TM-0586他

89-1 ソフトウェア科学会
日本数学会秋季総合分科会

August, 1988

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

TM 0586　Sup－Inf法に基づいた制約論理型プログラミング言語
大木　優，澤本　潤(三菱電機)，坂根清和，藤井裕一

TM 0589　汎用日本語処理系(LTB)文生成部の生成方式
池田光生(松下電産)，重永信一，幡野浩司，福島秀顕

TM 0590　オブジェクト指向表現のための同一化による構成支援
片山佳則(富士通)

TM 0592　失敗集合に基づく並列論理型プログラムの宣言的意味論
村上昌己

TM 0593　半順序による証明手続きの解析
越村三幸，瀧　和男

TM 0607　逐次型推論マシンPSI上の汎用構造エディタSEMACS
小久保岩生，藤田正幸，富谷嘉一，吉武　淳(三菱電機)，梶山拓也(㈱アーティフィシャル・インテリジェンス)，坂井　公，横田一正

TM 0619　対話における環境的情報に関する一考察
本池祥子，野口直彦，安川秀樹

TM 0623　多層並列環境でのプログラム・プロトタイピングについて
松本一教，内平直志，本位田真一(東芝)

TM 0570　論証支援システムにおける論理の記述について
南　俊朗，沢村　一(富士通)

Sup-Inf法に基づいた制約論理型プログラミング言語
A Constraint Logic Programming Language based on Sup-Inf Method

大木優、沢本潤*、坂根清和、藤井裕一
(新世代コンピュータ技術開発機構)

線形不等式で表わされた数値に関する制約を解く機構をPrologに付加した制約論理型プログラミング言語について報告する。本制約論理型プログラミング言語では、線形不等式の制約を解くためのアルゴリズムとしてSup-Inf法を使用する。実現に際して、①不等式が連続して追加される状況において、制約を満たすかどうか調べる変数の数が少なくなるように、Sup-Inf法の適用を図る、②変数の値の範囲が指定された上限値と下限値の範囲を満たすことが確定することによってゴールが実行される遅延評価機構を組み込む、ことを行った。

1. はじめに

制約プログラミングは直習的プログラミングの1つとして、最近、脚光を浴びている[1][2][3]。制約の中でも不等式で表わせられた数値に関する制約を記述したり解くことは、工学的な応用に必要となってくる可能性がある。本研究を始める発端も、定性推論システムや装置の設計システムを作成するために、不等式で表わされた数値に関する制約を解くことが必要となったからである。本研究では、そのような応用のために、線形不等式で表わされた制約を解く機構をPrologに付加した制約論理型プログラミング言語を試作した。線形不等式の制約を解くアルゴリズムとして、効率的であると思われるSup-Inf法のアルゴリズムを用いた[3][4][5][6]。本制約論理型プログラミング言語を実現するに当たって、

① 不等式が連続して追加される状況において、制約を満たすかどうかを調べる変数の数が少なくなるように、Sup-Inf法の適用を図る、

② 変数の値の範囲が指定された上限値と下限値の範囲を満たすことが確定することによってゴールが実行される遅延評価実行機構を組み込む、

③ 非線形不等式も取り扱えるようにGrobner基底[7]の非線形連立方程式を解くプログラム(非線形連立方程式解決機)と結合する、

などの改善を行った。

本論文では、まず、Sup-Inf法について簡単に述べ、次に、本制約論理型プログラミング言語について述べる。

2. Sup-Inf法

まず、Sup-In法について簡単に述べる。Sup-Inf法は、Bledsoeによって開発され[4]、Shostakによって改良された線形連立不等式の各変数の上限値と下限値を計算するアルゴリズムである[5](この論文では、Sup-Inf法とはShostakによって改良されたSup-Inf法を指す。)。

Sup-Inf法を線形連立不等式に適用するためには、与えられた線形連立不等式を次のように変形する。

① 線形連立不等式を、以下のようにX＝＜Yという標準形式に変形する。

X＝＜Y　→　X＝＜Y
X＞＝Y　→　Y＝＜X
X＝Y　→　X＝＜Y, Y＝＜X

例えば、線形連立不等式{1＝＜3×X, Y＝X＋2}は{3×X＝＜1, Y＝＜X＋2, X＋2＝＜Y}のように変形する。

② 個々の式を個々の変数に関して上限値、下限値を求める式に変形する。

①の例では、

Xについて{X＝＜1/3, Y−2＝＜X, X＝＜Y−2}.
Yについて{Y＝＜X＋2, X＋2＝＜Y}.

