項書き換え系に基づく制約プログラミング言語システムの実現方式について

永井保夫*

1　はじめに

制約プログラミング言語は計算機援用設計（CAD)、物理システムのシミュレーション、VLSI,グラフィックス、人工知能などのさまざまな領域において応用が期待されている新しいプログラミングパラダイムの一種である。

制約という概念は対象間に成立する関係をあらわし、たとえば、図1のような、摂氏Cと華氏Fの関係$C=(F-32)\times 5/9$、オームの法則$V=I\times R$、三平方の定理$c^2=b^2+a^2$などがその一例である。制約を用いたプログラミングというアイデア自体は、新しいものではなく、様々な言語において設計・実現されてきた。そこでは、制約を用いた問題解決をおこなうために、制約という概念が対象間の関係を記述するだけでなく、これらの関係に基づき値を計算するために利用されてきた。前者は制約自身の記述のために用いられ、後者は制約をどのように解いていくか、すなわち制約処理の制御情報の記述のために用いられている。制約処理の制御は、制約を取り入れる言語に依存するものであり、たとえば、手続き型言語、関数型言語、論理型言語などがある。このように制約の記述ならびに制約処理機構を組み込んだ言語を制約プログラミング言語という。制約プログラミング言語において宣言的であるとは、制約が従来の手続き型言語における代入とは異なり方向性をもたない関係をあらわす。

制約プログラミング言語ではこのような宣言的な記述能力を提供することにより、プログラマーは目標だけを記述し、その目標をどのように達成するかという具体的なアルゴリズムを記述する必要がなくなるところに特徴がある。従来の手続き型プログラミングと制約プログラミングとの違いは、制約プログラミングでは、プログラマーが命令型言語によるプログラミングほどアルゴリズムにかかわらないで問題を記述し解けるところにある。たとえば、等式の扱いを比較すると、前者では等号ならびに代入に関する操作が必要なのに対して、後者では代入に関する操作は不要であり関係に関する操作のみが取り扱えばよいことになる。

その反面、制約プログラミングでは、アプリケーション依存性により制約充足システムを汎用問題解決器とみなして利用できないので、宣言的な記述能力により制約充足システムが全く解けない問題を容易に記述できてしまうという問題点が指摘されている。さらに、今までに開発された制約プログラミング言語が普及していない原因として、1）制約充足システムのアプリケーション依存性の強さ、2）操作されるデータタイプの固定化ならびに新たに定義される制約の追加不可、3）計算における完全性の保証の困難性、4）高階制約の未提供、5）制約充足技法の不十分性を挙げている。

制約プログラミング言語で用いられている制約充足手法と呼ばれる問題解決方法（局所伝播法、

*東京情報大学総合情報学部情報システム学科助教授
2002年11月7日受理

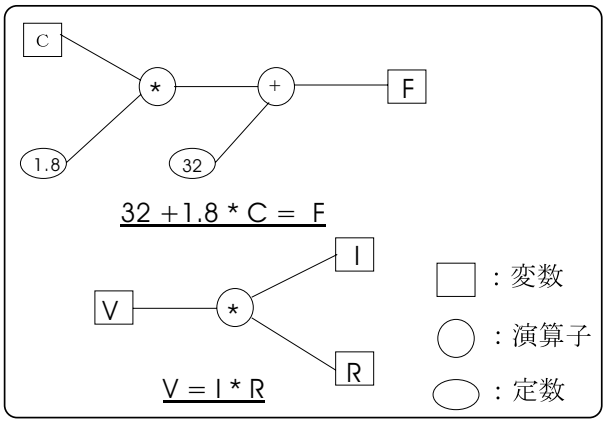

図1：制約の具体例

緩和法、グラフ変換ならびに等式解消法）の取り扱いでは, 次のような問題点があげられている［7］。

- 局所伝播法は制約プログラミング言語ではよく使われ、実現も容易で処理も高速である反面、サイクルを含んだ制約グラフを取り扱う（連立方程式を解く）場合には利用できないという欠点がある。
- 緩和法はサイクルを含んだ制約グラフを解くことはできるが、一般に処理に時間を要し数値制約しか取り扱えない。
- グラフ変換は項書換えを用いる方法であり、局所伝播法と同様に高速でコンパイル技法が使えるが、その反面、実現が困難なため制約充足システムでは利用されていない。
- 等式解消法はサイクルを含んだ多くの制約プログラムを解くために利用されているが、実現が困難で修正が容易でなく処理に時間がかかる場合がある。

このように、問題解決方法にはそれぞれ長所と短所があるので、目的に応じた使い分けが必要である。また、if/then形式の二階制約やメタ制約などの高階制約の取り扱い、時間を含む制約の取り扱い、デフォルト値の取り扱いについても説明されている［7］［20］。本論文では、上記のような問題点に対応するために、項書き換え操作を拡張した問題解決機構を検討し、これを組み込んだ制約プログラミング言語システムの実装、さらに適用問題とその実行例について説明する。

本論文の構成を以下に示す。第2章では、制約プログラミング言語システムと制約プログラミング言語の特徴ならびに研究動向について述べる。

第3章では、項書換え操作を拡張した新しい推論機構を組み込んだ制約プログラミング言語システムとその実装方式、さらに第4章ではこれに基づいたプログラミング言語と制約充足問題への適用例とシステムの実行例について説明する。

2 制約プログラミング言語システムの特徴ならびに研究動向

以下では、制約プログラミング言語の特徴ならびにこれらのシステムの研究動向について説明する。

2.1 制約プログラミング言語の特徴

制約を用いたプログラミングというアイデア自体は、新しいものではなく、様々な言語が設計・実現されてきた。そこでは、制約を用いた問題解決をおこなうために、制約という概念が対象間に

成立する関係を記述するだけでなく、これらの関係に基づき値を計算するために利用されてきた。前者は制約自身の記述のために用いられ、後者は制約をどのように解いていくか、すなわち制約処理の制御情報の記述のために用いられている。制約処理の制御は、制約を取り入れる言語に依存するものであり、たとえば、手続き型言語、関数型言語、論理型言語、オブジェクト指向言語などがある。手続き型言語や関数型言語では、プログラムの制御構造がプログラムの実行順序を示す。論理型言語に代表される宣言的言語では、計算順序の定義をユーザから開放し、プログラマーは特別なモジュールの実行条件だけを記述する。このように制約の記述ならびに制約処理機構を組み込んだ言語を制約プログラミング言語という。

通常のプログラミング言語では、プログラマーが問題の分析、問題を構成する対象間の関係の認識、解法の決定、問題の解き方を手続きとしてプログラムの作成をおこなう。これに対して、制約プログラミング言語ではプログラマーは問題を分析し、問題を構成する対象間の関係を認識し、問題中で成立する関係を制約充足に関する解法を意識せずに制約として記述する。そこで作成されたプログラムではプログラミング言語に組み込まれた制約充足技法により対象間の関係を満足する値が求められる（図2)。

以下では、図1の温度変換問題と電気回路のオームの法則を例として、制約指向技術の特徴を説明する。温度変換問題では式（1）という関係が成立する。摂氏温度から華氏温度を求める解法をFortranやCなどの手続き型言語で記述すると式（2）F=32＋C*1.8が必要となる。逆に、摂氏温度から華氏温度を求める解法を記述する場合には、式（3）C=（F - 32）*5/9が要求される。つまり、通常の手続き型のプログラミング言語では、このような温度変換問題において、双方向の変換を可能とするためには、式（2）と（3）の両方が必要とされる。また、オームの法則において多方向の計算を可能にする場合には、温度変換問題と同様に対象（変数）の数の増加に伴い、対象間の関係を記述する文は増加することになる。

一方、制約プログラミング言語では、対象間の関係の宣言的な記述を制約とみなすので、上記の温度変換問題では、式（1）のみを制約として与えればよいことになる。

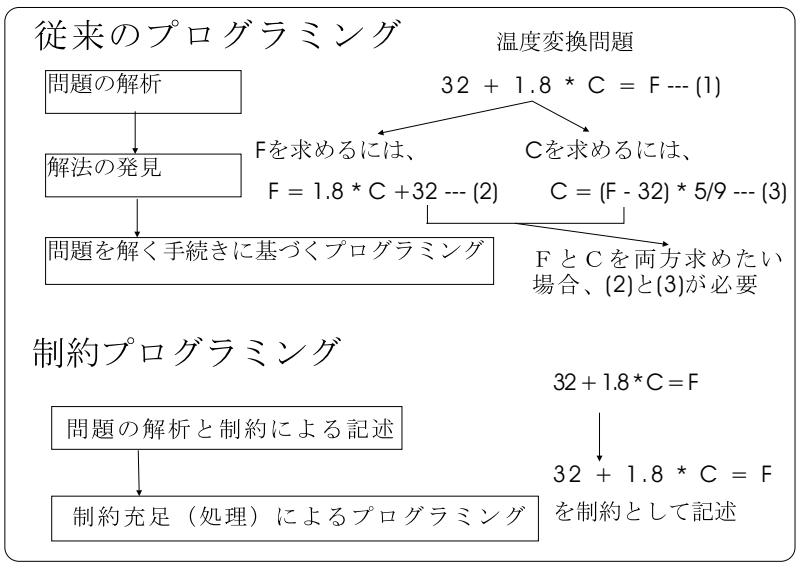

図2：制約プログラミング言語の特徴

このように制約プログラミングでは、制約知識を用いて記述能力の向上ならびに記述量の少ないプログラムのラピッド・プロトタイピングを容易におこなえる環境が提供される。したがって、プログラム開発における努力や開発されたシステムの保守や修正、あるいは拡張といった作業の困難さの軽減が期待できる。

