションにより得られる
Y = U          ユニフィケーションにより得られる
U - V = 3          qの節の制約
```

CLP($\mathcal{R}$)のインタープリタの最もナイーブな実現を考えると、この6本の方程式を制約評価系に送って、連立させて解くという方法が考えられる。しかし、これでは制約評価系の負担があまりに大きくなり、望ましくない。そこで、CLP($\mathcal{R}$)の現在のインタープリタでは、ユニフィケーションにより得られる等式は、Prologと同様、変数束縛によって扱う。したがって、制約評価系に送られる等式は、X + Y = 8とX - Y = 3の2本のみとなる。このような方法で、制約評価系に全てをまかせることによって生じる効率の低下を防いでいる。

CLP($\mathcal{R}$)においては、等式は消去法を用いて解かれる。一方、不等式はシンプレックス法を用いて解かれる。この制約評価アルゴリズムによって、CLP($\mathcal{R}$)の処理系は制約系が矛盾することがなければ解とyesを、矛盾すればnoを表示する。ただし、制約の中に最終的に非線形のままであるようなものが含まれる場合には制約系とmaybeが表示される。

CLP($\mathcal{R}$)における節は、Prologのそれと同様の構文を持っており、制約は本体中の任意の位置に置くことができる。したがって、CLP($\mathcal{R}$)の各節は次の形をしている。

$$t_0 : -t_1, t_2, \ldots, t_n$$

ただし、ここで$t_1, t_2, \ldots, t_n$は、それぞれ項、あるいは制約である。

次に示すのは、CLP($\mathcal{R}$)で記述した複素数の積に関するプログラムである。ただし、複素数はc(実部,虚部)という項によって表現されている。

```
zmul(c(R1,I1), c(R2,I2), c(R3,I3)) :-
        R3 = R1 * R2 - I1 * I2,
        I3 = R1 * I2 + R2 * I1.
```

このプログラムに対して、次のような一連のゴールが評価可能である。

```
?- zmul(c(1,1), c(2,2), Z).
?- zmul(c(1,1), Y, c(0,4)).
```

```
?- zmul(X, c(2,2), c(0,4)).
```

最初のゴールの評価は、直観的にも明らかなように、本体中の等式の右辺が計算され、左辺とユニファイされる。一方、次の2つのゴールを評価するには、連立方程式を解くことが必要である。しかし、いずれの場合もユニークな解が得られ、したがって、ゴールの評価は遅延されることはない。一方、次のようなゴールについて考えてみる。

```
?- zmult(c(X,Y), c(X,Y), c(-3,4)).
```

このゴールは、次のような制約系に展開される。

```
X*Y - Y*Y = -3,
2*X*Y = 4.
```

これらの制約は非線形であるので、その評価は遅延させられる。もし。これらの変数が具体化されるようなことがあると、その評価は再開されることになる。

また、次に示す例は、電気回路の問題をプログラムしたものである。次の節は抵抗における電圧と電流の間の局所的な関係を示す節である。

```
resistor(V, I, R) :- V = I*R.
```

この節を用いて、2つの抵抗を並列、および直列に接続した場合の回路の状態は、次の節によって示される。このとき、上の局所的な性質と、共有変数とを用いて、大域的な性質が表現される。

```
par_circuit(V, I, R1, R2) :-
        I1 + I2 = I,
        resistor(V, I1, R1),
        resistor(V, I2, R2).
ser_circuit(V, I, R1, R2) :-
        V1 + V2 = V,
        resistor(V1, I, R1),
        resistor(V2, I, R2).
```

ここでVは、この回路によって生じる電圧降下である。

このような方法を用いて、さらに複雑な回路を記述することが可能である。

```
par_series(V, I, R1, R2, R3, R4) :-
        V1 + V2 = V,
        par_circuit(V1, I, R1, R2),
        par_circuit(V2, I, R3, R4).
```

この節に対して、

