TM-1164

計画問題向き並列論理型言語 GCL

上田　晴康、国藤　進

March, 1992

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

COMP 91－64
SS 91－21

計画問題向き並列論理型言語 GCL

GCL: A Parallel Constraint Logic Programming Language for Planning Problem.

上田晴康* 國藤進
Haruyasu Ueda Susumu Kunifuji
(株) 富士通研究所 国際情報社会科学研究所
International Institute for Advanced Study of Social Information Science,
FUJITSU LABORATORIES LTD.

1 はじめに

人工知能で扱われる難しい問題の一つとして、計画型の問題がある。計画型問題では、得られるべき計画の満たす条件が制約で表現されることが多い。そして計画型問題は解の満たすべき制約を宣言することにより自動的に解を求めようとする制約論理型言語との整合性が良い。このため、CLP(R)[1],PrologIII[2],CHIP[3] などの制約論理型言語を用いて計画型問題を解く試みがなされている [3]。しかしこれらの制約処理系で扱える制約は一般的なものであり、計画型問題特有の制約を表現するのが難しかったり、明示的に問題解決アルゴリズムを記述しなくてはならないことが多い。そこで、特に計画型問題でしばしば用いられるような制約に注目して、制約論理型言語の設計を行なったのが、GCL(Gaurded Constraint Language) である。

GCL は、もともと flab-GHC[4] と同じシンタックスを持ち、並列計算機上において高い並列度で効率良く計画問題を解くことを目的として設計されたが [5]、全解探索を行なうため計算量の爆発が起きやすかった。そこで、計画型問題をより効率良く解くために、二つの特徴を加えて再設計された。一つは、計画型問題に特有の順列組み合わせを表現する制約を扱えるようにしたことである。このことにより記述力が上がり、またこのような制約に特有の効率的なアルゴリズムを考えることが出来た。もう一つは、並列論理型言語の実行制御を、制約を用いて行なったことである。論理型言語の一つの欠点は後戻りや or 並列にかかる計算量が無視できないほど大きく、単純な実行制御ではすぐに計算量が爆発してしまうことである。そこで、非決定的な実行をできるだけ減らし、また与えられた制約から最適な実行順序を推論し、後戻りや or 並列実行をできるだけしないようにした。

本論文では GCL の設計方針および仕様について述べる。

*連絡先: 上田晴康 〒144 大田区新蒲田 1-17-25
E-mail ueda@iias.flab.fujitsu.co.jp

2 GCL の設計方針

GCL は計画型問題を容易に記述でき、かつ並列問題解決を行う言語である。また、制約を自然に宣言的に記述できる言語である。一方で、計画型問題は本質的に探索問題であるため、GCL では効率的に解を得ることを重視し、外部の世界とのインタラクションや、効率を上げること以外を目的とした同期を考えない。

これらの条件から、GCL は、制約を宣言的に記述できる論理型言語に、並列実行と制約解消をするように機能拡張したものとした。

並列実行と制約解消の機能を除いたGCLの言語仕様はシンタックス上は極めてflat-GHC[4]に似ている。どちらもプログラムは述語の定義からなり、それぞれの述語は、ヘッド、ガード、ボディからなる。また、ガードには限られたシステム定義述語のみが記述でき、ボディには、ボディ用のシステム定義述語とユーザ定義述語が記述できる。このため一見GHCとGCLのプログラムは区別がつかない。例えばappendのプログラムはGHCでもGCLでも図1のようになる。

```
append([],L2,Dest) :- true |
        Dest=L2.
append([H|T],L2,Dest) :- true |
        append(T,L2,Dest1),
        Dest=[H|Dest1].
```

図1: GCLのプログラムの例1

しかし、GCLはGHCと異なり、ガード部には制約が記述できる。例えば、GHCなどで単なるテストを表す$X < Y$という述語は、GCLでは、そのような関係を満たしていなくてはならないという制約を宣言する。

```
sample1(X,Y) :- X<Y |
        sample2(X,Y).

sample2(1,2) :- true | true.
sample2(4,3) :- true | true.
```