しかしながら、制約プログラミング言語は記述能力に優れている反面、制約処理の部分がブラックボックス化されているために、ユーザからはプログラムの実行過程を把握しづらく、他の宣言型言語と同様にデバックが非常に困難なことが問題点である。

2.2 制約プログラミング言語の現状

2.2.1 手続き型言語

ALICE［5］は、CNRSで開発された組合せ問題向けの言語で、最適化問題をターゲットとしており、主にOR手法に基づいた問題解決機構が実現されている。

CHARME［10］は仏Bull社によりスケジューリング問題や資源割り当て問題を効率良く解くために設計された制約プログラミング言語である。制約論理プログラミング言語CHIPにおいて組み合わせ最適化問題を解くために用いられている技法を主要機能としてそのまま受け継ぎ、アプリケーション記述を容易にすることを目的として設計された。本言語の利用により記述量の少ない効率的なプログラムのラピッド・プロトタイピングならびにインプリメンテーションが容易になる。また、C言語の記述が可能なため、アプリケーションプログラムを効率的に実行できる。

EQUATRAN-M［19］は、方程式を翻訳してそれを解く手続きを作りだし、答を求めさらにその結果をグラフに表示する機能をもった方程式解法システムである。化学におけるプロセスの物質収支問題を解くためは大規模な方程式を解くことが必要となるが、その都度プログラムを作成することは大変であり、また同じ問題でも方程式を取り換えたりあるいは未知変数と既知変数を入れ替えたりすることがありプログラムの変更も容易ではない。EQUATRAN-Mは、このような問題に対処するために、方程式からプログラムを自動生成することを目的として設計されている。ここでは「制約」という概念は導入されていないが、方程式がまさに制約をあらわしていると考えられるので、制約プログラミング言語の一種であるといえる。

2.2.2 関数型言語

CONSTRAINTS［16］は、MITで開発された言語で、サイクルを含んだ制約プログラムを代数的に取り扱うために、局所伝播に基づいた代数的制約充足手法が用いられている。プログラム内での制約定義においてマクロの使用が許されている。しかしながら、この言語は再帰が許されず、制約のインスタンスの数ならびにインスタンス間の関係が固定される。

Bertrand［7］は、North Carolina大で開発された制約充足システム構築用言語で、項書き換えルールを用いて項書換え操作を拡張した推論機構が提供されている。

Mathematica［17］は、「制約」という概念を明示してはいないが、数式処理および数値計算をおこなうために設計された制約プログラミング言語の一種である。有理数や複素数ならびに記号式（制約）を混在して処理することが可能であり、厳密解の計算と近似解の計算も混在することが可能である。また、言語機能を用いてC言語などの言語ルーチンとリンクした新しい処理系を構築することが可能である。ここでは、計算結果をグラフや表にすることはもちろん、数式処理では数学的なあらゆる操作が可能である。たとえば、因数分解、代数方程式の求解、関数の微積分や微分方程式の求解などが可能である。特に、数式展開機能やグラフィックなどのインターフェイスは優れ

ている。対象は、化学、物理学、工学、経済学、統計学、数学などにおける数式処理を主とする分野である。

2.2.3　表計算言語（スプレッドシート言語）

表計算言語では、制約処理の順序が導出される解に対して影響を与えないため、制約指向プログラミングと非常に親和性がよい。制約プログラミングでは、対象間の関係を制約として宣言的に記述する能力を提供することにより、プログラマーは目標だけを記述し、その目標をどのように達成するかという具体的なアルゴリズムを記述する必要がなくなるという特徴を有する。制約指向表計算言語では、このような制約プログラミング言語と同様により記述能力の高い宣言的な関係が表現でき、柔軟でプログラミングの不用な環境を提供することができる。代表的な言語には、TK!Solver［21］がある。

2.2.4　制約論理プログラミング言語

制約論理プログラミング言語は制約に基づく問題解決機構を論理プログラミング言語に埋め込むことにより、問題の記述能力の向上と柔軟な問題解決能力の提供を目指したものである［15］［9］。

上記の点を考慮すると、制約論理プログラミング言語は次のような特徴を有する。

- 制約論理プログラミング言語では再帰的なルール適用が利用でき、実行時に制約のインスタンスならびにインスタンス間の関係について決定可能である。
- 制約の評価結果による計算の制御は操作性モデルに基づく。したがって、制約は実行の度に充足可能性の判断がおこなわれる。
- CLPスキームでは制約評価アルゴリズムは明記されていない。

制約論理プログラミング言語は、論理プログラミング言語PROLOGをベースとして、ある領域における制約の記述能力および制約処理能力を付加したものである。最初に実現された制約論理プログラミング言語はPROLOG-Ⅲ［23］である。この言語はPROLOG に制約を導入し、無限木上で線形等式制約ならびに不等式制約を取り扱い、有理数を計算領域としている。CLP（$\mathcal{R}$）は実数を計算領域として開発された言語であり、等式ならびに不等式が制約として記述できる。非線形の等式制約ならびに不等式制約は変数の値が具体化され線形になるまで、その評価が遅延される、CHIPは有理数上での線形制約ならびに線形不等式制約、ブール値に関する等式制約、離散領域を対象とした線形制約と線形不等式を取り扱うことができる言語である。その目的は制約を用いた探索問題を解くことにより組み合わせ（最適化）問題を取り扱うことにある。

- CHIP（Constraint Handling In Prolog）［3］
  スケジューリング、資源割り当て、レイアウト、故障診断、ハードウエア検証といった多くの実世界の問題はいずれも「制約」を利用した探索問題とみなすことができる。この種の問題は、そのほとんどがNP-完全問題のクラスに属している。このような問題を解く一般的な方法は手続き型言語で専用のプログラムを記述することであるが、プログラムの開発にかなりの努力が必要であり、さらに開発されたプログラムの保守や修正、あるいは拡張といった作業が困難になる。制約論理プログラミング言語CHIPはこのような問題点を従来の方法をもつ効率の良さを保持しながら、論理プログラミング言語の特徴である宣言性と柔軟性とを提供することにより解決しようとするアプローチをとっている。CHIP は記号的および数値的制約の問題解決技法により拡張されたProlog風の論理プログラミング言語であり、ECRCのDincbasとHentenryckにより開発された。

- CLP（$\mathcal{M}$）[11]
  多くの実際的な人工知能ならびに巡回セールスマン問題、ジュブショップ・スケジューリング、頂点被覆、ナップザック問題、混合整数計画問題に代表されるオペレーションズ・リサーチ研究では組み合わせ問題における解空間を探索することが要求される。このような問題の大部分はNP-完全問題のクラスに属し、適度な時間でよい近似解を得るためにヒューリスティクスを組み合わせた定式化が要求される。さらに、特殊なヒューリスティクスは定理証明で必要とされる充足可能性のテストというタスクやエキスパートシステムでの問い合わせ評価を高速化する。CLP($\mathcal{M}$）は、組み合わせ最適化問題のラピッドな定式化ならびにヒューリスティクスを用いた解法を提供し、問題を解くことを目的として設計されている。CLP($\mathcal{M}$）は、West Advanced Technologies社のLimらにより開発され、線形代数問題への適用ならびに数理プログラミング（混合整数計画法）に適用されている。特に後者の混合整数計画問題では、ヒューリスティックスを導入した分枝限定解法が提供されており、これを用いてORの代表的問題である施設配置問題が解かれている。

- CLP($\mathcal{R}$）[8]
  CLP($\mathcal{R}$）はMonash大のJaffarとIBM Watson Resarch CenterのLassezらにより開発された制約論理プログラミング言語である。「制約」という概念は、応用主体でさまざまな分野で利用されてきたが、このような概念を論理型プログラミング言語に導入し、従来の論理型言語が有している記号処理能力だけでなく、数値計算能力を強化した応用向け言語である。つまり、数値計算能力を強化するために論理型プログラミング言語に対して論理型プログラミング言語の論理的意味を崩すことなく、数理計画法で導入されている手法を取り入れ、拡張した言語であるといえる。そのため、言語自身も論理的にみて非常に美しい。

スケジューリングやレイアウト、資源割り当て問題といった実世界の問題を解くためには、ES構築支援ツールやAI言語を用いてエキスパートシステムを構築することが一般的なアプローチであるが、プログラム開発においてかなりの努力が必要であり、さらに開発されたシステムの保守や修正、あるいは拡張といった作業が困難である。制約プログラミングは、このような問題点を解決するために、宣言的知識表現を用いて記述能力を向上でき、記述量の少ない（効率的な）プログラムのラピッド・プロトタイピングが容易におこなえるので、システム構築支援に有益である。特に、CHIPはスケジューリングやレイアウト、資源割り当て問題などを取り扱うには非常に有効でありその研究動向には絶えず注目しておく必要がある。一方、CLP($\mathcal{R}$）では、オプション取引でのオプション解析などの論理的な制約と数値的な制約を組み合わせによる数式モデルの表現とその評価が必要とされる、すなわち推論（記号計算）と数値計算を結び付けた枠組の一例として非常に参考になり今後の研究についても注目していく必要がある。

