ICOT Technical Memorandom: TM-1055

TM-1055

制約論理プログラミング入門

相場　亮

May, 1991

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

制約論理プログラミング入門

Introduction to Constraint Logic Programming

相場 亮
(財) 新世代コンピュータ技術開発機構

1 はじめに

Constraint は、「制約」、あるいは分野などによっては「拘束条件」とも訳されるが、歴史的には「制約充足問題 – *Constraint Satisfaction Problem : CSP* –」という文脈で登場した。

「制約による問題解決」とは、探索問題の形式化の一形態であるということができるが、その中心的概念は、「制約」である。では、制約とは何であるのか。

計算機を利用して、ある問題を解決しようとする場合、まずその問題を厳密に記述することが必要である。そのためには、問題領域とその領域における対象、およびこれら対象間の関係を明確にしなければならない。

たとえば与えられたデータの平均値を求めるといったような数値的な問題を解く場合には、この問題の領域は実数の集合であろうし、その領域における対象とは実数である。そして、平均値を求めるという問題は、「データの総和のデータ数による商が平均値と等しい」という、実数間の関係で表される。

また、図形的な問題についていえば、2 次元空間が問題領域であり、そこにある各種図形が対象であり、これら図形の間の位置関係等が対象間の関係ということになる。

制約というのは、問題領域における対象間に成立する、このような関係を宣言的に記述したものである。

次に、「宣言的な記述」ということについて述べる。

ここでいう「宣言的な記述」というのは、問題を構成する対象間に成立する関係 – 制約 – を、それらの関係を満たすような対象の具体値をどのように求めるかということには触れずに述べることを指す。

このような関係のみの表現を許す、あるいはそのような表現が可能であるようなプログラミング言語を「制約プログラミング言語 (*Constraint Programming Language*)」と呼ぶ。

制約は上で述べたように「宣言的」であるから、その特色として、同一の問題表現 (プログラム) によって、様々な質問に対する答えを得ることが出来る。このことは、「宣言的」と対峙する概念である「操作的」な記述を行なう既存のプログラミング言語による問題記述と比較すると、より明らかになる。

Leler [Lel-88] は、制約プログラミング言語と通常の手続き型言語との、次のような比較を行なっている。

たとえば摂氏温度と華氏温度との間の温度変換の問題について考えてみる。与えられた華氏温度 (F) に等しい摂氏温度 (C) を求めるためには、手続き型言語では、代入文を用いて次のように書く。

$$C = (F - 32) * 5/9 \quad (1)$$

また、逆に、与えられた摂氏温度を華氏温度に変換するには、次のような代入文が必要である。

$$F = 32 + (9/5) * C \quad (2)$$

したがって、摂氏温度と華氏温度との間の変換を、どちらが与えられても、もう一方の値を求めることが出来るようにするためには、この両方の代入文が必要となる。さらにこれに絶対温度との変換を組み入れると、次の代入文をさらに付け加えることが必要となる。

$$K = C + 273 \quad (3)$$
$$C = K - 273 \quad (4)$$
$$K = (5/9) * F - 257.4 \quad (5)$$
$$F = (9/5) * K - 561.4 \quad (6)$$

このように新しい対象が増えるたびに、それらの間の変換を行なうために必要な代入文は、指数的に増加する。

一方、制約プログラミング言語においては、これらの対象間の関係を宣言的に記述するのであるから、たとえば摂氏温度と華氏温度との変換であれば、

$$F = 32 + (9/5) * C \quad (7)$$

という関係式を記述することだけが要求される。この = は、上の代入文における = が代入操作を示す演算子であるのに対して、その両辺が等しいことを示す演算子であることに注意してほしい。そのかわりに、この 1 本の関係式から必要な情報を得る、すなわち、既知の値を使って未知の値を求めることは、処理系の責任となる。