のように変形する。

Sup-Inf法による上限値と下限値を求めるアルゴリズムを図1に示す[5]。図1は、上限値を求める関数Supとその補助

If	Action	Return
1. J is a number		J
2. J is a variable		
2.1 J ∈ H		J
2.2 J ∉ H	Q＝Upper(J)	
	Z＝Sup(Q,HU{J})	Supp(J,Simp(Z))
3. J＝rA where r is a number		
3.1 r＜0		r Inf(A,H)
3.2 r≧0		r Sup(A,H)
4. J＝rv＋B where r is a number, v a variable : B'＝Sup(B,HU{v})		
4.1 v occurs in B'	J'＝Simp(rv＋B')	Sup(J',H)
4.2 v does not occur in B'		Inf(rv,H)＋B'
5. J＝Min(A,B)		Min(Sup(A,H),Sup(B,H))

(1) 関数Sup(J,H)のアルゴリズム

If	Return
1. y is a number	y
2. x＝y	∞
3. y＝Min(A,B)	Min(Supp(x,A),Supp(x,B))
4. y＝bx＋c where x does not occur in c	
4.1 b＞1	∞
4.2 b＜1	c/(1-b)
4.3 b＝1	
4.3.1 c not a number	∞
4.3.2 c＜0	−∞
4.3.3 c≧0	∞

(2) 関数Supp(x,y)のアルゴリズム

図1 関数Supと関数Suppのアルゴリズム

関数Suppを示している。下限値を求める関数Infは関数Supのアルゴリズムと双対である。関数Supの第1引数Jは上限値を計算すべき変数である。第2引数Hは再帰的に計算している間に上限値を求めた変数のリストである。第1引数は再帰的に呼ばれている間に＋や×、max、minなどの算術オペレータを含むようになることがある。図1において、Upper関数は変数Xの上限値を返す関数である。線形連立不等式が{X＝＜U1, X＝＜U2, .., X＝＜Un, L1＝＜X, ..}のように変形できたとすると、Upper関数は以下のように定義される。

*現在、(株)三菱電機コンピュータ製作所

Upper(X) = min(U1,U2,..,Un) (n > 1の場合)
U1 (n = 1の場合)
∞ (n = 0の場合)

例えば、先の②で示した例では、Xの上限値はmin(1/3,Y−2)となる。また、関数Simpは×や＋をmaxやminの中に組み込む手続きである。

個々の変数の上限値と下限値は、個々の変数に対してSup-Inf法のアルゴリズムを適用することによって求めることができる。もし、関数Supを使って得た上限値が関数Infを使って得た下限値より小さいならば、与えられた線形連立不等式は解を持たないことが分かる。

3. Sup-Inf法に基づいた制約論理型プログラミング言語

Sup-Inf法使って制約論理型プログラミング言語を作成するに当たって、第1章で述べたように、いくつかの改良や改善を行った。

3.1 線形不等式の連続的追加に対するSup-Inf法の効率的適用

制約論理型プログラミング言語を使って、制約を満足する解を求める問題、すなわち制約問題を解く場合、一度に制約が与えられるのではなく、問題を解く過程で制約が次々に与えられることが多い。このような状況では、一括して線形不等式を解くアルゴリズムであるSup-Inf法をそのまま適用するのでは、不要な計算を行っている場合がある。

制約論理型プログラミング言語で制約問題を解く場合の多くは、主な興味は制約を満足しているかどうかで、制約を解くことに興味がある訳ではない。もちろん、最後は制約を解いた結果が欲しい訳であるが、途中の段階では、制約を満足していることを調べるだけで良く、制約を解く必要がないことが多い。このような背景から、一括して線形連立不等式を解く方法であるSup-Inf法を、次々に不等式が追加される度に、追加された線形不等式を含んだ全体の線形連立不等式を解くのに使うのではなく、全体の線形連立不等式が解を持つ可能性があるかどうか調べるだけに、使うようにする。