図2: GCLのプログラムの例2

これをプログラム例で示すと、図2となる。ここで、sample1(X,Y)を実行すると、GHCではX<Yをテストをしようとして、デッドロックを起こしてしまうが、GCLでは、X<Yという制約を宣言して、sample2/2の実行をし、制約を満たしているX=1, Y=2という値を返して終了する。

また、GHCではガードとボディの間にあるコミットバーはCommit & Choiceを意味しており、ボディを実行できるのはガードが成功している節のうちの任意の一つの節である。これに対し、GCLではガードとボディの間にあるバーは、単に非決定性の検出を意味していて、ガードが成功している全ての節は、バーを越えるとともにor並列に実行される。すなわち、一般のor並列Prologでのヘッドユニフィケーションによる節の選択を拡張して、ガードの述語や制約も成功した節を選択するようにしたものと考えられる。

例えば図2でsample2(X,Y)を実行すると、GHCでは、デッドロックしてしまうのに対し、GCLでは、X=1, Y=2とX=4, Y=3という二つの独立した答が得られる。図2をGHCでもデッドロックしないようにしたプログラム(図3)でsample3(X,Y)を実行すると、GHCでは、X=1, Y=2またはX=4, Y=3のどちらかの答えが得られるのに対し、GCLでは、図2と同様にX=1, Y=2とX=4, Y=3という二つの独立した答が得られる。

```
sample3(X,Y) :- true | X=1,Y=2.
sample3(X,Y) :- true | X=4,Y=3.
```

図3: GCLのプログラムの例3

3 制約解消系

一般に計画型問題は、 • 計画として正しいかどうかを判定する条件、 • 計画がどの程度望ましいかを判定する評価基準、 を与えられて計画を探索する問題である。計画として正しいかどうかが制約を用いて記述できるのは自明なことであるが、計画型問題で計画の望ましさを判定する評価基準を与えるということも、見方を変えると「評価を最大にする」という暗黙の制約を与えているとみなすことができる。そこで、正しいかどうかを判定する条件を強い制約、評価を最大にするという条件を弱い制約と呼ぶ([6])。

GCLでは、ガード部分に制約を記述することで強い制約が宣言される。このことは、強い制約を満たさなかった解は、failすることを意味する。failしない範囲を計算したり、failしているかどうかを検出する方法は従来からある制約解消系を拡張したものを使う。

一方、弱い制約については、計画の望ましさの

度合いが数値化されていると考えられる。このため弱い制約は、一般にはBranch & Boundまたは Simplex 法を用いて解決される。制約論理型言語の CHIP[3] などでもこの両者が用いられる。しかし、Simplex 法では、弱い制約を表現する評価関数が線型に制限されてしまうし、Branch & Bound では、少なくとも一つの計画が出来上がるまでは、他の計画の制御ができなくなってしまうので、or 並列に探索している最中の制御には使えない。

そこで、GCL では横型の探索にも使えるように Branch & Bound を改良したものを使う。すなわち、Branch & Bound と同様に出来上がった計画の望ましさを計算する方法を GCL で記述すると、GCL の処理系がその計算式に含まれる制約を解釈して、制作途中の計画の望ましさを推論し、推論された望ましさの高い計画から精緻化する。実現上は推論された望ましさの度合いを用いて並列計算の優先度を決めることにより制御を行う。

3.1 強い制約の記述とその解消

GCL で記述できる制約は、 • 線型一次不等式 ($<, \leq, >, \leq$), • 非等価 ($\neq$), • 有限体上での操作 (member/2,assign/2) の 3 種類である。有限体上の制約とは、変数の値が有限個の値の中から選ばれるというもので、計画問題、特に組み合わせを計算するときにしばしば用いる制約である [3]。例えば、member(X,[1,2,3,4]) は、変数 X が、1,2,3,4 のいずれかであることを宣言し、assign([X1,X2,X3],[1,2,3]) は、変数 X1,X2,X3 の値は、1,2,3 を適当に並べ換えたものであることを宣言する。これらの制約では、数値の代わりに a,b などのシンボルや f(x),[c,d,e(y)] などの複合項を使うことも出来る。