このような相違は、通常のプログラミング言語において見ることのできる等式状の文があくまで代入文であり、制約は関係式であるという点に起因している。

このように、通常のプログラミング言語を用いる場合には、問題を解く手法を発見し、それをプログラムの形で記述するのに対して、制約を用いると、問題を構成する要素間の関係のみをプログラムとして記述し、そこから必要な情報を得るという部分は、処理系に任せてしまうのである。

まとめると、従来の手法において、問題の解決は、

1. 問題の解析
2. アルゴリズムの発見
3. アルゴリズムの記述

の3つのフェーズが必要であった。ただし、上の例や、前に述べたような「平均値を求める問題」のような場合には、アルゴリズムの発見といったような大袈裟なことは必要なく、式を変形したり、適当なアルゴリズムを選択すれば良いのであるが、いずれにせよ、プログラムの入力と出力を峻別し、それぞれの場合に応じた解法を特定する必要がある。

一方、「制約」による問題解決は、

1. 問題の解析
2. 問題の記述

の2つのフェーズからなる。この中でも、「問題の解析」が中心となる。すなわち、問題を構成する対象間の関係を明確化し、この関係を制約という形でプログラム中に記述する。したがって、問題を解く手法については、システム側にまかされることになる。

「制約による問題解決」に際しては、このような方法で記述された問題(プログラム)に対して、様々な形態の質問を行なうことが可能である。それらの例を以下にあげる。

ただし、実際の言語については、各言語の機能に従って、かならずしもこれらすべての質問についての回答を与えられるわけではない。

1. 解は存在するか
2. 解を見つけよ
3. すべての解を見つけよ
4. 最適な解を見つけよ
5. ...

また、「制約による問題解決」には、次のような応用領域が考えられており、また実際にいくつかのプログラムが書かれている。

1. 画像解析
2. 設計
3. スケジューリング
4. パズル
5. ...

制約充足問題は、次のように定義することが出来る。すなわち、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_k から見つけ出す問題である。

次に、これをどう解くかが問題になる。一般には、このような問題は次のような探索問題として解かれることになる。

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。

制約プログラミング言語においては、制約という要素技術を内包するプログラミング言語を考え、問題をこのプログラミング言語におけるプログラムとして記述し、解決することを考える。

さて、最初に見た通り、制約とは関係であった。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。論理プログラミング言語である。論理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論理プログラミングと結びついて「制約論理プログラミング」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。

さらに、探索ということについて言えば、論理プログラミング言語は、バックトラックに基づく探索によってそのプログラムを実行しており、これらの意味で、論理プログラミングは、このような制約による問題解決を記述し、解くのに適した性質を持っていることが分かる。

2 制約機能と論理プログラミング

この節においては、論理プログラミングに制約機能を組み込むことが、論理プログラミングにとってどのような意味を持つことになるのかについて考えてみたい。論理プログラミング言語として、ここではPrologについて考えてみる。

Prologの計算機構は項の間の構文的同値性を扱うユニフィケーションに依存している。したがって、「問題の対象領域における対象と、これらの間の関係を宣言的に記述する」ということを考えると、「対象」が項自身であるか、あるいは項を用いてコーディングすることが可能で、かつ扱う関係が「同値性」であれば、そのような「同値性」という関係を制約として扱えることになる[1]。

たとえば、自然数について考えてみる。

自然数を項でコーディングするために、0と後者関数 (*successor function*) とを用いて定義する。すると、

$$\begin{aligned} 0 &\Leftrightarrow 0 \\ s(0) &\Leftrightarrow 1 \\ s(s(0)) &\Leftrightarrow 2 \\ s(s(s(0))) &\Leftrightarrow 3 \\ &\vdots \end{aligned}$$

という対応が得られる。

これで自然数の同値性を、Prologのユニフィケーションを用いて表現することが可能となる。このようなコーディングを基に加算を定義すると、次のようになる。実際、これは制約の概念を利用したものとなっており、この同じプログラムを用いて、自然数上の減算も行なうことが出来る。