```
?- par_series(50, I, 10, 10, 10, 10).
```

なるゴールを評価すると、解I = 5が得られる。

3.3 CHIPについて

CHIP [Din-88] はECRCにおいてM. Dincbas等によって開発された言語である。CHIPの特徴は、前に紹介した2つの言語と異なり、有限領域をも計算領域に取り込んだ点にある。これを用いて離散的な組み合わせ問題としての制約充足問題を扱うことが出来る。すなわち、CHIPの目標は、たとえばスケジューリングやレイアウトといったようないわゆる広義の制約充足問題に分類出来るような問題を、制約ロジック・プログラミング言語の枠組の中で解こうとする試みであると言うことが出来る。このような問題が一種の探索問題であるということは既に触れたが、これをロジック・プロ

グラミングの持つナイーブな探索制御だけで解くことは効率面での問題が大きすぎる。したがって、CHIP においては、制約を能動的に用いて、探索空間を積極的に狭めるとともに、いくつかの探索制御のための機能を導入している。

CHIP で扱うことの出来る制約は次の 3 種類である。

1. 有理数上の線形方程式、および線形不等式系

2. ブール値に関する関係式系

3. 値域が有限領域に限定された変数を含むような関係式系

CLP($\mathcal{R}$) とは異なり、CHIP において算術制約の値域として有理数を用いた理由は、浮動小数点演算を行なうことに伴なう計算誤差の問題を排除するためであると述べている [Din-88]。

CHIP においては、有理数上の線形方程式、および線形不等式系はシンプレックス法によって解かれる。また、ブール値上の関係式系は、ブーリアン・ユニフィケーション・アルゴリズムによって解かれる。

値域が有限領域に限定された変数を含むような関係式系について以下に述べる。

有限領域上の制約としては、算術的制約、記号的制約、利用者定義制約、および高階制約の 4 種類がある。

1. 算術的制約としては、領域変数上の算術項に関する通常の関係を制約として記述、評価することができる。たとえば X、Y を項とするとき、X>Y、X>=Y、X<=Y、X<Y、X=Y、X~=Y はいずれも算術制約である。

2. 記号的制約の代表的なものは次の 2 つである:

   `element(Nb,List,Var)` var が List の Nb 番目の要素であるときに成立する。

   `alldifferent(List)` リスト List のすべての要素が互いに異なるとき、成立する。

3. 利用者定義制約は、利用者が定義した述語を、下で述べる「無矛盾性の技法」を用いて評価するような制約として定義することが出来る。

4. 高階制約としては、ある式をある評価関数に対して最大化、あるいは最小化するような解を求めるような制約 (`minimize(Goal, Function)`) がある。

さて、有限領域上の制約は、「無矛盾性の技法」を用いて評価される。これは、フォワード・チェッキングとかルック・アヘッドと呼ばれる技法である。たとえば、次のような制約が与えられたとする。

```
R + E + 1 = 10 + T
```

ここで R $\in \{0,1\}$ であり、かつ E,T $\in \{0,2,3,4,5,6,7,8,9\}$ であるとすると、T = 0 および E $\in \{8,9\}$ が得られる。これは、すなわちすでに得られている情報をもとに、探索空間を狭めていることに他ならない。

次に、CHIP のプログラム例をいくつか示す。CHIP の節は CLP($\mathcal{R}$) 同様、Prolog と同様の構文を持っており、制約は本体中の任意の位置に置くことができる。したがって、CHIP の各節は次の形をしている。

$$t_0 : -t_1, t_2, \ldots, t_n$$

ここで $t_1, t_2, \ldots, t_n$ は、それぞれ項または制約である。

CHIP においてはユニフィケーションを行なう際に、通常の (ロビンソンの) 構文的ユニフィケーションとブーリアン・ユニフィケーションの両方を用いる。したがって、システムに対して、どの引数にブーリアン・ユニフィケーションを用いれば良いのかを宣言してやる必要がある。このために declare

を用いる。例として次の and を見てみる。