特に assign 制約は計画型問題 (組合せ問題) のために、GCL 処理系ではじめて導入した制約である。assign 制約は本質的に NP-complete の計算量を持った制約であるため、ヒューリスティクスサーチの手法を用いて計算量の爆発をできるだけ押えている。

これらの制約の解消のために、GCL はデータドリブンなアルゴリズムとトップダウンなアルゴリズムを用いている。データドリブンなアルゴリズムは従来の制約処理系で用いられていたのと同様に、制約の付加された変数に別の新しい制約が加わった場合に、矛盾が起きていないかどうか調べ、更にそれらの制約から変数のとり得る範囲に関する新たな情報がないか調べ、変数の取り得る範囲に関しての情報を制約を共有する他の変数に伝播する。例えば、$X < Y, Z < X$ という制約がある時、$X > 2Y - 1$ という制約が加わると、$X < Y$ と $X > 2Y - 1$ という制約から $X < 1, Y < 1$ という制約が見つかり、更に $X < 1$ を $Z < X$ を共有する変数 Z にで伝播して $Z < 1$ が見つかり、最終的に $X < 1, Y < 1, Z < 1$ が導き出される。

これに対し、トップダウンなアルゴリズムは探索空間を二つ (あるいは複数) に分けて or 並列に計算するための方法である。探索空間を複数に分ける方法は無限に考えられるため、分けた後での探索範囲ができるだけ小さくなるように、それまでに宣言された制約に関するヒューリスティクスを使って推論する。例えば $1 < X < 3$ という制約がある時 $1 < X < 2$ と $2 \leq X < 3$ の二つの探索空間に分けてそれぞれの場合を or 並列に計算することができる。GCL で用いたアルゴリズムは探索空間の分割の仕方を単純化して常に 2 つの探索空間に分割する方法を提案する。提案された分割のうちのどの分割を実際に選ぶかは 4.2 で述べる and 並列制御機構で決定される。

データドリブンなアルゴリズムもトップダウンなアルゴリズムも制約ごとに異なったアルゴリズムが使われているので、以下に制約ごとに用いたアルゴリズムをまとめて説明する。

線型一次不等式 については、データドリブンなアルゴリズムとして simplex 法ではなく sup-inf 法を用いる。sup-inf 法では、不等式の解が存在するかどうかと不等式に関係した全ての変数に関して上下限があるかどうかを新しい制約が加わる

たびに漸次的に調べる。simplex 法ではなく sup-inf 法を使ったのは、制約処理と同時に上限と下限を求めるので、トップダウンなアルゴリズムを使うのに便利なためである。

トップダウンなアルゴリズムは、sup-inf 法で求められた上下限を用いて探索空間を二つに分けようとする。具体的には上限も下限も ±inf でない変数に関して上半分の値を取る場合と下半分の値を取る場合の二つの探索空間に分ける提案する。例えば変数 X に関して $1 < X < 3$ がわかっている時、$1 < X < 2$ と $2 \leq X < 3$ の二つの場合に分けて、それぞれで探索を行なうことを提案する。

非等価性の制約 の解消は、データドリブンなアルゴリズムだけを用いている。なぜならば非等価性の制約に関しては探索空間を狭めるようなヒューリスティクスが見つからないためである。データドリブンなアルゴリズムでは、非等価性の制約がついている変数に member/2 制約もついている時、member 制約から不必要な候補を除く。

有限体上での制約 は、member/2 と assign/2 とで、制約の解消方法が異なるため別々に説明する。

- **member/2** は、データドリブンなアルゴリズムとして、矛盾解消アルゴリズム [3] を用いている。これは、制約のついた変数と他の変数の間で矛盾のない値だけを member の要素として残すというものである。例えば $1 < X < 5$ という制約がある時、`member(X,[-1,2,3,4,6])` という制約が加わると、両方の制約のインターセクションを取って、`member(X,[2,3,4])` と $2 \leq X \leq 4$ の二つの制約が残る。不等式の制約が残るのは冗長に見えるが、不等式制約のトップダウンアルゴリズムを使う可能性があるため残してある。

  member/2 のトップダウンなアルゴリズムとしては、member の要素が二つまで減った時に、その変数の値を二つの値のそれぞれに確定して、並列に探索することを提案する。