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

この述語add/3は、その第1引数と第2引数の和を第3引数に求めるものである。この述語を使って、1+2の計算を行なってみると、次のようにして結果3が得られる。

1. 1+2を計算するための、述語addの呼び出しadd(s(0),s(s(0)),R)を行なう。
2. add(s(0),s(s(0)),R)が、addの2番目の節とユニファイされ、変数の割り当て{0/X,s(s(0))/Y,s(Z)/R}が得られる。
3. この結果、この2番目の節の本体中にある述語add(0,s(s(0)),Z)が呼び出される。
4. add(0,s(s(0)),Z)が、addの1番目の節とユニファイされ、変数の割り当て{s(s(0))/X、Z/X}が得られる。
5. この結果、Zはs(s(0))となる。
6. さらに、この結果、Rはs(s(s(0)))となり、計算は終了する。

このようにして、自然数を項を用いてコード化することにより、加算を通常のPrologのプログラムとして定義することは可能であるが、この方法では、扱う自然数が特に小さい場合を除き、効率は良いとは言えず、実際的な方法とも言えない。

このようなコード化のかわりに、自然数の集合を計算領域とし、自然数間の関係を記述することができれば、このような加算を表現することは可能である。ここで「関係」と書いたのは、本来Prologの述語が項間の関係を記述したものであり、「操作」を記述したのではこの「関係としての述語」の持つ利点を充分に生かせないということを意味している。

たとえばDEC-10 Prologのような商用Prolog処理系には、数値を扱う機能が用意されている。これは数値間の四則であり、計算結果をある変数にユニファイするための機能"is"である。これらを用いて上の加算の問題を解くと、次のようになる。

```
add(X,Y,Z) :- Z is X+Y.
```

ところが、このプログラムにはひとつの問題がある。この"is"は、その右側の式の値を求め、左側の変数にユニファイするという機能を持っている。したがって、右側の式中には数値しか現れてはならない。もし現れればエラーとなる。上の場合にはXとYは数値でなければならない。ところが、addはX、Y、およびZの間の関係を述べたものであるはずなので、YやZを変数にしての呼び出し、たとえば?-add(3,Y,5).のように減算としても利用できなければならないはずである。

少なくとも、エラーとしないためには、上の節は次のように変更しなければならないであろう。

```
add(X,Y,Z) :-
   integer(X), integer(Y),!,Z is X+Y.
```

integerとは、その引数が整数の場合に成功するような述語であるから、isはXとYが数値であるときしか実行されず、エラーとなることはない。しかし、これでは先程の減算は出来ない。これを可能とするためには、XとYは数値であるような場合のための節のみならず、XとZが数値である場合の節

```
add(X,Y,Z) :-
   integer(X), integer(Z),!,Y is Z-X.
```

や、YとZが数値であるような場合のための節

[1] 実際、(純)Prologは一種の制約論理プログラミング言語であるとみなすことが出来る。

```
add(X,Y,Z) :-
    integer(Y), integer(Z),!,X is Z-Y.
```

も必要になる。これでは、上に書いた、手続き型言語の代入文の例と全く同じことになってしまう。この原因は、isという、代入文と酷似した機能を用いていることである。したがって「関係」として表現することが重要になるのである。

このような、たとえば自然数の間の等式を制約として扱うためには、このような関係式を扱えるような機能をPrologに付加すれば良い。これが、制約論理プログラミング言語の基本的な発想である。これを扱うような、制約論理プログラミング言語処理系の一部を「制約評価系 – *Constraint Solver* –」と呼ぶことにする。

制約論理プログラミング言語における制約の取り扱いは、意味的には、Prologにおけるユニフィケーションの一般化という形でとらえることが出来る。すなわち、Prologにおけるユニフィケーションは、エルブラン領域における等式の可解性に関するものであるのと同様、制約の評価は、ある問題領域における制約の可解性に関するものである。すなわち、Prologにおけるユニフィケーション手続きは、エルブラン領域における等式の可解性を決定し、可解であれば最汎ユニファイアを計算するのに対して、制約評価においては、問題領域における制約の可解性を決定し、可解であれば解を含むような標準形を計算する。すなわち、制約論理プログラミング言語における制約評価系は、Prologにおけるユニフィケーション・アルゴリズムの実装部分に対応することになる。