```
?- declare and(h,h,bool,bool,bool).
and(M,N,X,Y,X&Y).
```

この例では、最初の2つの引数については通常のユニフィケーションを用い、次の3つの引数についてはブーリアン・ユニフィケーションを用いるということが、declare を用いて宣言されている。また、この節では最後の引数が第3引数と第4引数の論理積になっていなければならないということが述べられている。

ブーリアン・ユニフィケーションを行なった場合には、通常のユニフィケーションにおいて構文的な等式が得られ、それを満たすような最汎ユニファイア (mgu) が求められるのと同じように、ブール等式が得られ、制約評価によってその解が求められる。

次に示すのは、xor- ゲートの検証の例である。

```
?- declare eq(bool,bool).
eq(X,X).

?- declare n_switch(bool,bool,bool).
n_switch(Drain,Gate,Source) :-
        eq(Drain&Gate,Gate&Source).

?- declare p_switch(bool,bool,bool).
p_switch(Drain,Gate,Source) :-
        eq(Drain&not(Gate),not(Gate)&Source).

?- declare xor(bool,bool,bool).
xor(A,B,X) :-
        p_switch(1,A,T1),
        n_switch(0,A,T1),
        p_switch(B,A,X),
        n_switch(B,T1,X),
        p_switch(A,B,X),
        n_switch(T1,B,X).
```

この回路の出力を記号的に計算する場合には、各スイッチ毎に与えられたブール等式を解く必要がある。その結果、X と T1 とに、あるブール項が束縛される。各ステップ毎の X と T1 の値を示す。下線で始まる変数はブーリアン・ユニフィケーションのアルゴリズムによって導入された変数である。

```
?- xor(a,b,X).

1) T1 = 1 # a # _A&a
2) T1 = 1 # a
3) X = b # _C&a # a&b
4) X = b # _C&a # a&b
5) X = a # b # _D&a&b
6) X = a # b
X = a # b
```

計算が終了すると、値 X = a # b が得られる。

3.4 CAL について

CAL(*Contrainte Avec Logique*) [Sak-89] は現在 ICOT において開発中の言語であり、ICOT で開発された PSI-machine の上に ESP を用いて実装されている。

現在の CAL においては、次の 4 種類の制約を扱うことが出来る。

1. 複素数上の線形・非線形代数方程式
2. ブール値上の方程式
3. 有限集合 / 有限集合の補集合 (Finite-Cofinite set) 上の制約
4. 有理数上の線形方程式、線形不等式

複素数上の線形・非線形代数方程式[3]の値域は複素数であるが、その実部、虚部、および方程式の係数は、それぞれ有理数である。一方、ブール値上の方程式[4]の値域は真偽値 {0, 1} である。また、有限集合 / 有限集合の補集合上の制約[5]は、集合と集合、あるいは集合と要素との関係を制約として記述することが出来る。有理数上の線形方程式、線形不等式[6]の値域、および係数は有理数である。

代数制約は Buchberger アルゴリズムによって評価され、Gröbner 基底が求められる。一方、ブール制約はこのアルゴリズムを元にして、ブール値に対して適用出来るように、ICOT で開発されたアルゴリズムによって評価され、Boolean Gröbner 基底が求められる。また、集合制約についても、Buchberger アルゴリズムに基づいて ICOT で開発されたアルゴリズムを利用している。線形制約については、Simplex 法を用いているが、現段階では線形の制約しか記述することを許していない。このような制約評価アルゴリズムによって CAL の処理系は制約系が矛盾することがなければ解と yes を、矛盾すれば no を表示する。

CAL においても Prolog III や CLP($\mathcal{R}$) と同様、変数の値を確定するのに充分な制約が無い場合には、変数間の関係が出力される。

CAL における節は Prolog のそれと同様の構文を持っており、制約は本体中の任意の位置に置くことが出来る。したがって、CLP($\mathcal{R}$) と同様、CAL の各節は次の形をしている。

$$t_0 : -t_1, t_2, \ldots t_n$$

ただし、ここで t_0 は項であり、$t_1, \ldots t_n$ は、項あるいは制約である。また、t_i 制約である場合には、それが代数制約、ブール制約、集合制約、線形制約のいずれであるのかを示すために、等式の前に “alg:”、“bool:”、“setgb:”、“smplx:” を置くことによって区別する。たとえば、alg:A=B は、A=B が代数制約であることを表わし、bool:A=B は A=B がブール制約であることを表わす。

以下に示す例は、CAL における非線形方程式制約の処理を利用したものである。

次のような、三角形に関する知識を CAL のプログラムとして表現する。

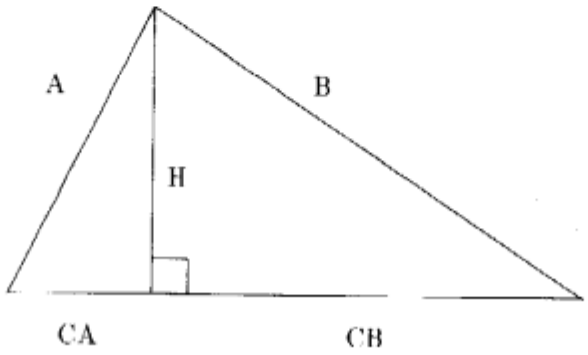
```
surface(Base,Height,Surface) :- alg:2*Surface = Base*Height.