- **assign/2** はデータドリブンな制約解消を行なわない。なぜならば assign/2 の宣言により暗黙のうちに member/2 の制約も宣言され、member/2 がデータドリブンな制約解消を行なってくれるためである。

  assign/2 のトップダウンのアルゴリズムは、組合せ爆発をできるだけ避けるように問題をより小さな部分問題に分割しようとする。

  例えば

$$assign([X_1, X_2, X_3, ..., X_n], [k_1, k_2, k_3, ..., k_n])$$

という制約が与えられている時、この制約の変数 $X_1..X_n$ と定数 $k_1..k_n$ を適当な々くらいの大きさの二つのグループに分けて

$$assign([X_{11}, X_{12}, ..., X_{1m}], [k_{11}, k_{12}, ..., k_{1m}])$$
$$assign([X_{21}, X_{22}, ..., X_{2o}], [k_{21}, k_{22}, ..., k_{2o}])$$
$$\text{ただし} n = m + o$$

という二つの制約に分けることができた場合、探索空間が極めて小さくなっている。なぜならば、組合せ問題では問題の大きさ n に対して $O(e^n)$ の探索空間があるので、二つのグループに分けた後の探索空間は $O(2e^{\frac{n}{2}})$ となり、$O(e^n)$ よりも極めて小さな空間となるためである。

この様なことが可能なのは、それぞれの変数 $X_1..X_n$ と定数 $k_1..k_n$ を初めのグループに入れるか後のグループに入れるかが二通りの可能性を持つので、グループ分けの仕方は全部で 2^{2n} 通り作れるためである。

GCL ではヒューリスティクスサーチの手法を用いて、2^{2n} 通りの中から弱い制約を満たす可能性のある少数の組合せを見つけようとする。

このために、GCL の制約解消系はは assign/2 の制約が宣言されると、それに対して $2n$ 個の or 並列計算の提案を行なう。この提案のうち n 個は変数 X_i を X_{i_1} とするか X_{i_2} とするか $(i = 1..n)$ の提案であり、後の n 個は定数に関する同様の提案である。実際にこれらの提案をいつどのような順番で行なうかは 4.2 で述べるアルゴリズムを用いて決められる。そのアルゴリズムの概略は、or 並列可能な二つの計算環境のうちの片方の計算環境の優先度が低くてそれ以上計算を続行できない

(延期される) ような提案から選ぶというものである。

3.2 弱い制約の記述とその解消

3で述べたように、弱い制約は計画がどの程度望ましいかを数値的に表現したものである。そして、出来上がった計画から望ましさの数値を計算する方法をGCLで記述すると、GCLの処理系がその計算式に含まれる制約を解釈して、作成途中の計画の望ましさを推論して望ましい計画から順に作成されるように制御を行なってくれる。

まず弱い制約の記述方法を述べ、次に弱い制約の解消方法、すなわちどのように制約を解釈し、作成途中の計画の望ましさを推論するかを述べる。

3.2.1 弱い制約の記述方法

弱い制約は強い制約と異なり、何かのテストをするという性格を全く持たない。また、出来上がった計画を入力とし、数値を出力する述語として定義できる。GHCでこの様な述語を定義する時は、計画の一部をガードで見て、それに対応した値をボディで数値計算するという形を取る。そこで、GCLではボディに書かれた数値計算は全て弱い制約として扱うことにした。

図4が弱い制約を記述した述語の例である。preference/2では、コストに関する望ましさと満足感に関する望ましさをそれぞれ求めて4:6の割で足し合わせている。また、cost_to_priでは、コストからその望ましさを計算している。

この例で示されたPri := 0.4*Pri1+0.6*Pri2やPri1 := 4096*exp(-Cost/250)などはどれもGCLの処理系に望ましさの度合を計算するための方法とみなされる。