このような意味で、制約論理プログラムの操作的モデルは、ユニフィケーションに基づいた論理プログラムのそれの拡張になっている。

すると、制約論理プログラミング言語において、制約評価系で扱える制約とは、前に述べた一般的な定義と比較して、かなり限られたものとなる。具体的には、制約論理プログラミングにおいて、このような方法で扱われる制約は、次の要件を満たしているべきであると考える。

1. 制約の充足可能性(解を持つかどうか、可解性とも言う)が決定できること。
2. 充足可能性を決定する効率の良いアルゴリズムが存在すること。
3. その計算領域は、制約の領域として導入することが正当化できる性ど、頻繁に用いられること。
4. 充足可能な制約については、ある種の標準形が計算でき、それがその制約の解と同一視できること。

では、前に述べた一般的な制約は、制約論理プログラミング言語においてはどのように扱われるべきものなのであろうか。

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、制約論理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。

最初の部分にも書いたように、一般的な制約とは、探索によって解かれる。したがって、どの変数を最初に選択するか、であるとか、どの値を候補とするか等、様々なヒューリスティクスがありうる。

これらのヒューリスティクスを、制約評価系を使いつつ、制約論理プログラミング言語によるプログラムとして記述するというのが、制約論理プログラミング言語と制約による問題解決との関係であると考える。しかし、この「関係」は、まだ明らかにされているとはいいがたい。

3 制約論理プログラミング言語

論理プログラミングは、1972年にColmerauerによって提唱されたPrologをその嫡嫡とする。それ以降Prologを母体にさまざまな改良や拡張の努力が続けられている。制約論理プログラミングについても、やはりColmerauerに負うところが大きい。Prolog IIが1980年に発表され[Col-84]、Prolog IIIは文献としては1987年に発表された[Col-87][2]。これらの影響を受け、オーストラリア、アメリカ、ヨーロッパ、日本などで制約論理プログラミングに関する研究開発が活発化してきている。

ここで、制約論理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[Yok-89]と、ナイーブなメタ・インタープリタ[Coh-90]について述べる。

単純化された操作モデルについては、「拡張ホーン節」、「拡張レゾルベント」、および拡張評価について述べる。

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

[2] アイデアとしては、1984年頃にはあったようである

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論理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、あるいは 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、$RC' =$ false となってこの評価のパスは 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。

次に、J. Cohen による制約論理プログラミング言語についての解説 [Coh-90] にもあげられている、メタ・インタープリタについて、簡単に述べる。

まず、節 Head :- Body,Constraints を、次のような単位節で表す[3]。

```
clause(Head, Body,Constraints)
```

ただし、Head はリテラルで、Body はリテラルのリストである。

また、単位節は、次によって表す。

```
clause(Head,[ ])
```

このとき、制約論理プログラミング言語のインタープリタは、次のように書くことが出来る。

```
solve([ ],C,C).
solve([Goal|Goals],C_Old,C_New) :-
    solve(Goal,C_Old,C_Inter),
    solve(Goals,C_Inter,C_New).
solve(Goal,C_Old,C_New) :-
    clause(Goal,Body,C_Current),
    constraint_solver(C_Old,C_Current,C_Temp),
    solve(Body,C_Temp,C_New).
```

ここで、述語 solve の 3 つの引数は、それぞれ次のような意味を持っている。

1. ゴール、あるいはゴールのリスト
2. その時点における制約の集合 (の標準形)
3. 新しく得られた制約 (の集合) を、2 に付け加えて得られる新しい制約の集合 (の標準形)。