right(X,Y,Z) :- alg:yX^ 2+Y^ 2 = Z^ 2.

tri(A,B,C,Surface) :-
        alg:C = CA + CB,
        right(CA,Height,A),
```

[3]以下、「代数制約」と略す。
[4]以下、「ブール制約」と略す。
[5]以下、「集合制約」と略す
[6]以下、「線形制約」と略す。

```
        right(CB,Height,B),
        surface(C,Height,Surface).
```

ここで、記号 ^ は、べき乗を表わす。このプログラムの各節の意味は次の通りである。

1. 最初の節 surface は、三角形の底辺の長さ (Base) と高さ (Height)、そしてその三角形の面積 (Surface) との間の関係を表わしたもので、いわゆる三角形の面積の公式である。
2. 次の節は直角三角形における、直角をはさむ2辺の長さ (X と Y) と、斜辺の長さ (Z) との関係を表わしたもので、いわゆるピタゴラスの定理である。
3. 最後の節は、「任意の三角形は2つの直角三角形に分割できる」という事実を述べたものである (下図参照)。



このプログラムに対し、次のようなゴールを評価する。

```
?- tri(a,b,c,s).
```

すなわち、すべての引数を未知数として、これらの間の関係を求めさせるのである。

明かに、この処理においては非線形方程式の扱いが不可欠となるが、システムはこの問い合わせに対して、次のような式を出力する。

```
s^ 2 = (-c^ 4+ -1*a^ 4+2*(2*b^ 2*a^ 2)+ -1*b^ 4+2*(2*c^ 2*a^ 2)+2*(2*c^ 2*b^ 2))/16
```

この式は、変数としては a、b、c、および s 以外を含んでいない。ところでこの a、b、c は、それぞれ任意の三角形の3辺の長さであり、s はその三角形の面積であったから、この式は三角形の3辺の長さと、その三角形の面積との関係を示しているものである。すなわち、この式は、ヘロンの公式を展開したものに他ならないのである。

次に、同じプログラムに対して、下のようなゴールを評価させることを考える。

```
?- tri(3,4,5,s).
```

すなわち、3辺の長さがそれぞれ3、4、5であるような三角形 (実際にはこれは直角三角形である) の面積を求めさせようとするものである。この場合には、次のような出力が得られる。