3.2.2 弱い制約の解消法

GCLは、処理系が特別扱いするトップレベルの述語gcl/4を持っている。このgcl/4は、最初の引数に実行するゴール、2番目の引数に計画、

```
preference(Plan, Pri) :- true |
    cost_pri(Plan, Pri1),
    satisfaction(Plan, Pri2),
    Pri := 0.4 * Pri1 + 0.6 * Pri2.

cost_pri(Plan, Pri1) :- true |
    cost(Plan, Cost),
    cost_to_pri(Cost, Pri1).

cost_to_pri(Cost, Pri1) :- Cost>1000 |
    Pri1 := 0.
cost_to_pri(Cost, Pri1) :- Cost<=1000 |
    Pri1 := 4096 * exp(- Cost / 250).

cost([Ele_of_Plan |T], Cost) :- true |
    fee(Ele_of_Plan, Fee),
    cost(T, Fee2), Cost := Fee + Fee2.
cost([], 0) :- true | true.

...
```

図4: 弱い制約の記述例

3番目の引数にその計画の望ましさの数値を入力として取り、4番目の引数に望ましさの数値の大きい順に計画が並んだリストを出力するメタな述語である。例えば図4の例から計画を作るには、

```
gcl(preference(Plan, Pri), Plan, Pri,
    List_of_Plans)
```

などと実行する。

この述語の機能は、全ての計画を調べて望ましさの度合の一番大きな計画から順に並べ換える述語のように見える。しかし、実際には gcl 述語はどの数値を最大化するかを確かめるだけの述語であり、GCLの処理系がその望ましさの数値の大きい計画だけをつくり出すように実行を制御する。

このためにGCLの処理系は、出来上がった計画の望ましさだけではなく、作成中の部分的に変数の含まれた計画の望ましさを推測しなくてはならない。作成中の計画は変数に色々な値を入れることで、望ましい計画も望ましくない計画も作ることができるので、作成中の計画を完成させた時にもっとの望ましい計画の望ましさを推測する必要がある。

GCLはこの推測を行なうために二つの方法を組み合わせて用いる。一つは変数の値を、制約を満たす範囲でランダムに与えてサンプリングし、そのサンプルの計画の望ましさを計算する方法であり、もう一つは、sup-inf法で求められた変数の値の上限と下限を弱い制約を通して伝播させ、最終的な計画の望ましさの度合の上限と下限を計算する方法である。サンプリングの方法は、いくつかのサンプルの望ましさの最大値を取ることによって、比較的良いと思われる計画の望ましさが得られる。上下限を求める方法は、サンプルの望ましさのばらつきの目安が得られる。

これらの値を用いて、現在作成中の計画の中でもっとも良い計画の望ましさの度合を計算するのに以下の式を用いる。

$$Priority = max(Max, SampledMax + (Max - Min)/4)$$

ただし$SampledMax, Max, Min$はそれぞれ、サンプリングして取った望ましさの最大値、計画の望ましさの最大値、最小値を表す。

この式はSampledMaxが大体平均の値を取り、また、推測された上下限の範囲が真の上下限の範囲に比べて広い範囲を示すという仮説を用いている。

SampleMaxはサンプリングして取った望ましさの最大値であるが、サンプル数に比べて探索範囲が広いためほぼ平均的な望ましさとみなすことができる。

また推測された上下限が真の上下限の範囲より広いのは、上下限の伝播が乗除に関しての推測が不正確なためである。例えば、$-2 < X < 3$という上下限と$Xsquare := X * X$という弱い制約からは$0 < Xsquare < 9$ではなく$-6 < Xsquare < 9$という上下限しか推測できない。このため推測された上下限の幅を半分に割り引いて計算している。

この推測された計画の望ましさの最大のものから計算を進めて計画を作るようにする。

この推測された望ましさの度合を用いて実際の優先度制御を行なうために、実現上の制限として、弱い制約で得られる望ましさの度合の数値は並列処理OSの優先度の範囲に収まらなくてはならないというのを加えた。[1]