ここで、述語 constraint_solver が、制約評価系である。

次に、この制約評価系について述べる。

制約評価系は、上にも述べたように、ある制約の集合の標準形を保持していて、これに新しい制約が付け加わると、充足可能性を検査し、充足可能であれば新しい標準形を求めるという動作を行なう。

たとえば実数上の等式を制約として考えてみる。すると、たとえば新たに得られる制約が、< 定数 >=< 定数 >、あるいは < 変数 >=< 定数 > という形のみであるとするならば、上に述べたような制約評価は非常に容易になる。すなわち、値の同値性のチェックと、変数値の無矛盾性のチェックのみをすれば良いことになる。

ところが、一般にはこのような単純な制約のみを制約評価系に与えるとは限らない。したがって、与えられた制約に何等かの操作を施して、より単純な形へと変形、あるいは変換することが考えられる。たとえば、線形連立方程式を消去法で解くなどというのが、これにあたる。

ここでは、このような単純な形への変形 / 変換のことを、簡約と呼ぶことにする。

このような簡約を行なうためには、そのためのアルゴリズムが存在して、それが制約評価系の中に実装されている必要がある。

もし、この簡約アルゴリズムが扱えるような制約と、ユーザに記述を許している制約とが一致していれば、ユーザが書いた制約を直ちに簡約することが出来る。ところが、ユーザに記述を許している制約の中に、簡約アルゴリズムでは直ちに簡約出来ないものが含まれている場合には、この簡約アルゴリズムをそのまま適用することが出来ない。

このような制約は、ユニフィケーションや、他の制約評価の結果によって情報を得、その結果、簡約アルゴリズムを適用できるようになるまで待つのが一般的である。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

[3] ここでも、前の操作モデルと同様、制約は節毎にまとめて記述されるものとする。

のある変数が具体的な値を持つのを待って、その結果、制約が線形になれば、消去法で扱うことが出来るようになる。

これを制御するのを「遅延評価機構」と呼ぶことにする。

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。このような制約の扱いのことを「受動的な制約処理」と呼ぶ。

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。たとえば、消去法による制約評価は、「連立」方程式のためのものであって、能動的なものである。

上にも書いたように、遅延評価機構を利用して、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を遅延によって埋めようとしても、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うる。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、このような意味で「不完全な制約評価系」と呼ぶ。

もちろん、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。

また、簡約アルゴリズムは「漸増性 (incrementability)」[Miz-89] という性質を持っていることが望ましい。

「漸増性」とは、次のことを言う。

> 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。

4 制約論理プログラミング言語の例

この節においては、代表的な制約論理プログラミング言語について、例を中心に簡単に紹介する。この節では、Prolog III、CLP(ℛ)、CHIP、CAL の4つの言語について取り上げる。

4.1 Prolog III

Prolog III [Col-87] は、A. Colmerauer によって提唱された、最も初期の制約論理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。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 においては、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。

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

実際の処理系においては、非線形制約も扱うことが出来るが、これは遅延によって扱われる。すなわち、処理系は非線形制約に対して、他の制約やユニフィケーションによってある変数の値が確定し、線形制約になるまでその処理を中断する。これらの数値的な制約においては、各数値は有理数で表され、したがって計算誤差は発生しないようになっている。

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 } とで囲んで表す。

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) →
   Appetizer(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) →;
```

このプログラムに対して、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[4]。

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

また、次の例は、ブール制約の例で、以下の5つの前提を使って、「何かが常に存在していた」ことを示すのが目的である。

1. 何かが存在する。
2. もし何かが存在するならば、それは常に存在したか、もしくは無から生じたかのいずれかである。
3. もし何かが存在するならば、それはそれ自身の性質の必然によって存在するか、もしくは他のものの意思によって存在するかのいずれかである。
4. もし何かがそれ自身の性質の必然によって存在するならば、それは常に存在した。
5. もし何かが他のものの意思によって存在するならば、今存在するものは無から生じたとする過程は偽である。

ここで、次の命題の真偽値を示す5つの(ブール)変数を導入する。