```
s^2=36
```

このような3辺については、この三角形の面積は明かに6なのであるにもかかわらず、ここでは s^2=36 という結果が得られるのは、もともとのヘロンの公式

$$S = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \frac{a+b+c}{2}$$

と、先に得られたその展開形とを比較すれば、すぐ分かる。すなわち、ヘロンの公式においては平方根をとっているので、正の値しかとらないのであるが、先の出力においては、面積の自乗の式になっ

ているため、正の場合と負の場合とを含んでしまう。したがって、代数CALにおいては不等式が扱えないため、s = 6とs = -6とを区別できず、この両者を表現するような解s^ 2 = 36が得られるのである。

次に示すブールCALの例は、電子回路の検証に関する例題である。

今、入力中の1(on)の個数を数え、その個数を3bitの2進数で表わし、これを出力するような5入力、3出力の回路を考える。

次のプログラムは、この回路中の各ゲートに対応する論理記号を用い、適当な中間変数を導入して表現したものである。

```
circuit(X1, X2, X3, X4, X5, Y1, Y2, Y3) :-
    I1 = X1 & X2:bool,          I2 = X1 ∨ X2:bool,          I3 = X3 & X4:bool,
    I4 = X3 ∨ X4:bool,          I5 = I1:bool,               I6 = I2:bool,
    I7 = I3:bool,               I8 = I4:bool,               I9 = I1 ∨ I3:bool,
    I10 = I1 & I3:bool,         I11 = I6 ∨ I8:bool,         I12 = I6 & I8:bool,
    I13 = X5:bool,              I14 = I5 & I2:bool,         I15 = I7 & I4:bool,
    I16 = I14:bool,             I17 = I15:bool,             I18 = I15 ∨ I16:bool,
    I19 = I14 ∨ I17:bool,       I20 = I14 ∨ I15:bool,       I21 = I16 ∨ I17:bool,
    I22 = I9 & I4 & I2 & X5:bool, I23 = I11 & I7 & I5 & I13:bool, I24 = X5 & I18 & I19:bool,
    I25 = I13 & I20 & I21:bool, I26 = I22 ∨ I10:bool,       I27 = I26 ∨ I23 ∨ I2:bool,
    Y1 = I26:bool,              Y2 = I27:bool,              Y3 = I24 ∨ I25:bool.
```

このとき、次のようなゴールの評価を行なう。

```
?- circuit(1,0,1,1,0,y1,y2,y3).
```

すなわち、入力が1、0、1、1、0のときの出力を求めるゴールである。このとき、次が出力される。

```
y1 = 0
y2 = 1
y3 = 1
```

このゴールは信号の流れる方向に合わせて計算を行なわせた例であるが、これとは逆にある出力を満たすような入力を求めることも出来る。

```
?- circuit(x1,x2,x3,x4,x5,1,0,1).
```

これに対しては、次が出力される。

```
x1 = 1
x2 = 1
x3 = 1
x4 = 1
x5 = 1
```

このゴールの場合には、求められる入力は1通りであるが、与えられた出力を得るような入力が複数存在するような場合には、その複数の場合を表現するようなブール式が結果として得られる。

```
?- circuit(x1,x2,x3,x4,x5,0,0,1).
```

この場合には、x1からx5の入力のうち、1つが1で、残りが0であれば良いことになる。このゴールを評価すると、次が得られる。