そして、弱い制約の解消系により推測された望ましさの度合の数値を直接に並列実行の優先度として用いて、制御を行なっている。

現在のOSでは、優先度の高い実行環境がある限り優先度の低い実行環境が実行されないため、or並列に計算できる実行環境の数を抑制するのに役立っている。

4 実行制御機構

GCLの実行制御機構は、●リダクションを行なう時にヘッド、ガード、ボディをどのように実行するか(リダクションの制御)、●ゴールキューに入っているゴールをどのような順序に実行するか(and並列実行の制御)、●非決定的な選択の結果並列に計算できる計算環境をどのような順に計算するか(or並列実行の制御)の3つの制御機構からなっている。

or並列実行の制御については、3.2で述べたので、ここでは、リダクションの制御とand並列の制御について述べる。

4.1 リダクションの制御

リダクションは、通常のPrologではゴールにマッチした節のボディを単純にリダクションキューに加え、後戻りのための情報を残しておくだけである。また、一般のor並列(pure)Prologの場合は、ゴールにマッチした節の数だけその時の実行環境をコピーし、それぞれの実行環境でマッチした節のボディをリダクションキューに加える。さらに、GHCの場合は、ゴールにマッチした各節のガードを実行してみて、ガードの実行が全て終了した節があれば、それらの節のうち任意の一つの節を選んで、ボディをリダクションキューに加

[1]現在の処理系はPIMOS上で実現されており、優先度の範囲は0～4096である。

える。GHC では後戻りをしないため、計算環境を残す必要はない。

これらに対し、GCL では概念的には以下のようなリダクションを行なう。ゴールにマッチした各節のガードを実行してみて、ガードの実行が失敗しなかった節があれば、それらの節の数だけその時の実行環境をコピーし、それぞれの実行環境でマッチした節のボディをキューに加える。

ただし、GCL では実行環境のコピーをできるだけ押えるために上のリダクションアルゴリズムをさらに改良してある。

まずリダクションキューが通常の1本ではなく、 • 実行環境のコピーを必要としない決定的ゴール用キュー、 • ゴールに対して複数の実行環境を必要とする非決定的ゴール用キュー、 • 節のボディから取ってきたゴールを入れる一時キュー の3本ある。非決定的ゴールキューに入っている節は、一度キューに入れられた後も実行できる節が一つになって、決定的になると決定的ゴールキューに移される。非決定的ゴールが決定的になることを検出するために、非決定的ゴールは、ガードでテストしている全ての変数にフックしてある。

リダクションキューを3本持つため、GCL のリダクションはより正確には以下のようになる。

Step 1　一時キューが空でない間、そこからゴールを取り出し、 Step 1.1を繰り返す。

Step 1.1　ゴールにマッチした各節のガードを実行してみて、ガードの実行が失敗しなかった節を集める。そして、集まった節が一つしかなければ決定的ゴール用キューに、複数あれば非決定的ゴール用キューに入れる。

Step 2　決定的ゴール用キューが空でなければそちらから、空なら非決定的ゴールキューから、節を取りだす。
もしも両方とも空ならば実行終了。

Step 3　非決定的ゴールキューから取り出したら節の数だけ実行環境をコピーする。

Step 4　それぞれの実行環境でマッチした節のボディを一時キューに入れる。

Step 5　Step 1 ～ Step 4 を繰り返す。

以上のアルゴリズムからわかるように GCL では、and 並列を疑似並列で実現している。

ガードの実行に関しては、注目すべき点が二点ある。一点は、制約の処理をガードの実行中と (非) 決定的ゴールキューから取り出して or 並列に実行を始めてからの二回に分けていることである。わざわざ二回に分けているのは、ガードが逐次的に実行されるため、ガードの述語や制約の数が多いと効率が悪くなり、特に制約解消では完全な無矛盾性を追求すると大変に時間がかかるためである。ガードの実行中は既に sup-inf 法で用意された上下限や member,≠ に矛盾がないことだけを確認し、or 並列に実行を始めてから、3.1 で述べた制約解消をする。