```
a  :  何かが存在する
b  :  何かは常に存在した
c  :  何かは無から生じた
d  :  それはそれ自身の必然によって存在する
e  :  それは他のものの意思によって存在する
```

このとき、プログラムは、以下のようになる。

```
ValueOfSomethingHasAlwaysExisted(b) →
    {a=1,
    a ⇒ (b ∨ c) ∨ ¬ (b ∨ c),
    a ⇒ (d ∨ e) ∧ ¬ (d ∧ e),
    d ⇒ b,
    e ⇒ ¬ c};
```

これに対して、ゴール

```
ValueOfSomethingHasAlwaysExisted(x) ?
```

を実行すれば、解

```
{x=1}
```

が得られる。この場合、1はtrueを意味する。

また、次に示す例は、リストに関する制約を用いたものである。問題は、長さ10のリストで、左からリスト<1, 2, 3>を接続した結果と、右からリスト<2, 3, 1>を接続した結果が等しいものをみつけることである。

この場合、次のゴールを実行すればよい。

```
{|z|=10, <1,2,3> · z = z · <2,3,1>}?
```

すると、解

```
{z=<1,2,3,1,2,3,1,2,3,1>}
```

が得られる[5]。

この解はユニークであることに注意されたい。

4.2 CLP($\mathcal{R}$)

CLP($\mathcal{R}$) [Hei-87]は、制約論理プログラミング・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。すなわち、次がCLP(X)における主要な結果である。

論理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. 論理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. 論理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の論理的帰結であるとき、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

[4] Prolog IIIにおいては、与えられたゴール列を満す解が複数ある場合には、これらを次々に表示する。

[5] 実際の処理系では、これはリストではなく、タプルと呼ばれており、その長さはたとえば|x|=10と書くかわりにx::10と書く

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論理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論理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

   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。

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。

CLP($\mathcal{R}$) の処理系は、次の3つの構成要素から成る [Hei-87]。

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。

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。

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。これによって、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、不等式はシンプレックス法を用いて解かれる。このような制約評価アルゴリズムによって、CLP($\mathcal{R}$) の処理系は解が存在すれば、その解と yes を、矛盾すれば no を表示する。また、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。

```
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、複素数の除算が必要である。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*X - 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。

応用プログラムとして、CLP($\mathcal{R}$) で記述された、「オプション取り引き解析システム」 OTAS が作成されている。

ただ、CLP($\mathcal{R}$) の問題点は、浮動小数点数を用いているということで、そのために誤差が発生し、その誤差のためにプログラムの挙動が安定しない、あるいはユーザの望む動作をしないような場合が生じる[6]。

4.3 CHIP

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制約充足問題に分類出来るような問題を、制約論理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、これを論理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、Prolog III と同様に、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。

[6] そのかわり、効率は良い。

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。

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。ただし、ここで # は排他的論理和を表す。[7]

現在、ECRC で CHIP を開発したメンバーのほとんどはパリの Cozytech という会社に移っており、今後、どのような経緯になるかが注目される。

4.4 CAL

CAL(*Contrainte Avec Logique*) [Sak-89] は現在 ICOT において開発中の言語であり、ICOT で開発された逐次推論マシン PSI 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 (実部、虚部はそれぞれ有理数) 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

[7]

x	y	x#y
0	0	0
0	1	1
1	0	1
1	1	0

複素数上の線形・非線形代数方程式[8]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。

ブール値上の方程式[9]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[10]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[11]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、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($\mathcal{R}$) や CHIP 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 が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。また、何も書かなければ、ユニフィケーションを表す。