その方法は、解がある可能性がある(制約を満たしている)線形連立不等式にある線形不等式を追加した場合に、全体の線形連立不等式が解を持つ可能性があるかどうかは、全ての変数の上限値と下限値を調べなくても、次の方法で解を持つ可能性が分かる。

① 初めて現われる変数が追加された線形不等式に含まれているならば、変数の上限値と下限値を全く調べなくても全体の線形連立不等式は解を持つ可能性がある。

② 初めて現われる変数が追加された線形不等式に含まれていないならば、その不等式に含まれる任意の1つの変数について上限値と下限値を調べ、上限値が下限値より大きければ、全体の線形連立不等式は、解を持つ可能性があり、上限値が下限値より小さければ、全体の連立不等式は解を持たない。

この方法が正しいと証明するためには次の定理を証明すれば良い。

[定理] k個の線形不等式を持つ矛盾していない線形連立不等式に1個の線形不等式を追加した場合、全体(k+1個)の線形連立不等式が矛盾しているか否かを知るためには、線形連立不等式の中のある変数の上限値と下限値を(正しく)求める手続きを使って、以下のことを行うだけで十分である。

(1) 追加された線形不等式の中の変数にk個の線形連立不等式に含まれる変数以外の変数が含まれている場合は、矛盾かどうかを調べることなく、k+1個の線形連立不等式が矛盾していないことが分かる。

(2) 追加された線形不等式の中の変数にk個の線形連立不等式に含まれる変数以外の変数が含まれていない場合は、追加された線形不等式の中に含まれる高だか1個の変数が矛盾しているかどうかを調べるだけで十分である。

[証明]

まず、(1)の場合について証明する。

初めて現れた変数をZとすると、追加された不等式は線形不等式であるため、次の形のどれかに変形される。

Z < 線形の式、Z = < 線形の式、Z > 線形の式、
Z > = 線形の式、Z = 線形の式

Zは初めて現われたため、これ以外にZを制約する不等式はk+1個の線形連立不等式の中にはない。そのため、追加された式によって、k+1個の線形連立不等式は矛盾することはあり得ない。

次に、(2)の場合を証明する。まず、k個の線形連立不等式が矛盾していず、k+1個の線形連立不等式が矛盾するような場合を考える。

図2の(1)において、Ⓐをk個の線形連立不等式が満たす

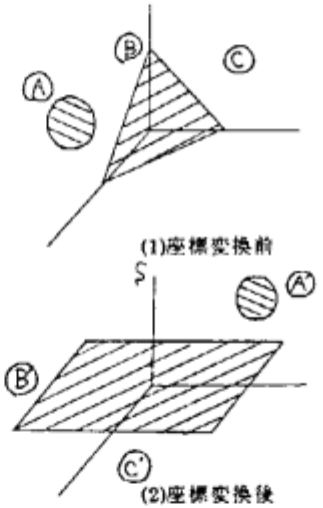

図2 新しい不等式を追加した状況

空間、Ⓑを追加された線形不等式が全空間を分割する平面、Ⓒを追加された線形不等式が満たす空間(この場合は、Ⓐに対してⒷの反対側)とする。このような場合、Sup-Inf法(以下、ある変数の上限値と下限値を求める手続きをSup-Inf法で代表させる。他の同様な能力を持つ手続きを使っても同じである。)は完全であるため全ての変数の上限値と下限値を求めたならば、ある変数で上限値と下限値が矛盾することを必ず見つける。今、Ⓑ平面に垂直に軸を取るように図2の(2)のように座標変換を行う。座標変換を行っても全体の線形連立不等式が解を持つか持たないかは保存されるため、この座標軸Sに関して、Sup-Inf法を使って上限値と下限値を求めれば矛盾することが分かる。追加された線形不等式に含まれる任意の変数IとSとの間には、座標変換の関係式I = g(..,S,..)という関係があり、この式と座標変換された全体の線形連立不等式を使って、Iの上限値と下限値をSup-Inf法で求めたならば、その上限値と下限値が矛盾することが求められる。すなわち、全体の線形連立不等式が矛