3 項書換えシステム（TRS）に基づく制約プログラミング言語システム

3.1 項書換えシステム

項書換えとは、規則を用いて、項（term）を異なる表現の項に置き換えることである［25］［4］。ここで書き換えられる項を主項表現（subject expression)、規則を書換え規則（rewrite rule）と

呼ぶ。書換え規則は以下のようにヘッド、ボディの2要素から成り、「任意の項において、ヘッドにマッチする項があるならば、その項をボディで置き換えなさい」という手続き形態をあらわしている。

HEAD :- {BODY}

書換え規則の集合R、任意の項Eがあり、Eのなかに存在する項S（subexpression）がRのなかの規則R_kとマッチするものとする。このとき、SとR_kのヘッドをマッチさせ、ボディで置き換えることにより、新たな項S'が得られる。さらに、S'をEの元の位置に戻すことにより、Rによって書換えられたEの新たな形態E'が求められる。（Rのもとにおいて、EとE'は同一のものである。）

マッチングにおいて、書換え規則のヘッド、ボディには任意の項とマッチ可能な変数が含まれているため、項SとR_kのヘッドをマッチさせ、ボディに置き換える場合、変数の情報を伝播する必要がある。

以下は、その簡単な例である。ヘッド、ボディに変数Xを含む書換え規則R_kと、R_kとマッチング可能な項Sを持つ項Eである。

$X+0$: - {X}　　……（R_1）
$X\times1$: - {X}　　……（R_2）
$((5-3)+0)\times1$　　……（E）

R_kとEにおけるSは2箇所ある。まず、R_1とEでは（$(5-3)+0$）であり、（$(5-3)+0$）はR_1によって、（$5-3$）と書き換えられる。その結果、E'は（$5-3$）$\times1$と変形される。つぎに、R_2とE'では（$5-3$）$\times1$全体がSとなり、R_2によって（$5-3$）と変形される。

$$\underline{((5-3)+0)\times1}$$
$$\hookrightarrow \underline{((5-3))\times1} \quad \ldots \text{rewrited by } R_1$$
$$\hookrightarrow ((5-3)) \quad \ldots \text{rewrited by } R_2$$

このような変形を計算とみたとき、この書換え規則の集合は、ある種の計算過程を規定していることになる。つまり書換え規則の記述により計算手続きをプログラムすることができる。この書換え規則の集合を項書換えシステム（TRS：Term Rewriting System）と呼ぶ。

書換え規則により項をより簡単な表現の項に書き換えることを簡約（reduction）、書換え規則のヘッドにマッチング可能な項S（下線部）をリデックス（redex：reducible expression）と呼ぶ。また、リデックスの存在しない項、つまり、それ以上簡約できない項を既約項（irreducible）と呼ぶ。項書き換え系の簡約戦略には、最左最内戦略、最左最外戦略、並列最内戦略、並列最外戦略がある［25］、［4］。

3.2　制約プログラミング言語システム

本システムは、Prologによる推論エンジンと項書換えに基づく制約ソルバーからなる制約プログラミング言語システムの処理系である。（図3参照）

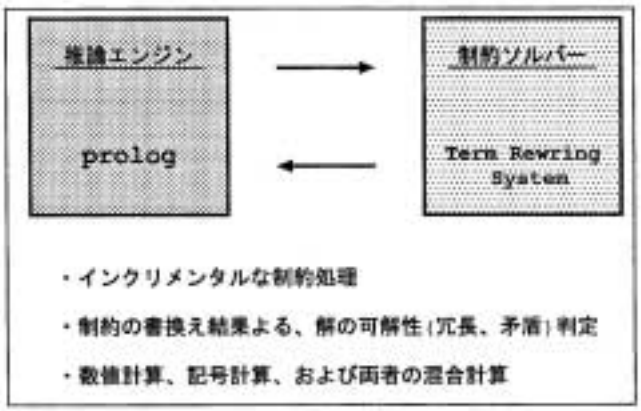

図3：推論エンジンと制約ソルバーの関係

本システムは、以下の機能と特徴を有する。

- インクリメンタルな制約処理が可能である。

- 制約の書換え結果より、制約の可解性（冗長性、矛盾性）が判定可能である。つまり、書換え結果が存在しない場合、解が存在しないことを示すことができ、書換え結果が存在する場合、解の存在を示すことができる。

- 数値計算、記号計算、および両者の混在計算が可能である。

また、繁雑になりがちな制約言語プログラミング、デバッグ環境を、柔軟なユーザインターフェースを構築することにより、快適なプログラミング環境を提供する。

3-2-1 制約ソルバー

制約ソルバーの構成を図4に示す。制約ソルバーは規則集合（rules)、規則集合を内部データに変換するパーザー（Parser)、内部データ、システムデータを登録、参照するデータベース(DataBase)、および項書換え処理系本体（TRSsolver）から構成される。制約ソルバーの処理手順は次のとおりである：まず、ユーザは問題（spec：書換え対象項）と解法に必要な規則（rule）を定義する。次に、ユーザが定義するシンタックスは文字式や数式といった一般に理解しやすい形式をとり、パーザーを用いて内部データのシンタックスに変換、登録される。最後に、項書換え処理系本体による書換え手続きを行なう。項書換え処理系本体はデータベースに登録してある規則を利用して、書換え対象項を書換える。書換え対象項が現規則において規約項となった場合、これを書換え結果（spec'）として出力する。

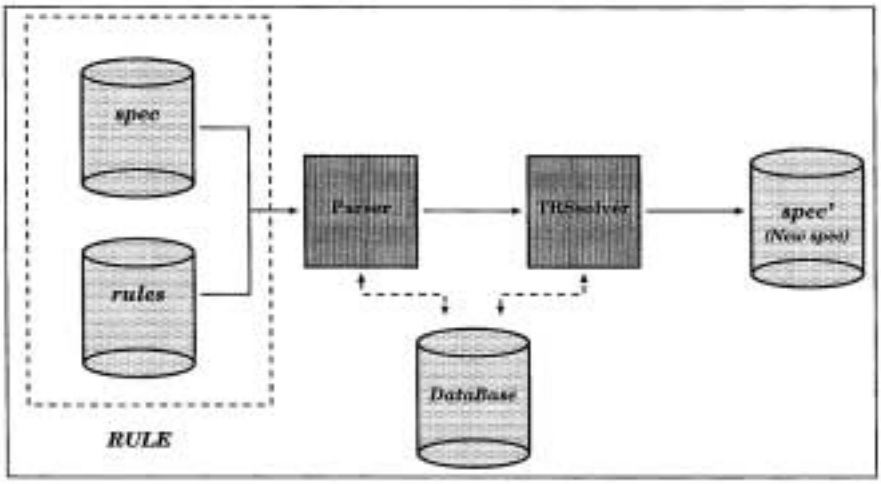

図4：制約ソルバーの構成

以下では、図4に示したシステム構成をもとに、各構成要素（規則集合、パーザ、データベース、TRSsolver）の機能と処理手順を示す。

（1）規則ライブラリ

規則ライブラリで取り扱う規則は、実行時にライブラリとして取り込む。これにより、問題に応じた規則の選択が可能となり、広範囲な問題を対象とすることができる。論理演算、四則演算、三角関数などを扱うための規則ライブラリを構成するデータであるオペレータ、タイプ、規則についてそれぞれ説明する。

- オペレータ

  基本ライブラリで用いるオペレータは表1に示されるように、引数無し（nullary）、引数1（unary）または引数2（binary , infix）を定義することができる。

  また、オプションとしてunaryは前置、後置（prefix , postfix）のいずれか、binary、infixは左右どちらかの方向性を示すleft-associative、right-associativeまたは方向性のないnonassociativeを宣言することができる。

  表1では、使用されているオペレータ名、意味、型、スーパータイプが示されている。各オペレータはカテゴリ毎（Statement , Expression , Logical , Relational , Arithmetic , Constants）に分けて表記してある。

- タイプ

  タイプは表2に示されるように、パラメータ変数やフリー変数の型であり、変数の束縛値を制限するものである。ヘッドのパラメータ変数をタイプ宣言することにより、ヘッドにパターンマッチ可能なサブジェクトは、ガードのような制限を受けることになる。タイプ宣言において、自身の宣言以前に宣言されたタイプを継承することができる。継承するタイプをスーパータイプ、継承を受けるタイプをサブタイプと呼び、サブタイプはスーパータイプの機能を引き継ぐ新たなタイプとして宣言される。

  上記の静的なタイプ宣言に加え、規則のタグを用いることにより、書換え手続きの実行中にタイプ宣言することも可能である。（動的なタイプ宣言）サブジェクトに適合する規則がタグ付き規則である場合、サブジェクトのラベルと規則のタグの組合せ（ラベル、タグ）