```
x5 = (((1++x1)++x2)++x3)++x4
x2*x1 = 0
x3*x1 = 0
x3*x2 = 0
x4*x1 = 0
x4*x2 = 0
x4*x3 = 0
```

この結果の中で、++ は排他的論理和[7]を表わし、* は論理積を表わす。したがって、この結果のうち、2本目から7本目までは、x1 から x4 までの、どの2つの入力も同時に 1 になることはないということを表わしている。したがって、もし x1 が 1 であれば、x2、x3、x4 は 0 である。また、このとき、x5 もこれらの結果と排他的論理和の定義から 0 となり、結局 x1 のみが 1 となり、その他の入力は 0 となる。これは、これ以外の入力が 1 であると仮定しても同様であるので、この結果は入力のうち、1 つだけが 1 で、それ以外は 0 であるということを表わしている。

4 CLP(X) の理論

さて、前にも書いたが、CLP(X) は、制約ロジック・プログラミング・スキーマであって、このスキーマに対して、意味論が与えられているという点で重要なものである。この節においては、この結果について簡単に述べる [Coh-90]。

ロジック・プログラミングにおける重要な性質に次の 4 つがある。

1. ロジック・プログラムの正しい処理系は、グラウンド・クエリ Q に対して、Q がプログラム P の論理的帰結であるとき、かつそのときに限り yes と答える。
2. ロジック・プログラムの正しい処理系は、グラウンド・クエリ Q に対して、$\neg Q$ がプログラム P の completion の論理的帰結でありとき、かつそのときに限り no と答える。
3. プログラム P とグラウンド・クエリ Q に対して、Q が P の最小不動点 $T \uparrow \omega$ の要素であるとき、かつそのときに限り Q は P の論理的帰結である。
4. プログラム P とグラウンド・クエリ Q に対して、Q が集合 $T \downarrow \omega$ の要素ではないとき、かつそのときに限り $\neg Q$ は P の completion の論理的帰結である。

Jaffar と Lassez 等は、これらの性質が制約ロジック・プログラミングについても成立するための条件を求めた。すなわち、制約を記述する領域が次の 2 つの性質を満たせば、それを扱うような制約ロジック・プログラミングにおいても、上の性質を満たす。

1. satisfaction-complete

   任意の制約について、充足可能であるか、充足不可能であるかのどちらかが証明可能であること。

2. solution-compact

[7]

x	y	x++y
0	0	0
0	1	1
1	0	1
1	1	0

その領域の任意の要素は、ある制約の集合(無限でも良い)のユニークな解として表現されること。また、ある制約の有限集合の解ではないような要素を解として持つ制約の集合(無限でも良い)が存在すること。

5 これからの展開について

これまで見てきたように、制約ロジック・プログラミングは、「制約」に基づく問題解決に対して、記述の面で大変有用であるが、制約充足の能力には問題がある。それは、処理系に組み込みになっているので、制約評価アルゴリズムを自由に変更したり、問題特有のヒューリスティクスを導入することが困難な点である。たとえば、つぎのようなスケジューリングの問題を考えてみよう[Fox-90]。いま、全体で n 個の作業を考え、それらを A_i とする。また、各 A_i には、「締め切り」 d_i が付随しており、作業 A_i は日程 d_i までに終了していなければならないものとする。また、各作業には優先順位が与えられており、あらかじめ定められた順序で作業は行なわれなければならないものとする。しかも、各作業を実行するのに、m 個の資源のうち、ひとつを使うものとする。このとき、制約充足問題としてこのスケジューリング問題を捉えると、次のように定式化することが出来る。

すなわち、

- 各作業 A_i は、値域 $A_i \in \{< T_i, R_i > | 1 \leq T_i \leq n, 1 \leq R_i \leq m\}$ を持つ。ここで、T_i は作業 A_i に要する時間であり、R_i は選択された資源である。
- $d_i \in \{1, 2, \ldots, n\}$ を締め切り日とする。
- P_{ij} を優先順位を表す行列とする。ただし、$P_{ij} = 1$ の場合には A_i が A_j よりも優先されるものとする。

このとき、各値は以下の制約を満たすように選択される。

- $\forall i[T_i \leq d_i]$、すなわち、各作業はその締め切り日以前に終了していなければならない。
- $\forall ij[(P_{ij} = 1) \supset (T_i < T_j)]$、すなわち、作業 i と作業 j との間に優先関係が存在すれば、T_i は T_j より以前に行なわれなければならない。