盾する場合は、追加された線形不等式に含まれる任意の変数の上限値と下限値をSup-Inf法で求めることにより、全体の線形連立不等式が矛盾することを見つけることができる。

線形不等式を追加してもk+1個の線形連立不等式が矛盾しないような場合は、k+1個の線形連立不等式が矛盾する場合の証明と同様な方法で、k+1個の全体の線形連立不等式が矛盾しないことを証明することができる。(証明終わり)

3.2 Grobner基底による非線形連立方程式解決機との結合

Grobner基底による非線形連立方程式解決機[7]とSup-Inf法による線形連立不等式解決機とを結合することにより、非線形連立不等式を受け付けることが可能となる。さらに、ある範囲の非線形連立不等式を解くことが可能になる。結合したシステムにおいて、線形及び非線形の方程式は、非線形連立方程式解決機に入力する。そして、その解いた答のうち線形の結果を線形連立不等式解決機に渡す。線形不等式は、直接、線形連立不等式解決機に入力する。非線形不等式は非線形方程式と線形不等式に分割し、非線形方程式は非線形連立方程式解決機に、線形不等式は線形連立不等式解決機に入力する。例えば、

X × Y > Z

は、次のように変換する。

T=X × Y, T > Z　　(ここで、Tは新しい変数)

3.3 不等式の評価の確定を使った実行順序制御機能

3.3.1 不等式の評価が確定することを待ってゴールの実行を行う遅延評価機構

Prolog IIなどに組み込まれているような変数の具体化を待ってゴールの実行を行う遅延評価機構に相当する機構[8]を導入する。この機能は、次の述語で実現される。

freeze__ineq(不等式、ゴール)

この述語は、第1引数の不等式が成り立つと確定されたならば、実行すべきゴールを登録する。もし、freeze__ineq述語を登録した時点でその不等式が成り立つと確定しているならば、直ちに、第2引数のゴールが実行される。不等式が成り立たないと確定した場合でも、失敗(fail)とはならない。

freeze__ineq述語の目的の1つは、深いバックトラッキング(deep-backtracking)を回避することである。例えば、次のような節がある場合、

```
p(X) :- X >= 0, q1(X).
p(X) :- X < 0, q2(X).
```

Xの範囲が分からないで、p(X)の最初の節を実行した後、X<0と分かると、q1などの実行がバックトラッキングされ、多くの無駄な計算を行う可能性がある。もし、X >= 0と分かった場合はq1(X)を実行し、X < 0と分かった場合はq2(X)を実行し、それ以外の場合には、q1あるいはq2の実行を遅延したいならば、freeze__ineq述語を使って、次のように記述することができる。

```
p(X) :- freeze__ineq(X >= 0,q1(X)),
        freeze__ineq(X < 0,q2(X)).
```

2引数のfreeze__ineq/2述語を拡張した3引数のfreeze__ineq/3述語を使うともっと簡単に書くことができる。

```
p(X) :- freeze__ineq(X >= 0,q1(X),q2(X)).
```

freeze__ineq/3述語の第1引数は不等式で、第2引数は不等式が成り立つと確定した時に実行されるゴールで、第3引数は不等式が成り立たないと確定した時に実行されるゴールである。図3の(1)にfreeze__ineq述語を使った例を示す。constraint述語は、本制約論理型プログラミング言語において不等式の制約を登録する基本的な述語である。(1)のtest11を実行すると3つのfreeze__ineq述語の条件部が成り立つことが確定し、そのゴールが実行されるが、test12を実行しても、どのfreeze__ineq述語の条件部も成り立つと確定できないため、freeze__ineq述語のどのゴールも実行されない。実行において、ゴールに先立つ"\"は不等式の制約を使うことを指定するものである。

```
test11 :-
    constraint([X>Y,A>B]),
    freeze__ineq(C>1,(write('C>1'))),
    freeze__ineq(A>1,(write('A>1'))),
    freeze__ineq(X>1,
     (constraint([B>2]),write('B>2'),constraint([C>2]))),
    constraint([Y>2]),
    nl.
test12 :-
    constraint([X>Y,A>B]),
    freeze__ineq(C>1,(write('C>1'))),
    freeze__ineq(A>1,(write('A>1'))),
    freeze__ineq(X>1,
     (constraint([B>2]),write('B>2'),constraint([C>2]))),
    constraint([Y>0]),
    nl.
```