を作成し、これをタイプ空間に登録する。ラベルとタグの組合せは静的なタイプ宣言の変数とタイプの組合せに相当する。以下はサブジェクトx : : aNumberを書換え実行中に数値型としてタイプ宣言する規則である。x : : aNumberは規則aNumber :- true@numberを適用することにより、タイプnumber付きの変数x@numberとなる。

```
aNumber :- true@number        ...rule with tag
x : : aNumber                 ...subject（labeled term）
          ∇
x@number                      ...parameter with guards
```

図5は、規則ライブラリで使用されているタイプ間の関係を示す。タイプ宣言は処理系内部において、実装済みのbuildin、規則ライブラリ、基本オペレータ定義において静的宣言されているものに分けられる。

- 規則

規則はHead :- ｛Body｝ またはHead :- ｛Body｝@Tagの形で表現する。ヘッドとボディは記号（:-）とボディの括弧 ｛...｝ で区別し、ボディの後にアットマーク（@）とタグを記述する。タグを必要としない場合、アットマーク以降は省略することができる。

表3と表4では、それぞれ規則ライブラリで使われる基本オペレータ定義と規則ライブラリの等式規則が示される。各表は論理演算（Boolean）、比較演算（relational）、四則演算（algebra）などのカテゴリ毎に表記されている。**addition_primitive，subtraction_primitive**のように**primitive**の付く演算子は実計算による数値演算をあらわす。これら演算子は処理系による実数値計算を行なった後、計算結果を戻り値として出力する。規則のヘッド、ボディには、それぞれラベル、タグを設定することができる。ラベルはヘッドの前に記述することにより、オブジェクト（規則）を識別するインスタンス（ラベル）として機能する。タグは前述した動的なタイプ宣言に用いられる。

```
aNumber :- true@number        ...rule with tag
a : b : c : : aNumber         ...compound labeled term
```

動的なタイプ宣言はラベル、タグの組合せにより設定され、静的なタイプ宣言と同じようにタイプ空間に登録される。つまり、動的なタイプ宣言のラベルとタグは、静的なタイプ宣言の変数とタイプに相当する。

静的なタイプ宣言	変数	タイプ
動的なタイプ宣言	ラベル	タグ

このようにして設定されたタイプ空間は、サブジェクトとヘッドに含まれる変数のパターンマッチにおいて使用される。サブジェクトが変数、規則がタイプ付きパラメータ変数である場合、サブジェクトの変数タイプと規則のパラメータ変数タイプが同一である場合、または継承関係において同一である場合に限り、パターンマッチ可能となる。ここで、「継承関係において同一である」とは、同じスーパータイプを継承することを意味する。

以下にタイプ空間を扱う例題“datatype”の書換え過程を示す。例題“datatype”の目標はライン“l”が水平なラインとして定義されているか確認することである。規則（r2)、(r3)、(r4）はラインの特性をあらわしており、(r3）はラインが2つのポイント（p,q）から成り立つことをあらわし、(r2）はポイント（x,y）が数値x,yから成り立つことをあらわし

ている。さらに（r4）はポイント（p,q）のy 座標が一致する（水平なラインである）ことをあらわしている。規則（r1）の“t”はaLineのラベルであり、かつ、horizの引数（フリー変数）でもあることに注意する。

```
main          :- {t : : aLine ; horiz t}                              … (r1)
aPoint        :- {x : : aNumber; y : : aNumber ; true} @point         … (r2)
aLine         :- {p : : aPoint; q : : aPoint ; true} @line            … (r3)
horiz l@line  :- {l: p : y=l : q : y}                                 … (r4)
```

以下に、上記規則によるt : aLine ; horiz tの書換え過程を示す。

1. **t : : aLine**の書換え

   t : : aLineは規則（r3）の適用により、以下のように書き換えられる。(r3）はタグ（line）付きであるから、aLineのラベルtとの組合せによりタイプ宣言（line , t）が生成される。このタイプ宣言により、変数tのタイプがlineと定義されたことになり、以降に出現する変数t はline型として使用される。

   t : : aLine ==> t : p : : aPoint ; t : q : : aPoint ; true …rewrited by(r3),etc.

   [(t , line)]　　…new type space

2. **t : p : : aPoint ; t : q : : aPoint ; true** の書換え

   t : : Lineの書換え結果t : p : : aPoint ; t : q : : aPoint ; trueは規則（r1)、および規則ライブラリにより、以下のように書き換えられる。(r2）はタグ付き（point）であるから、前ステップ同様にあらたなタイプ空間が生成される。

   t : p : : aPoint ; t : q : : aPoint ; true ==> true　　…rewrited by(r2),etc.

   [(t , line) , (t : p , point) , (t : q , point) , (t : p : x , numvar) , (t : q : x , numvar)]

3. **horiz t**の書換え

   （r4）のパラメータ変数tはタイプ付き（line）パラメータ変数であるため、パラメータ変数tにパターンマッチする値は同一タイプlineのものでなくてはならない。そこで、horizの変数tのタイプが重要になる。horizの変数tは、これまでの手続きによりline型と定義されていることから、horiz tと（r4）は適合可能であることが分かる。実際に規則（r4）を適用すると、以下のように書き換えられる。

   horiz t@line ==> t : p : y@numvar=t : q : y@numvar …rewrited by(r2),etc.

   [(t , line) , (t . p , point) , (t . q , point) , (t . p . x , numvar) , (t . q . x , numvar)]

以上、ラインtはt : p : y@numvar=t : q : y@numvarに書き換えられた。このことから、ライン t は水平なラインであることが分かった。

(2) パーザー

ユーザが定義する問題や規則は文字式、数式といった一般に理解しやすいシンタックスを採用する。これにより、ユーザは規則を容易に記述することができる。一方、書換え処理系は、解析の柔軟性、容易性の面からPrologのリスト構造によるシンタックスを採用する。パーザーは、このようなシンタックスの相違を解消するために、ユーザ定義シンタックスを解析し、リスト構造シンタックスに変換するモジュールである。

表5はユーザ定義シンタックス（Operational）とリスト構造シンタックス（Executable）との

対応関係を示している。

以下では、これらの対応関係の中で用いられているパラメータ変数とフリー変数（var, parameter, compound var, compound parameter）、ガード（parameter with guard）、定数（numeric constant）、項（term, labeled term, compound labeled term）、is演算子（is expression）の表現方法をそれぞれについて説明する。

- パラメータ変数とフリー変数

  単一の変数はアトムで表現し、複合変数はコロン（：）を区切りとして単一変数を組み合わせて表現する（オペレータ宣言されていないアトムは全て変数として扱う）。また、規則のヘッド、ボディ両方に出現する変数は、変数にマッチした値を引き継ぐ目的から、パラメータ変数と呼び、ボディにのみ出現する変数をフリー変数と呼ぶ。フリー変数はヘッドに出現することはなく、作業エリアとして使用する。

- ガード

  タイプ付きパラメータ変数は、アットマーク（@）を区切りとしてパラメータ変数とタイプの組合せで表現する。パラメータ変数は付随するタイプと同一タイプの値にのみ束縛され、その他のタイプ値では束縛されない。（タイプは、自らの宣言以前に宣言されたタイプを継承することができる。継承するタイプをスーパータイプ、継承を受けるタイプをサブタイプと呼ぶ。サブタイプとスーパータイプは一方向の同一関係にあり、サブタイプのパラメータ変数はスーパータイプの値に束縛することができる。）

  タイプ付きパラメータ変数はヘッドにのみ出現し、ボディに出現することはない。

表1：規則ライブラリで使われる基本オペレータ

Category	Operator	Description	Associativity	Supertype
Statement	;	assert	right	numop
	;;	assert	right	numop
Expression	,	list	right	numop
Logical	->	implication	right	boolean
	or	Boolean or	left	boolean
	~ or	nor	left	boolean
	&	Boolean and	left	boolean
	~ &	nand	left	boolean
	~	Boolean not	prefix	boolean
Relational	=	equality	nonassociative	boolean
	~ =	inequality	nonassociative	boolean
	>	greater than	nonassociative	boolean
	>=	greater or equal	nonassociative	boolean
	<	less than	nonassociative	boolean
	<=	less or equal	nonassociative	boolean
Arithmetic	+	addition	left	numop
	-	subtraction	left	numop
	*	multiplication	left	numop
	/	division	left	numop
	round	round to integer	prefix	numop
	floor	greatest integer less than	prefix	numop
	abs	absolute value	prefix	numop
	sin	sine	prefix	numop
	cos	cosine	prefix	numop
	tan	tangent	prefix	numop
	atan	arc tangent	prefix	numop
	^	raise to power	right	numop
	-	unary minus	prefix	numop
	^^	scientific notation	nonassociative	numop
Constants	true	Boolean	nullary	boolean
	false	Boolean	nullary	boolean
	nil	distinguished	nullary	
	++	linear expr	right	linearop
	**	linear term	non	
	lx_merge	used for adding linear expr's	prefix	
Typed object	aNumber	declare a number	nullary	