- $\forall ij[((i \neq j) \wedge (R_i = R_j)) \supset (T_i \neq T_j)]$、すなわち、同じ時間に複数の作業が同一の資源を利用することはない。

これに加えて、たとえば資源の種類が複数あるなどの条件を付け加えていけば、より問題を複雑化することが可能である。このように、一般に制約充足問題は極めて複雑な問題となり、これを解くためには、相当の手間を要することになる。したがって、このような制約充足問題を解くようなプログラムを書くにあたっては、様々なヒューリスティクスが必要であるし、それでもなお、充分な効率で解くことが出来ない場合もありうる。

すなわち、現在の制約ロジック・プログラミング言語におけるひとつの大きな課題は、このような言語の枠組の中で、ヒューリスティクスをどのように表し、また扱うかということにあるように思われる。

現状においては、制約評価アルゴリズムを比較的熟知している人であれば、制約の呼びだしの順番などのいわばプログラミング技法を用いた多少の工夫が可能ではある。しかし、これでは「宣言性」を掲げている制約ロジック・プログラミング言語としては、少しおかしいのではないだろうか。

たとえば制約評価系に対する様々なコマンドであるとか、あるいはメタ述語を用いた「制約集合」の認識と、それに基づく制御であるとかの機能をうまく取り込むことによって、ある程度のヒューリスティクスが表現可能なのではないか。

さらに、複数の制約評価系にまたがるような問題記述をどのように扱うかもひとつの課題であろう。

数値データは、多くの制約ロジック・プログラミング言語で扱うものであるが、実際的なプログラミングを前提とすると、この扱い方にも問題はある。たとえばCLP($\mathcal{R}$)では浮動小数点を用い、それ以外の上述の言語においては有理数表現を用いる。有理数表現は一般に浮動小数点を用いるよりも効率が低下するが、浮動小数点には誤差の問題が常につきまとう。制約ロジック・プログラミングの場合、特に問題になるのが、数値の同値性が制御に大きな影響を与えるということである。また、もうひとつの大きな問題は、制約評価系内部がユーザに対して開放されていないため、最終的に出力された解の誤差解析がこのままでは非常に難しいという点である。

我々も応用上の必要から、代数制約の評価アルゴリズムであるBuchberger Algorithmについて、従来用いてきた有理数を用いたバージョンに加えて、浮動小数点を用いるバージョンを開発中である。この実装において、計算誤差の問題は大きな課題となっている。

また、我々はICOTにおいて、制約ロジック・プログラミング言語の新しい使い方を模索中である。我々はCALに、ある時点における制約集合をセーブしたりであるとか、ある制約集合をロードしてくるなどといった機能を付加している。ある問題をある様々な特定の事例について解くような場合、その問題を制約ロジック・プログラミング言語を用いて記述し、これを一般的な場合について解いておき、これをセーブしておく。さらに特定事例ごとにこれをロードして、その特定事例を表すような制約を付加して解くことによって、この問題を解くということを考えている。これは、特定事例ごとに全く別に問題を解かせる場合と比較して、効率的に有利になると考えている。

このような考え方は、解として得られた集合を一種のプログラムとみなすという立場であり、これは制約ロジック・プログラミング言語の新しい使い方をもたらすものであると考えている。

もうひとつ制約ロジック・プログラミング言語の今後の展開にとって重要なキーワードは「並列」である。制約ロジック・プログラミング言語と並列とのかかわりあいかたとしては、ひとつは制約評価系の並列化が考えられ、もうひとつは並列制約ロジック・プログラミング言語が考えられる。

制約評価系の並列化の目的は、制約評価の効率向上に他ならない。現在、ICOTにおいては、Buchbergerアルゴリズムの並列化の実験を行なっている。

一方、並列制約ロジック・プログラミング言語の目的は、より柔軟な制御を実現し、さらに並列事象における制約問題解決を目指すことである。これについてもICOTでは並列環境における並列制約評価系の効率的実装を目的にCommitted-Choiceに基づいた並列制約ロジック・プログラミング言語GDCCを試験的に実装中であり、さらにAndorraモデル[Har 88]に基づく並列制約ロジック・プログラミング言語について研究中である。

また、制約ロジック・プログラミング言語の研究で、もうひとつ興味深いテーマとして、制約階層がある[Bor-89]。これまで述べてきたような制約ロジック・プログラミング言語においては、制約は成立するか否かである。制約階層を導入することによって、制約に成立要求の強さを付け加えることが出来る。たとえば、ある制約は必ず成立しなければならないし、別の制約は、出来れば成立してほしいという程度であるとする。このような問題は制約ロジック・プログラミング言語を用いて記述することも可能であるが、その場合には、もし後者の制約が成立しない場合には、この制約は全く無かったものとしてあつかわれることになる。しかし、たとえば前者をある製品を作るための物理的制約、後者をコストの計算式とすると、最初に考えていたコストが守れなければこれを全く無視して良いということはない。たとえばこのような場合には、後者の制約は成立しないまでも、その制約からの「ずれ」を最小に留めたい。このような問題は、階層制約ロジック・プログラミング言語を用いることによって記述可能となる。