(1) freeze__ineq述語のプログラム例

```
|?-\test11.
A>1
B>2
C>1
yes
|?-\test12.
yes
```

(2) (1)のプログラムの実行結果

図3 freeze__ineq述語のプログラム例と実行結果

3.3.2 場合分けされた不等式の評価が確定することを待ってゴールの実行を行う遅延評価機構

設計問題や計画問題では不等式による条件部が成り立つと確定した時に本体を実行するような形式の表現がしばしば使われる。そのような形式の表現を記述し易くするために、

① 条件部と本体の組を和(disjunction)の形で記述でき、
② 本体は条件部が成り立つと確定するまでその実行を遅延し、
③ ある時点まで条件部と本体の組の和で結ばれた物のどの条件部も成り立つと確定できなければ、どれかを強制的に成り立つと仮定し、本体を実行する、

機能を追加する。この機能はif__freeze__ineq節とmelt__if__freeze__ineq述語によって実現される。if__freeze__ineq節は、以下のようなシンタックスを持つ。

```
<if__freeze__ineq節> ::=
    <ヘッド> :- <条件部> $ <本体> #
                <条件部> $ <本体> #
                    :
                <条件部> $ <本体>.
<条件部> ::= <不等式> {,<不等式>}
<本体> ::= <ゴール> {,<ゴール>}
```

melt__if__freeze__ineq述語は引数を持たない述語である。if__freeze__ineq節の機能は以下の通りである。

① if__freeze__ineq節のある条件部が成り立つと確定すると、その本体が実行され、他の条件部の評価は中止される。

② 2つ以上の条件部が同時に成り立つことが確定したならば、上に書かれた条件部と本体が有効になる。その他は使われない。しかし、バックトラックされると、先に使われた条件部と本体の次に位置する条件部と本体の組が使われる。

③ どの条件部も成り立たないことが確定すると、この節の実行は失敗する。

melt_if_freeze_ineq述語は、if_freeze_ineq節の中でまだ本体の実行が遅延されているものにおいて、成り立たないと確定していない条件部のうちある1つが成り立つと仮定して、本体を実行する。もし、バックトラッキングが起きると、成り立たないと確定していない条件部のうち上から順に成り立つと仮定して、本体を実行する。成り立つと仮定できる条件部がなくなるとmelt_if_freeze_ineq述語は失敗する。簡単なプログラム例を図4の(1)に、そしてその実行結果を図4の(2)に示す。test21では、P1もP2もmelt_if_freeze_ineq述語の実行まで値も範囲も求まっていないが、melt_if_freeze_ineq述語でif_freeze_ineq節の条件部を評価して、最初の解を求める。test22では、P1の値がmelt_if_freeze_ineq述語を実行する前にわかっており、P2の値や範囲はわかっていない場合である。melt_if_freeze_ineq述語はP1が1の場合のP2の条件を求める。test23では、P1とP2の値がmelt_if_freeze_ineq述語を実行する前に、求まっている例である。この例では、melt_if_freeze_ineq述語は何もしない。

4. むすび

Sup-Inf法を利用した制約論理型プログラミング言語をProlog上に実現した。Grobner基底を求めるプログラムを除いたプログラム行数は、約1500行である。不等式の評価が確定することによる実行順序制御機能は、当初、3.1で述べた線形不等式の連続的追加に対するSup-Inf法の効率化を行う前に、実現していた。その時点では、線形不等式が追加される度にすべての変数の上限値と下限値を計算しており、不等式の評価はその値を使って行っていたので、不等式の評価が確定することによる実行順序制御機能は実行速度にそれほど影響を与えなかった。しかし、3.1で述べた方法によって効率化を行うと、実行速度に影響が出てくる。それは、実行順序制御に使われる不等式に含まれる変数が、追加された線形不等式に含まれていない場合、線形不等式が追加される度に、それらの変数の上限値と下限値を計算する必要があるからである。