表2：規則ライブラリで使用されるタイプ

Category	Type	Description
builtin	`positive`	positive numeric constants
	`negative`	negative numeric constants
	`nonzero`	numeric constants except zero
	`constant`	numeric constant
	`literal`	string constants
	`true`	the boolean constants
	`false`	the boolean constants
operator	`numvar`	variables that represent numbers
	`sterm`	simple numeric variables and constants
	`numop`	arithmetic operators
	`boolean`	boolean expressions
	`string`	string expressions
rule	`expr`	any expression
	`boolean`	boolean expression
	`string`	string expression
	`number`	numeric expression
	`numop`	numeric operators
	`linear`	linear expression or term
	`linearop`	linear operator
	`sterm`	simple term

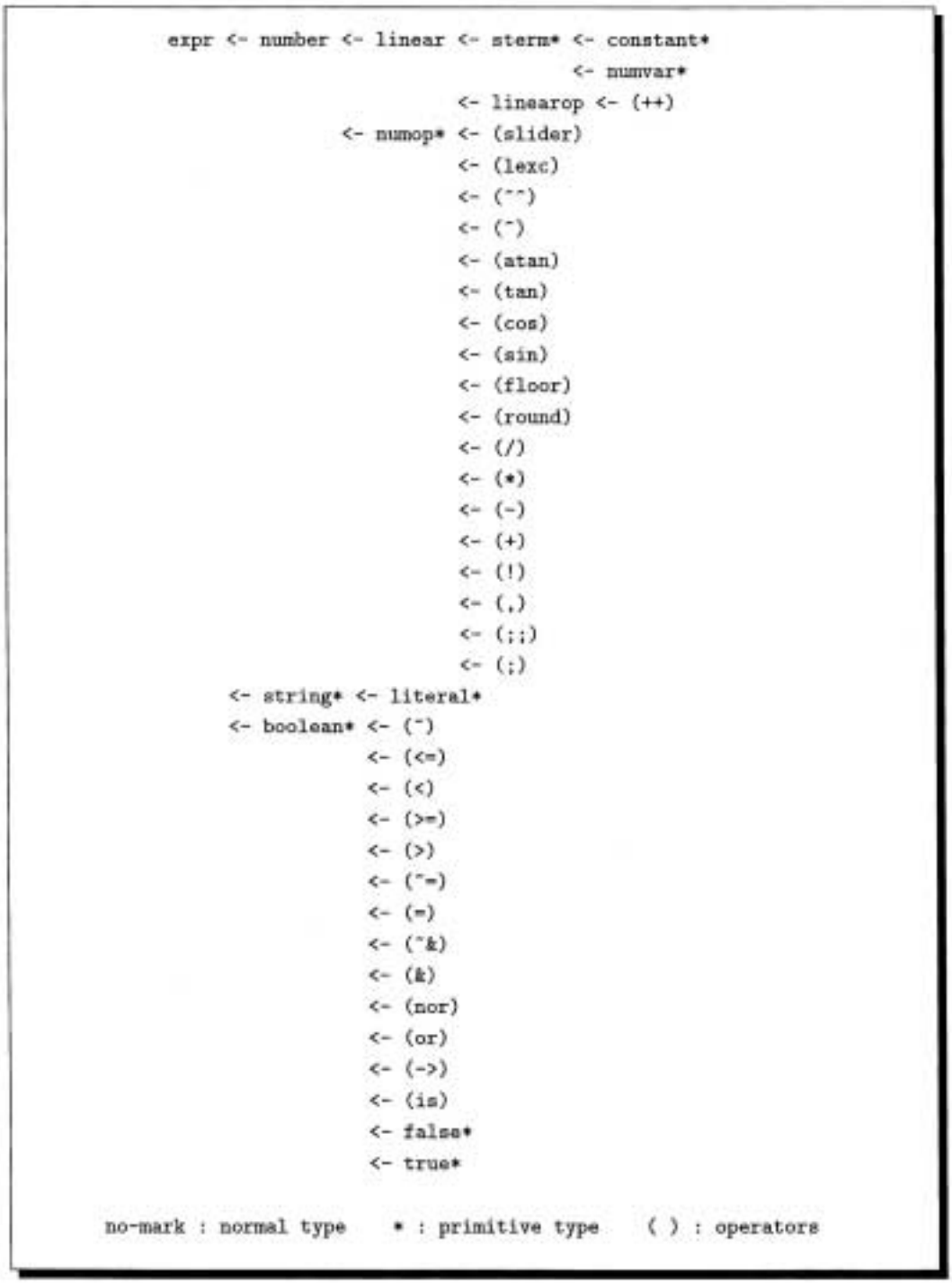

図5：規則ライブラリで利用されるタイプの継承関係

表3：規則ライブラリで利用される基本オペレータ定義

... Semicolon	
true;a	{ a }
(a;b);c	{ a;b;c }
(a&b);c	{ a;b;c }
... Boolean	
a -> b	{ ~ (a & ~ b) }
a ~ & b	{ ~ (a & b) }
a or b	{ ~ (~ a & ~ b) }
a nor b	{ ~ a & ~ b }
false & a	{ false }
a & false	{ false }
true & a	{ a }
a & true	{ a }
~ ture	{ false }
~ false	{ true }
~ ~ a	{ a }
... Relational	
a > b	{ b<a }
a >= b	{ b<=a }
a ~= b	{ ~ (a=b) }
a@constant < b@constant	{ lessthan_primitive }
a@constant <= b@constant	{ lessorequal_primitivew }
a@constant = b@constant	{ equality_primitive }
... Numbers	
aNumber	{ true }@numvar
a@constant + b@constant	{ addition_primitive }
0 + b	{ b }
a@constant - b@constant	{ subtraction_primitive }
- b@constant	{ 0 - b }
a@constant * b@constant	{ multiplication_primitive }
0 * b	{ 0 }
1 * b	{ b }
a@constant / b@nonzero	{ division_primitive }
0 / b	{ b ~=0 ; 0 }
a@constant ∧ b@constant	{ power_primitive }
a ∧ 0	{ 1 }
a ∧ 1	{ a }
a ∧ ∧ b	{ a * 10 ∧ b }
... Misc. Functions	
sin a@constant	{ sin_primitive }
cos a@constant	{ cos_primitive }
tan a@constant	{ tan_primitive }
atan a@constant	{ atan_primitive }
round a@constant	{ round_primitive }
floor a@constant	{ floor_primitive }
... Builtins	
a@numvar lexc b@numvar	{ lexcompare_primitive }
v@numvar is e	{ bind_primitive }

表4：規則ライブラリの等式規則

... quick and dirty	
`n@numvar = k@constant ; rest`	`{ n is k ; rest }`
... Creating linear expressions	
`k@constant * v@numvar`	`{ k * ((1**v)++0) }`
`v@numvar * k@constant`	`{ ((1**v)++0) * k }`
`v@numvar + n@number`	`{ ((1**v)++0) + n }`
`n@number + v@numvar`	`{ n + ((1**v)++0) }`
`v1@numvar + v2@numvar`	`{ ((1**v1)++0) + ((1**v2)++0) }`
`v@numvar = n@number`	`{ ((1**v)++0) = n }`
`n@number = v@numvar`	`{ n = ((1**v)++0) }`
`v1@numvar = v2@numvar`	`{ ((1**v1)++0) = ((1**v2)++0) }`
... Standard form	
`a@number - b@number`	`{ a + (-1 * b) }`
`- a@number`	`{ -1 * a }`
`a@number / k@nonzero`	`{ (1/k) * a }`
`a@nonzero = b@number ; c@expr`	`{ 0 = a - b ; c }`
`a@linearop = b@number ; c@expr`	`{ 0 = a - b ; c }`
... Moving terms together	
`a@linear + (b@linear + c@numop)`	`{ (a + b) + c }`
`a@numop + b@linear`	`{ b + a }`
`a@linear * (b@linear * c@numop)`	`{ (a * b) * c }`
... Nonlinear transformations	
`a@constant = b ^ c@constant`	`{ a^(1/c) = b }`
`a@constant = b / c`	`{ c =0 ; b/a = c }`
... multiply a linear expression by a constant	
`k@constant * ((c**v) ++ rest)`	`{ ((k*c)**v) ++ (k * rest) }`
`((c**v) ++ rest) * k@constant`	`{ ((k*c)**v) ++ (k * rest) }`
... add a constant to a linear expression	
`k@constant + ((c**v) ++ rest)`	`{ (c**v) ++ (k + rest) }`
`((c**v) ++ rest) + k@constant`	`{ (c**v) ++ (k + rest) }`
... add two linear expressions	
`((c1**v1@numvar)++r1) + ((c2**v2@numvar)++r2)`	`{ lx_merge( (1+(v1 lexc v2)), ((c1**v1)++r1), ((c2**v2)++r2)) }`
`lx_merge( 0, (t1++r1) , lx2 )`	`{ t1 ++ (r1 + lx2) }`
`lx_merge( 2, lx1 , (t2++r2) )`	`{ t2 ++ (lx1 + r2) }`
`lx_merge( 1, ((c1**v1)++r1) , ((c2**v2)++r2) )`	`{ lx_merge( (c1 = -(c2)), ((c1**v1)++r1), ((c2**v2)++r2)) }`
`lx_merge( true, (t1++r1) , (t2++r2) )`	`{ r1 + r2 }`
`lx_merge( false, ((c1**v1)++r1) , ((c2**v2)++r2))`	`{ ((c1+c2)**v1) ++ (r1 + r2) }`
... solving	
`0 = ((c**v@numvar) ++ rest@number) ; ex`	`{ (v is ((-1/c) * rest)) ; ex }`
... cleaning up after a bound variable	
`(c@constant ** k@constant) ++ rest`	`{ (c*k) + rest }`
`(c@constant ** lx@linearop) ++ rest`	`{ (c*lx) + rest }`