いずれにせよ、制約ロジック・プログラミングの研究の歴史はまだ数年しかない。それに先立つ制約充足問題の長い歴史の延長線上に何を付け加えることになるのかは、まだはっきりしていないといえる。すべてはこれから生まれようとしているということが、制約ロジック・プログラミング研究の現状であると考えている。

[謝辞]

制約ロジック・プログラミング研究の機会を与えて頂いたICOT淵所長に感謝する。また、制約ロジック・プログラミングに関する様々な研究、実装、実験にあたり、熱意をもって研究を行っているICOT第4研究室のメンバに感謝する。さらに、常に我々の研究を見守り、有益な助言を頂いている制約プログラミング・ワーキンググループの委員、オブザーバの皆様に謝意を表する。

[参考文献]

[Aib-88] Aiba, A. et al.: Constraint Logic Programming Language CAL. In Proc. of FGCS'88 (1988).

[Bor-89] Borning, A., et al. : Constraint Hierarchies and Logic Programming. Proc. of ICLP89, (1989).

[Coh-90] Cohen, J.: Constraint Logic Programming Languages. Communications of the ACM, Vol.33, No. 7, July 1990, pp.52–68.

[Col-84] Colmerauer, A.: Equations and Inequations on Finite and Infinite Trees. In Proc. of FGCS'84 (1984).

[Col-87] Colmerauer, A. : Introduction to Prolog III. Proc. of the 4th Annual ESPRIT Conference, Brussels (1987).

[Din-88] Dincbas, M. et al. :The Constriant Logic Programming Language CHIP. In Proc. of FGCS'88 (1988).

[Fox-90] Fox, M. S., and N. Sadeh: Why Is Scheduling Difficult? - A CSP Perspective.

[Har-88] Haridi, S., and P. Brand : ANDORRA PROLOG - An Integration of PROLOG and Committed Choice Languages. In Proc. of FGCS'88 (1988).

[Hei-87] Heinze, N. et al. : Constraint Logic Programming – A Reader. 4th IEEE Symposium on Logic Programming, San Fransisco (1987).

[Lel-88] Leler, W.: Constraint Programming Languages – Their Specification and Generation. Addison-Wesley Publishing Co., ISBN 0–201–06243–7 (1988).

[Miz-89] 「制約論理プログラミング」溝口文雄、古川康一、J-L. Lassez 編、淵一博監修、知識情報処理シリーズ 別巻 2、共立出版株式会社 (1989).

[Muk-90] Mukai, K.: A System of Logic Programming for Linguistic Analysis. ICOT TR–540 (1990)

[Sak-89] Sakai, K., A. Aiba : CAL : A Theoretical Background of Constraint Logic Programming and its Applications. J. Symbolic Computation, Vol.8, No.6, December 1989, pp.589–603.

[Yok-89] 横井俊夫、相場亮、「制約ロジック・プログラミング – 知識処理への新しいパラダイム – 」、情報処理、Vol.30、No. 1、Junuary 1989、pp. 29–38.