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.
```

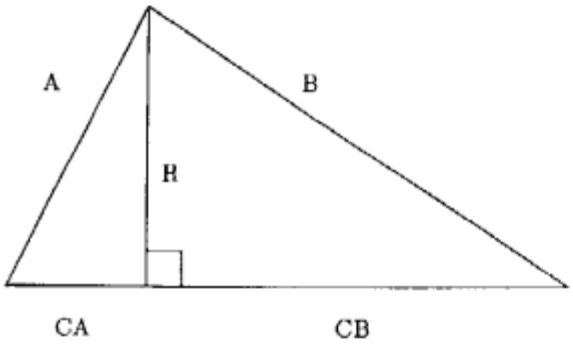
[8] 以下、「代数制約」と略す。
[9] 以下、「ブール制約」と略す。
[10] 以下、「集合制約」と略す
[11] 以下、「線形制約」と略す。

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

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

現在、Strum列を用いた、1変数代数方程式の実根求解機能との連結を進めており、これを行なえば、この解からs=6が得られる予定である。

5 現状における問題点と今後の課題

これまで見てきたように、制約論理プログラミングは、「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。

また、実際の問題に即した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、現在の制約論理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」を掲げている制約論理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。

数値データは、多くの制約論理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。制約論理・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。

ICOTにおいても、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、1変数多項式の実根を求める機能を付加するために、ある程度の計算誤差を容認するようなバージョンを開発したが、この実装においても、計算誤差の問題は大きな課題となった。

また、我々はICOTにおいて、制約論理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、これは制約論理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。

もうひとつ制約論理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。制約論理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論理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、並列制約論理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論理プログラミング言語GDCCを実装中であり、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、制約論理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論理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。すなわち、制約の集合に含まれるすべての制約を満たすような解が存在しなければ、全体の計算が失敗に終わり、要求された解は存在しないという結果に終わる。

ところが、問題によっては、制約を満たすにあたって、優先順位がつけられるようなものがある。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。これは、そのようにプログラミングを行なえば、制約論理プログラミング言語でも書くことは出来るが、そのためには、その言語の操作モデルがどのようなものであるかを知っていなければならない。

このような問題を制約論理プログラミング言語のときと同様に、宣言的に記述して解くことが、制約階層を導入する目的である。

このような、制約を満たすことに関する優先順位のみならず、制約が満たされない場合には、その「満たさない具合」を最小にするなどということも、この制約階層の考え方によって扱うことが出来る。

たとえば、ある製品を作るための物理的制約とコストの計算式について考えてみると、最初に考えていたコストが守れないからといって、これを全く無視して良いということはない。このような場合には、コストに関する制約は成立しないまでも、そこからのコスト上昇は最小に留めたいはずである。

このような制約階層を導入した制約論理プログラミング言語を階層制約論理型言語と呼ぶことにする。

ICOTにおいては、CALの処理系に基づいて、階層制約論理プログラミング言語CHAL (Contrante Hierarchique Avec Logique) の処理系を開発した。

現在、CHALで扱うことの出来る問題は、制約間に優先順位を導入したもののみであるが、例題として、次の2つの問題を記述し、解いている。ひとつはブール制約を用いた例で、自然言語を解釈する際にあらわれる曖昧性に優先順位を持たせることによって、構文解析を行なうような例である。また、もうひとつは、代数制約を用いた例で、規格部品を用いて変速機を設計する場合、規格部品を利用することを優先させて、それを用いては求める変速機が出来ない場合に新たな部品について考えるという優先順位を導入した例である。

いずれにせよ、制約論理プログラミングの研究の歴史はまだ数年しかない。行なわれるべき研究は、数多く、それに比較して行なわれてきた研究はまだ少ない。

制約による問題解決における一種のアドホックさ、問題毎にプログラム開発をしなければならない生産性の低さは、ある程度制約論理プログラミング言語によって解決されるにせよ、この制約論理プログラミング言語自身の歴史が浅いこともあって、従来の制約充足問題として解かれていた問題はどのようにすればこの言語を使って解けるのかが、まだ明確にされていない。これらを明確にし、要求される機能を洗い出すことが、制約問題解決における制約論理プログラミング言語の役割をはっきりとさせるためには必要である。

[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.