残っている課題は以下の通りである。

① if_freeze_ineq節とmelt_if_freeze_ineq述語を設計などの実際的な例に適用する。

② 変数の依存関係によるグルーピング化を行うによって、実行順序制御を行う実行速度を改善する。使われている変数を依存関係によってグルーピング化することにより、追加された線形不等式に含まれる変数と依存関係のない実行順序制御に使われる不等式に含まれる変数は、その線形不等式が追加されても、それらの変数の上限値と下限値が影響を受けない。そのため、それらの変数の上限値と下限値を計算する必要がないことが分かる。

③ if_freeze_ineq節を呼び出したゴールの実行順序を動的に変更することによって、melt_if_freeze_ineq述語の実行速度を改善する。melt_if_freeze_ineq述語を実行する際、本体の実行が遅延しているif_freeze_ineq節を呼び出しているゴールを深さ優先に実行するのではなく、それらのゴールの実行順序を成り立つ可能性がある条件部の数が少ない順に、それらのゴールの実行順序を変更する。

```
test21 :- x(P1,P2),y(P1,P2), melt_if_freeze_ineq,
      output(P1,P2).

test22 :- x(P1,P2), constraint([P1 = 4]), y(P1,P2),
      melt_if_freeze_ineq,
      output(P1,P2).

test23 :- x(P1,P2), constraint([P1 = 4]),
      y(P1,P2), constraint([P2 = 6]),
      melt_if_freeze_ineq,
      output(P1,P2).

output(P1,P2) :-
    write('P1 = '), write(P1), write(' P2 = '), write(P2), nl.

x(P1,P2) :-
    P1 = 3, P2 = 2 $ true #
    P1 = 1, P2 = 3 $ true #
    P1 = 4, P2 = 5 $ true #
    P1 = 4, P2 = 6 $ true.

y(P1,P2) :-
    P1 = 1, P2 = 2 $ true #
    P1 = 1, P2 = 3 $ true #
    P1 = 3, P2 = 5 $ true #
    P1 = 4, P2 = 6 $ true.
```

(1) if_freeze_ineq節およびmelt_if_freeze_ineq述語の例

```
| ?- \test21.
P1 = 1 P2 = 3
yes

| ?- \test22.
P1 = 4 P2 = 6
yes

| ?- \test23.
P1 = 4 P2 = 6
yes
```

(2) (1)のプログラムの実行結果

図4 if_freeze_ineq節およびmelt_if_freeze_ineq述語の例と実行結果

謝辞

本研究のきっかけは、溝口教授と川村氏(以上、東京理科大学)の開発したSup-Inf法を使った制約論理型プログラミング言語のデモンストレーションを見ることができたこと、そしてSup-Inf法の能力を知ることができたことによる。ここに、溝口教授と川村氏に感謝致します。また、本研究を進める上で、援助をして頂いた相場氏、市吉氏、坂井氏、近山氏(以上、ICOT)に感謝致します。

参考文献

[1] J. Jaffer, S. Michaylov, Methodology and Implementation of a CLP System, Proc. of the 4th ICLP, 1987, 196.

[2] 川村、溝口、制約論理プログラミング言語とその適用例、日本ソフトウェア科学会第4回大会、D-2-3、(1987).

[3] 川村、大和田、溝口、CS-Prolog:拡張単一化に基づくConstraint Solver、Proc. of LPC'87, 1987, p21.

[4] W. W. Bledsoe, A New Method For Proving Certain Presburger Formulas, 4th IJCAI, 1977, p15.

[5] R. E. Shostak, On the SUP-INF Method for Proving Presburger Formulas, J of ACM, 24, 1977, p529.

[6] A. Bundy, The Computer Modelling of Mathematical Reasoning, Chapter 8, Academic Press, 1983.

[7] 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、SS87-28、1988.

[8] A. Colmerauer, Prolog II: Reference Manual and Theoretical Model, Internal Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1982.