- 定数
  定数は整数または実数で表現する。
- 項
  項はオペレータ宣言されたアトム（オペレータ）と引数、および、コロン２つ（::）を区切りとするラベルを組合せて表現する。（ラベルを不要とする場合、記述する必要はない。しかし、内部ではラベルを必須とするため、Parserはラベルのない項を検出した場合、数値0を規定値ラベルとして追加変換する）ラベルはパラメータ変数、フリー変数同様、複合ラベルを表現することができ、コロン（:）を区切りとして単一ラベルを組み合わせて表現する。
- is演算子
  is演算子はフリー変数の束縛を行なう演算子である。オペレータ“is”と左辺のフリー変数、右辺の束縛値で表現する。is演算子の左辺は必ずフリー変数である。

表5：ユーザー定義シンタックスとリスト構造シンタックス

Expression Type	Operational	Executable
variable	`name`	`[var,name]`
compound variable	`n1:n2:n3`	`[var,n1,n2,n3]`
parameter	`pname`	`[parameter,pname]`
compound variable with first element a parameter	`p1:p2:p3`	`[parameter,p1,p2,p3]`
parameter (with guard)	`pname@type`	`[typed,pname,type]`
parameter (multiple guards)	-N/A-	`[typed,name,t1,t2,t3]`
numeric constant	`123`	`[constant,123]`
term	`op(args)`	`[term,[0],op,args]`
labeled term	`label::op(args)`	`[term,[label],op,args]`
compound labeled term	`l1:l2::op(args)`	`[term,[l1,l2],op,args]`
is expression	`expr1 is expr2`	`[is,expr1,expr2]`

階乗を求める例題のシンタックス変換例を図6に示す。[1)]

(3) データベース

データベースは問題、規則、システム状態をあらわすフラグなど本機能が必要とするデータを登録参照する領域である。データは節形式で登録し、必要に応じて追加、参照、削除（Prolog言語のassert, clause, retract, eraseを利用）を行なう。以下では、データベースに登録するデータである規則とその登録形式を示す。

---

1) 実際には、ヘッド、ボディ、タグ、演算子、規則IDなどを1つにまとめた項（'{DB-rul}>>>'(TopOp,SubOp,Head,Body,Tag,ID)）に変換している。ここではヘッド、ボディの変換を確認し易くするため、Head :- Bodyの形を崩さず記載した。

```
#include beep.
#operator(fact,prefix,900).
fact 1                              :-  { 1 }.                    (r1)
fact n@constant                     :-  { n * fact (n - 1) }.      (r2)
                                    ▽
[term,[0],fact,[constant,1]]        :-  [constant,1]              (r1')
[term,[0],fact,[typed,n,constant]]  :-
  [term,[0],*,[parameter,n],
    [term,[0],fact,[term,[0],-,[parameter,n],[constant,1]]]]    (r2')
```

図6：階乗のシンタックス変換

```
#include beep.
main :-             {
                     4.6237 * a + 2.6914 * b - 3.7517 * c =  1.4023 ;  (r11)
                    -2.4037 * a + 1.0432 * b + 0.7589 * c =  0.3724 ;  (r12)
                     1.0462 * a + 2.0495 * b + 6.3524 * c = -2.4728 ;  (r13)
                     a, b, c                                           (r14)
                    }.
                                    ▽
[term,[0],main] :-
                    [term,[0],=,[term,[0],-,[term,[0],+
                      [term,[0],*[constant,4.6237],[var,a]]
                        [term,[0],*,[constant,2.6914],[var,b]]
                          [term,[0],*,[constant,3.7517],[var,c]]]]
                            [constant,1.4023]] ;                       (r11')
                    [term,[0],=,[term,[0],+,[term,[0],+
                      [term,[0],*,[constant,-2.4037],[var,a]]
                        [term,[0],*,[constant,1.0432],[var,b]]
                          [term,[0],*,[constant,0.7589],[var,c]]]]
                            [constant,0.3724]] ;                       (r12')
                    [term,[0],=,[term,[0],+,[term,[0],+
                      [term,[0],*[constant,1.0462],[var,a]]
                        [term,[0],*,[constant,2.0495],[var,b]]
                          [term,[0],*,[constant,6.3524],[var,c]]]]
                            [constant,-2.4728]] ;                      (r13')
                    [var,a],[var,b],[var,c]                            (r14')
```

図7：連立方程式のシンタックス変換

- 規則

  規則はヘッド、ボディ、タグ、ヘッドのトップオペレータとサブオペレータ、規則IDを要素とする項で表現する。（タグは不要な場合、省略される）

  ```
  '{DB-rul}>>>'(TopOp,SubOp,Head,Body,Tag,ID)
  ```

  ヘッド（Head）、ボディ（Body）、タグ（Tag）のシンタックスについては前述のパーザを

参照して頂きたい。ここでは、トップオペレータ、サブオペレータ、規則IDについて説明する。トップオペレータ（TopOp)、サブオペレータ（SubOp）は、ヘッドの第1、第2 演算子であり、これらはパターンマッチの処理効率を改善するためのものである。問題と規則のパターンマッチは書換え手続きの最も重要な部分であり、全工程の大半を占めている処理である。パターンマッチの処理効率は、規則数の増加と項の複雑な構造に左右され易い。そこで、Prolog処理系のハッシュ関数による節選択を利用したパターンマッチを導入する。Prolog処理系では、節選択において、第1、第2引数をハッシュ関数にかけることで、節の選択処理を効率よく行なっている。規則節にトップオペレータ、サブオペレータ（第1、第2引数）を設定することにより、Prolog処理系による規則節の選択処理軽減が期待できる。IDはトレース情報などに用いる規則毎の識別番号である。詳細は以下の規則IDを参照。

- システムラベル
  システムラベルは本機能が自動的に付加するラベル（整数値（n≥0)）である。データベースには、整数値を登録しておき、必要に応じて追加、参照を行なう。ラベル参照後は、次回参照に備えて、新たなラベル（参照値+1）を登録する。

Labeled term	label (in database)
[term,op,[0],args]	'{SYS}CNT'(label,1)
[term,op,[0,1],args]	'{SYS}CNT'(label,2)
[term,op,[0,1,2],args]	'{SYS}CNT'(label,3)

- 規則ID
  規則IDは規則毎の識別番号であり、トレース情報などに用いる。データベースには、整数値（n ≥1）を登録しておき、必要に応じて追加、参照する。参照後は、次回参照に備えて、新たなID（参照値＋1）を登録する。

RULE	ID (in database)
	'{DB-rul}NUM'(1)
'{DB-rul}>>>'(TopOp,SubOp,Head,Body,Tag,r1)	'{DB-rul}NUM'(2)
'{DB-rul}>>>'(TopOp,SubOp,Head,Body,Tag,r2)	'{DB-rul}NUM'(3)

- グローバルネーム（変数束縛情報）
  グローバルネーム空間は、問題と規則のパターンマッチにおける変数の束縛情報を管理する領域である。データベースには、変数と束縛情報を登録しておき、is演算子等により変数の置換処理が生じた場合、これを参照する。

```
'{DB-spc}GSP'(Var1,Restriction1)
```

- タイプ
  タイプ空間は、タイプ、プリミティブタイプ宣言されたアトムを管理する領域である。データベースには、タイプ宣言されたタイプ名と、タイプまたはプリミティブ属性を登録する。また、継承するスーパータイプがある場合、そのタイプ名を登録する。
  タイプ宣言次第では、サブタイプ、スーパータイプ間の継承関係は複雑になる可能性がある。データベースには、複雑な継承関係を明確にするため、サブタイプ、スーパータイプ

の継承関係を対1で登録する。

プリミティブタイプτ1、タイプτ2、τ3の継承関係が、τ3→τ2→τ1であるとき（τ1はτ2のスーパタイプであり、τ2はτ3のスーパタイプである。また、τ1はτ3のスーパータイプである。）、タイプ空間は以下のようになる。

```
'{DB-opn}TYP'(τ1,'{ROOT}',primitive)
'{DB-opn}TYP'(τ2,τ3,type)
'{DB-opn}TYP'(τ3,τ2,type)
'{DB-opn}TYP'(τ3,τ1,type)
```

- オペレータ

  オペレータ空間は、オペレータ宣言されたオペレータを管理する領域である。データベースには、オペレータ宣言されたオペレータ名、型、結合力と、本宣言以前に、宣言されていた型、結合力を登録する。また、継承するスーパータイプがある場合、そのタイプ名を登録する。

  登録するオペレータ名、型、結合力はProlog処理系のオペレータ宣言と同じものであり、オペレータ名は宣言されたアトム名、型はf[2] fx, fy, xf, yf, xfx, xfy, yfxのいずれか、結合力は整数値（$0 \leq P \leq 1200$）で登録する。