もう一点は、ガードやヘッドユニフィケーションで確定した値を保存するローカル情報を持っていることである。なぜならガードの実行をしている間は、変数環境はまだコピーされていないので、それぞれの節で勝手に書き込んでしまってはいけないためである。そこで、GCL ではガードを実行する時には、ローカルな情報だけを書き換え、節が (非) 決定的ゴールキューに入れられる時にこのローカル情報も共にキューに入れられる。そして、このローカルな情報は、キューから取り出されて or 並列な実行が始まってから、グローバルな変数の環境に書き出される。

4.2　and 並列実行の制御

4.1 で述べたように、ゴールの選択はリダクションキューを複数持つことで制御される。しかしそれだけでは十分に効率的な実行はできない。なぜならば、制約処理系のトップダウンなアルゴリズムで作られた多くの並列計算の提案を何らかの方法で評価して、or 並列計算ができるだけ少なく

なるように実行の順序を最適化しなくてはならないためである。

非決定的実行の候補は非決定的ゴールキューにあるゴールと制約処理系から提案された分割方法とがあるが、GCLではゴールのリダクションの方を優先して行なう。なぜならば、ゴールのリダクションをすることにより、プログラムに記述された新たな制約が加わって、探索空間が大幅に減る可能性があるためである。

次に制約処理系から提案された分割方法から一つの分割を選ぶため、制約から提案された分割により優先度の上限と下限がどのように変わるかをそれぞれの提案ごとに調べる。そして以下の二つの条件を満たした提案の中から任意の一つを取り出す。

- 提案を実行した後できる二つの計算環境での優先度の上限が大きく異なる。すなわち、片方の計算はほとんど見込みがなくて、それ以上計算が進まなくなる。
- 提案を実行した後の優先度の上下限の幅が現在よりも狭くなる。すなわち、現在よりも計画が精緻化される。

このヒューリスティクスは、並列探索の環境を爆発的にコピーしないように、ヒューリスティクス・サーチしていることに相当する。これにより、組合せ問題などの計算量の多い問題でも明示的にプログラム中にヒューリスティクスを記述することなく解を得ることができる。

5 終りに

本稿では制約を用いて計画問題を記述するための言語 GCL とその処理系について述べた。GCLは以下のような特徴を持つ。

- 組合せ問題を直接記述できる記述力の高い制約を扱うことができる。またその制約を改良したBranch & Bound の手法と、ヒューリスティクス・サーチの手法を用いて効率良く解くことができる。
- 並列処理を記述するためのシンタックスとセマンティクスを持ち、理想的な場合、台数効果がほとんど台数分出る。
- GHC と同じシンタックスを持ち、GHC と同様に容易にプログラミングできる。

今後の課題としては、優先度の極めて低い計算環境の実行のために、変数環境のコピーをする代わりにバックトラックを用いて逐次型の非決定的計算をしてメモリ効率をあげることが考えられる。

謝辞 本研究の一部は第五世代コンピュータプロジェクトの一環として行なわれました。また、本処理系の実現検討にあたり、(株)富士通SSLの的野文夫、岩内雅直両氏に大変お世話になりました。

参考文献

[1] J. Jaffar and J-L Lassez. 単一化から制約へ. 制約論理プログラミング, pp. 31–50. 共立出版, 1989.

[2] A. Colmerauer. Prolog III 入門. 制約論理プログラミング, pp. 51–74. 共立出版, 1989.

[3] M. Dincbas, P. Van Hentenryck, H. Simonis, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. of FGCS88*, pp. 693–702, 1988.

[4] K. Ueda. Guarded horn clauses. Technical Report ICOT Tech. Report TR-103, Institute of New Generation Computer Technology, 1985.

[5] 上田晴康, 國藤進. 計画問題支援のための並列論理型制約処理系の設計と実現. 人工知能学会第5回全国大会, volume 1, 1991.

[6] 上田晴康, 國藤進, 岩内雅直, 大津建太. GRAPE の計画問題支援機能 — 並列制約論理型言語に基づくアプローチ —. 第12回知能システムシンポジウム資料, pp. 35–38, Oct 1990.