  スーパータイプはタイプ名を登録する。

  ```
  'DB-opnOPN'(Op,Type,Prec,TypOrg,PrecOrg,SuperType)
  ```

- タイトル、状態フラッグ、その他

  問題のタイトル、システム状態を示すフラグ、その他システムに必要な規定値、ファイル名などの情報をデータベースに登録する。表6に前述のデータを含めたデータベースに登録するデータと形式を示す。

（4）TRSsolver

TRSsolverでは、書換え手続き戦略として最左最外（outermost）、最左最内（innermost）、両戦略の組み合わせという3つの戦略を実現している。以下では、各書き換え戦略の処理概略を示す。

---

[2] Prolog処理系にf は存在しない。本機能ではオペレータ宣言されていないアトムを全て変数として取り扱うため、引数なしの単一アトムにもオペレータ宣言を必要する。Prolog処理系は単一アトムにオペレータ宣言を必要としない。

表6：データベースへのデータの登録形式

Type	Expression Type	Executable
title	problem's title	'{DB}TTL'(Title)
operators	type relation	'{DB-opn}TYP'(Type,SuperType,Kind)
	operator definition	'{DB-opn}OPN'(Op,Type,Prec,TypeO,PrecO,SuperType)
rules	rule	'{DB-rul}>>>'(Op,SubOp,Head,Body,Tag)
	rule id	'{DB-rul}NUM'(Int)
	macro expand	'{DB-mac}MAC'(Macro,Expand)
space	global name space	'{DB-spc}GSP'(Var,Rest)
histories	deleted data	'{HIS}'(Clauses)
system	defaults	'{DFT}'(Clauses)
	trace flags	'{FLG}TRC'(Int)
	history flags	'{FLG}HIS'
	command file flags	'{FLG}COM'
	bell flags	'{FLG}BEL'
	undo flags	'{FLG}UND'(File)
	mode and prompt	'{SYS}MOD'(Mode,Prompt,Mrem)
	command file name	'{SYS}COMfile'(File,Flag)
	current file name	'{SYS}CURfile'(File,Flag)
	undo file name	'{SYS}UNDfile'(File,Flag)
	tty flags	'{SYS}TTY'
	working directory	'{SYS}WKD'(Dh,DT)
	macro expand	'{SYS}MAC'(Macro,Expand)
	macro operator	'{SYS}MACop(Op,Type,PrecOrg,Prec)
	update flags	'{SYS}UPD'
	counter	'{SYS}CNT'(Kind,Int))

- **最左最外戦略**

本手続きは、書換え対象項の優先順位を左側から右側（最左)、外側から内側（最外）とすることにより、停止性を保証しないTRSにおいても、無限ループを起こすことなく終了させることが可能な戦略である。

以降では、書換え対象とする項をサブジェクトまたは項、項の一部を部分項（項自身を含む）と表記する。但し、サブジェクト、項、部分項は同一のものとして表記する場合もある。

1. 束縛変数の展開
   グローバルネーム空間の変数束縛情報を参照し､サブジェクトの未束縛変数を展開する。
2. 規則の適用
   項全体に対して規則を順次適用する。途中、部分項の書換えが生じた場合、項を再構築し、あらたな項として再帰的な書換えを行なう。逆に、いずれの規則も適用されず、項の書換えが起きない場合、以降の手続きによる部分項の書換えを行なう。
3. 書換え可能な部分項の検索
   既存の規則において、書換え可能な部分項を検索する。部分項の選択は最左最外戦略に基づいて行なわれる。
4. 部分項への規則適用
   検索した書換え可能な部分項に規則を適用し、部分項を規則ボディに書換える。書換え結果を新たな部分項として項の再構築、再帰的な書換えを行なう。
5. 書換え手続きの終了
   既存の規則において、項が規約項となった場合、書換え手続きは終了である。

- **最左最内戦略**

  本手続きは、書換え対象項の優先順位を左側から右側（最左）、内側から外側（最内）とすることにより、前述の最左最外戦略より効率よい処理が期待できる戦略である。但し、停止性を保証しないTRSでは、無限ループを起こす可能性がある。

  1. サブジェクトの部分項展開

     サブジェクトがフリー変数、かつ束縛済みフリー変数の場合、束縛値への置換と書換えを行なう。サブジェクトがフリー変数以外の項である場合、項の部分項展開と各部分項毎の書換え手続きを行なう。

  2. 部分項の書換え

     部分項と規則のパターンマッチ、部分項の規則ボディへの置換などの書換え処理を繰り返す。書換え処理は部分項が規約項となるまで再帰的に繰り返される。

  3. 項の再構築、書換え

     規約項となった部分項をもとに項の再構築と再帰的な書換え処理を繰り返す。

  4. 書換え手続きの終了

     既存の規則において、項が規約項となった場合、書換え手続きは終了である。

- **最左最外戦略と最左最内戦略との組み合わせ**

  本手続きは、最左最外戦略と最左最内戦略を組み合わせた特殊な戦略である。まず、最左最内戦略により、項の外側から内側に向けて書換えを行ない規約項とする。つぎに、最左最外戦略により、規約項となった各部分項をもとに、項全体の再構築と書換えを行なう。

  1. 束縛変数の展開

     グローバルネーム空間の変数束縛情報を参照し、サブジェクトの未束縛変数を展開する。

  2. 書換え可能な部分項の検索

     既存の規則において、書換え可能な部分項を検索する。部分項の選択は最左最外戦略に基づいて行なわれる。

  3. 部分項への規則適用

     検索した書換え可能な部分項に規則を適用し、部分項を規則ボディに書換える。部分項は規約項になるまで、再帰的に書換え手続きを繰り返す。

  4. 項全体の書換え

     規約項となった部分項をもとに、項全体の再構築と書換えを行なう。

  5. 書換え手続きの終了

     既存の規則において、項が規約項となった場合、書換え手続きは終了である。

4　適用問題と実行例

4.1　適用問題

（A）階乗計算

図8は引数nに与えられた数値の階乗を求める規則集合である。規則（r1）は数値1の階乗は数値1であることをあらわし（終了条件）、規則（r2）はnの階乗計算$n\times(n-1)\times(n-2)\times\cdots\times1$を再帰的に表現したものである。

ここで、引数*n*に付随する@*constant*は、*n*に適合する型を定数に制限するものである。*n*=5の場合の書き換え過程を示す。まず、問題*fact5*を与える。*fact5*は（r2）のヘッド*fact n@constant*とパターンマッチ可能であるから変数*n*を数値5に束縛し、これをボディ5×*fact*(5－1）に置換する。次に5－1の減算処理を行ない、5×(*fact*(4)）に置換する。(ここでは、四足演算の規則を明記していないが、減算処理も規則による書き換え手続きとして行なわれる）さらに*fact*(4）について同様の書換え手続きを繰り返すことにより、5の階乗120を得ることができる。

このように規則のヘッドとのパターンマッチ、ボディへの置換手続きを繰り返すことにより最終的な解答を得ることができる。

（B）連立方程式

図9は変数*a*，*b*，*c*の解を求める連立方程式である。変数*a*，*b*，*c* の係数は固定値として与えられており、各式をオペレータ（；）によりassertした後、*a*，*b*，*c* のそれぞれの値を求めるものである。なお、先の例題では、数値*n*をパラメータとして外部から与えたため、規則と問題は個別に与えたが、本例題では変数*a*，*b*，*c* の係数を固定値としており、外部からのデータは必要としないため、問題と規則を同一ボディ内に定義している。

```
fact 1           :-  { 1 }                          ... (r1)
fact n@constant  :-  { n * fact (n-1) }             ... (r2)
                          ▽
>> fact 5
   => fact 5  --> 5*fact (5-1)
>> 5*fact (5-1)
   => 5-1  --> 4
>> 5*fact 4
   => fact 4  --> 4*fact (4-1)
>> 5* (4*fact (4-1))
   => 4-1  --> 3
>> 5* (4*fact 3)
   => fact 3  --> 3*fact (3-1)
>> 5* (4* (3*fact (3-1)))
   => 5* (4* (3*fact (3-1)))  --> 5*4* (3*fact (3-1))
>> 5*4* (3*fact (3-1))
   => 5*4  --> 20
>> 20* (3*fact (3-1))
   => 3-1  --> 2
>> 20* (3*fact 2)
   => fact 2  --> 2*fact (2-1)
>> 20* (3* (2*fact (2-1)))
   => 20* (3* (2*fact (2-1)))  --> 20*3* (2*fact (2-1))
>> 20*3* (2*fact (2-1))
   => 20*3  --> 60
>> 60* (2*fact (2-1))
   => 2-1  --> 1
>> 60* (2*fact 1)
   => fact 1  --> 1
>> 60* (2*1)
   => 2*1  --> 2
>> 60*2
   => 60*2  --> 120
>> 120
```

図8：階乗計算プログラムと実行例

```
main :-  {
          4.6237 * a + 2.6914 * b - 3.7517 * c =  1.4023;
         -2.4037 * a + 1.0432 * b + 0.7589 * c =  0.3724;
          1.0462 * a + 2.0495 * b + 6.3524 * c = -2.4728;
         a, b, c
         }
                          ▽
              -0.188747, 0.23866, -0.435184
```

図9：連立方程式求解プログラムと実行例

4.2　実行例

(A) 階乗計算

最初に、実行例を示す。以下の規則は数値の階乗を求めるプログラムfactである。ユーザーは、この規則とシステムで提供されているライブラリを用いて、数値4の階乗を求める。

“factorial”.　　**fact 1 :- {1}.**
#include beep.　　**fact a@constant :- {a * fact (a-1)}.**
#op(fact , prefix , 900).　　**main :- {fact 4}.**

(1) システムを起動する。

```
PROMPT% ./cpl <CR>                                  ---システムの起動
% loading file /home/nagai/src/prolog/cpl
% cpl loaded in module user, 0.217 sec 142,896 bytes
CPL on Prolog Languages 0.02 #0:  Tue Apr 16 13:48:18 1996.
[ new working directory is "~/src/prolog/example/"]
[CPL]-> _
```

(2) 既存のプログラム“fact.axm”を読み込む。

```
[CPL]-> read fact <CR>                              ---ファイル入力の指示
<< read data from a file >>
( factorial )
[ 191 data read from " /src/prolog/example/fact.trs".  ]
[CPL]-> _
```

(3) 既存のプログラム“fact.axm”を読み込む。

```
[CPL]-> list <CR>                                    ...状態表示の指示
<< types and rules >>
"factorial"                                          ...タイトルの指示
types:                                               ...タイプ情報の指示
   (type) sterm
   (type) numvar supertype sterm
                :
rules:                                               ...書換え規則の指示
   r1:  true;a :- {a}
                :
   r78:  fact 1 :- {1}
   r79:  fact a@constant :- {a*fact(a-1)}
   r80:  main :- {fact 4}
[CPL]-> _
```

(4) 書換え手続きにより4の階乗を求める。

```
[CPL]-> reduce auto <CR>                             ...書換え手続き起動の指示
<< reduce a term (in automatic mode) >>
Term.  <cr>          ; term to be reduced            ...入力ガイダンスの表示
exit.  <cr>          ; for exit
Term> main.  <CR>                                    ...サブジェクトの入力

Type:                                                ...タイプ空間の表示



main
   -*-> 24                                           ...書換え結果の表示
[ time used 66 msec.  ]  Term>                       ...書換え手続き終了の指示
exit.  <CR>
[ exit REDUCE ]
[CPL]-> _
```

以上により、数値4 の階乗24 を求めることができた。

(5) システムを停止する。

```
[CPL]-> quit <CR>                                    ...システムの停止要求
[ exit CPL ]
[ See you again !!!  ]
PROMPT% _
```

（B）連立方程式

書換え手続きの実行例を示す。規則ライブラリとしてシステムで提供されているライブラリを用いて、以下の連立方程式に出現する変数a, b, c の解を求める。

```
main :- { 4.6237 * a + 2.6914 * b − 3.7517 * c= 1.4023;
          -2.4037 * a + 1.0432 * b + 0.7589 * c= 0.3724;
           1.0462 * a + 2.0495 * b + 6.3524 * c=-2.4728;
           a, b, c }
```

(1) ファイル“simul.axm”から例題を読み込む。

```
[CPL]-> read simul
<< read data from a file >>
[ 89 data inserted from "~/src/prolog/example/bops.axm". ]
[ 44 data inserted from "~/src/prolog/example/beep.axm". ]
[ 2 data read from "~/src/prolog/example/simul.axm". ]
```

(2) プログラムを表示する。

```
[CPL]-> list
<< types and rules >>
    "length of banana 1"
types:
  (type) sterm
  (type) numvar supertype sterm
  (prim) constant supertype sterm
                  :
rules:
  r1: true;a :- {a}
  r2: (a;b);c :- {a;b;c}
  r3: a&b;c :- {a;b;c}
                  :
  r76: c@constant**k@constant++rest :- {c*k+rest}
  r77: c@constant**lx@linearop++rest :- {c*lx+rest}
  r78: main :- {a::aNumber;b::aNumber;c::aNumber;
                4.6237*a+2.6914*b-3.7517*c=1.4023;
               -2.4037*a+1.0432*b+0.7589*c=0.3724;
                1.0462*a+2.0495*b+6.3524*c= -2.4728;
               a,b,c}
```

(3) 自動実行による書換え手続きを起動する。

```
[CPL]-> reduce
<< reduce a term (in automatic mode) >>
   Term. <cr>                          ; term to be reduced
   exit. <cr>                          ; for exit
Term>
```

(4) 書換え手続きを開始する。

```
Term> main.
   Type: [(a@numvar),(b@numvar),(c@numvar)]
   main
   -*-> -0.188747,0.23866,-0.435184
[ time used 3.217 sec. ]
```

(5) 書換え手続きを終了する。

```
Term> e.
[ exit REDUCE ]
[CPL]->
```

以上により、変数a, b, c の解 −0.188747, 0.23866, −0.435184を求めることができた。

5 おわりに

本論文では, 制約プログラミング言語で用いられている制約充足手法と呼ばれる問題解決方法(局所伝播法、緩和法、グラフ変換ならびに等式解消法) の取り扱いにおける問題点に対応するために, 項書き換え操作を拡張した問題解決機構ならびにこれを組み込んだ制約プログラミング言語システムの実装方式について検討した。基本となる推論機構に項書換え手続きをベースとすることにより、あらゆる問題が取扱い可能となる。ここでは, いくつかの適用問題とその実行例について説明することで, 提案する方式の有効性を示した。

参考文献

［1］相場亮、古川康一：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―、人工知能学会誌、Vol.6, No.1, pp.47-59, January（1991）.

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

［3］P.V. Hentenryck : Constraint Satisfaction in Logic Programming, Logic Programming Series, The MIT Press（1989）.

［4］井田哲雄：計算モデルの基礎理論、岩波講座　ソフトウエア科学12, 岩波書店（1991）.

［5］J.-L. Lauriere : A Language and a Program for Stating and Solving Combinatorial Problems, Artificial Intelligence 10, pp.29-127（1978）.

［6］H.-W. Guesgen : CONSAT A System for Constraint Satisfaction, GMD PhD. Thesis, November（1987）.

［7］W. Leler : Constraint Programming Languages Their Specification and Generation,Addison-Wesley Pub. Comp.（1988）.

［8］J. Jaffar and J.-L. Lassez : Constraint Logic Programming, Proc. of Conf. on Principle of Programming Languages（1987）.

［9］J. Jaffar et al. : The CLP（R）Language and System, CMU Technical Report CMU-CS-90-181, Oct.（1990）.

［10］D. Sciamma, J. Gay and A. Guillaud : CHARME A Constraint Oriented Approach to Scheduling And Resource Allocation, pp.71-76, Procs. of Pacific Rim Int’l Conf. on Artificial Intelligence ’90（1990）.

［11］P. Lim, M.L. Epstein and E.H. Freeman : A Constraint Logic Programming Language for Combinatorial Optimization and Linear Programming, Procs. of CAIA ’90, pp.299-306（1990）.

［12］D. Giuse : KR: Constraint-Based Knowledge Representation, CMU Technical Report CMU-CS-89-142, April（1989）.

［13］A. Borning : The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, ACM trans. on Programming Languages and Systems, Vol.3, No.4, pp.353-387（1991）.

［14］I. Sutherland : SKETCHPAD: A Man-Machine Graphical Communication System, IFIPS Procs. of the Spring Joint Computer Conf.（1963）.

［15］J. Cohen : Constraint Logic Programming Languages, Communications of the ACM,Vol.33, No.7, July（1990）.

［16］G.J. Sussmann and G.L. Steele : CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions, Artificial Intelligence 14, pp.1-39（1980）.

［17］S. Wolfram : Mathematica A System for Doing Mathematics by Computer Second edition, Addison-Wesley（1991）.

［18］S. Minton, M.D. Johnson, A.B. Philips, and P. Laird : Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method, Procs. of AAAI-90, pp.17-24（1990）.

［19］三井東圧EQM研究会：パソコンのための方程式解法ソフトEQUATRAN-M入門、（財）省エネルギーセンター（1987）.

［20］B. Horn: Siri: A Constrained-Object Language for Reactive Program Implementation,Carnegie Mellon University Technical Report CMU-CS-91-152（1991）.

［21］M. Konopasek and S. Jayaraman: Constraint and Declarative Languages for Engineeering Applications: The TK!Solver Contribution, Procs. of the IEEE, Vol.73, No.12, pp.1791-1806（1985）.

［22］D.Giuse : KR:Constraint-Based Knowlede Representation, The Garnet Reference Manuals Revised for Version 2.0, CMU-CS-90-117-R2, pp.73-110（1992）.

［23］A. Colmerauer: An introduction to Prolog-III, Communications of the ACM, Vol.33, No.7, pp.69-90, July（1990）.

［24］K. Marriott and P. J. Stuckey : Programming with Constraints: An Introduction, The MIT Press（1998）.

［25］J. V. Leeuwen（Editor）: Handbook of Theoretical Computer Science, Elseveir Science Publishers（1990）.