自 然 言 語 処 理 101－2
（1994．5．27）

Prologプログラム変換に基づく否定情報の扱える制約システム

森 辰則　　中川 裕志
横浜国立大学 工学部 電子情報工学科
E-mail: mori@forest.dnj.ynu.ac.jp, nakagawa@naklab.dnj.ynu.ac.jp

梗概

自然言語処理において様々な場面で利用できる制約システムの一つにPrologプログラム変換に基づく論理型制約解消システムがある．この枠組は非常に強力な記述力を持つが，Prologプログラム変換は副作用を持たないプログラムを対象にしているため，カットオペレータを必要とするNegation as Failureによる否定も表現することはできない．しかし，言語情報を簡潔に記述できることのみならず，否定が本質的に必要になる場合があることからも否定記述の必要性が指摘されている．そこで，我々は全称限量可能なnot-equalをプログラム変換の枠組に沿う形で実現することによりある程度の否定知識を表現できるように拡張した．

Constraint solver based on the Prolog program transformation with negative constraints

Tatsunori MORI and Hiroshi NAKAGAWA
Division of Electrical and Computer Engineering,
Faculty of Engineering, Yokohama National University
Tokiwadai, Hodogaya-ku, Yokohama 240, Japan
E-mail: mori@forest.dnj.ynu.ac.jp, nakagawa@naklab.dnj.ynu.ac.jp

Abstract

Although the logic-based constraint solver based on the Prolog program transformation has strong expressive power, we cannot use negative expressions, which are usually represented by the 'negation as failure', because the Prolog program transformation can treat only programs without side effects. However, it is pointed out that negative expressions are not only useful for making representations be concise, but also essential in some cases. Accordingly, we introduce the goal 'not-equal', which can be quantified by universal quantifiers, in a suitable way for the Prolog program translation. With the goal, we can represent negative information to a certain extent.

1 はじめに

自然言語処理において，様々な情報が制約として記述される．様々な文法理論では，数多くの制約の集合により言語現象を捉えている(Sells 1985)．例えば，JPSG(Gunji 1987)などの単一化文法(Shieber 1986)では，文の文法性を構文木における局所分岐が持つ素性に対する制約として表現している．また，各種語用論的性質も制約として記述されるものが多い(森 1994)．

制約の表現するものが構造的な一致であったり，有限領域で数え上げできるものであるならば，単一化や選言形式で制約を表現すれば良い．実際，統語的な制約などでは選言形式が用いられることが多く，素性構造においていかに効率良く選言を表現するかが議論されている(小暮 1993b)．しかし，語用論的な制約を含めてすべての制約が常に選言形式に展開できるわけではない．例えば，JPSGにおける下位範疇化素性原理では集合演算が必要になる．

このような問題点を克服する強力な記述力をもつ制約システムとしては，Prologプログラム変換に基づく論理型制約解消システムがある．この枠組では，制約に含まれている情報が失われないように，制約すなわちPrologゴール列は手続きとして一度に実行されるのではなく，プログラム変換により等価ではあるがより簡単な標準形へ書換えられる．この種のシステムとしては，橋田らの条件付単一化(橋田 1986; Hashida 1986)をPrologに組み込んだシステムであるcu-Prologがある(Tsuda 1992)．

Prologプログラム変換は副作用を持たないPure Prologを対象にしている．このため，カットオペレータを含むプログラムはプログラム変換の対象とはならず，同時に，Negation as Failureによる否定も表現することはできない．この制限によりcu-Prologでも否定知識は扱えない．しかし，言語情報を簡潔に記述できることのみならず，否定知識が本質的に必要になる場合が存在することからも否定的素性記述の必要性が指摘されている(小暮 1993a; Kogure 1992)．そこで，我々は全称限量可能なnot-equalをプログラム変換の枠組に沿う形で実現することにより，否定技法(佐藤 1987)を使ってある程度の否定知識を表現できるように拡張した．

また，制約変換のアルゴリズムもcu-Prologと異なるものを用いている．cu-Prologでは制約の標準形としてモジュラー性を用いている．あるゴール列がモジュラーであるとはゴール列中のすべてのゴールの間に依存関係がないことである．制約がモジュラーでないゴール列になった場合に，等価でモジュラーな制約に変換される．しかし，すべてのPrologのゴール列が等価でモジュラーな制約に変換できるわけではなく，使用できるプログラムのクラスはある程度限定される．そこで，我々は，「ゴール列の一意展開」の概念により制約の簡単化を行なう方式を導入した．これにより，制約を記述するプログラムのクラスに制限が無くなる[1]．

2 Prologプログラム変換に基づく制約解消系

制約系の問題領域として自然言語処理など記号処理を考える場合，問題領域の対象として項を考えれば十分である．Prologは基本的にHerbrand領域上で宣言的プログラムが可能であり，その意味ではHerbrand領域上の制約解消系となりうるが，Prologプログラムが手続き的に解釈されている点が問題である．すなわち，実行順序に対して制限があるのにもかかわらず，通常の単一化では，項の構造中の不定部分に情報を持たせることが出来ないため，述語の引数の入出力に注意しないと，効率が悪くなったり，最悪の場合無限ループに陥る．PrologIIにおける遅延実行用述語**freeze**はこの問題に対して一つの手立てをあたえてくれるが，受動的な制約解消であるために問題は多い．

これらの問題を解決した制約解消系として，Prologプログラム変換に基づく論理型制約解消系がある．この枠組では，制約すなわちPrologゴール列は手続きとして一度に実行されるのではなく，プログラム変換により等価ではあるがより簡単な標準形へ書換えられる．この方法は，表1の対応のように，制約をPure Prologのゴールとして記述するため，Pure Prologと同等の記述能力がある．なお，Prologプログラム変換については付録Aを参照されたい．

表1: Prologプログラム変換に基づく制約解消系

制約を受ける対象	項．特に，変数および素性構造
制約の記述	Pure Prologのプログラムの記述
対象への制約の適用	対象となる項を含むゴール列
制約の解消	上記ゴール列の等価変換

また，自然言語処理では素性構造をデータとして扱うことが多い(小暮 1993a; 小暮 1993b)．単一化器を素性構造を扱えるものに拡張すれば，この制約解消系の項として素性構造を扱えるようにできる[2]．

以上の点を実現したシステムとしては，橋田らの条件付単一化(Hashida 1986)をPrologに組み込んだシステムであるcu-Prologがある(Tsuda 1992)．

上記の制約解消系を用いるシステムにおいては，各データは素性構造を含めた項`T`に，その項に対する制約を表すPrologのゴール列`C`を付随させた，制約付き項`T|C`により表現される[3]．制約付き項は，選言的素性記述を扱うための効率の良い方式であるMaxwellとKaplanその他の文脈付き制約記述とほぼ対応している．文脈付き制約記述では$\phi \vee \psi$を文脈を表す論理式

[1] もちろん，もとのプログラム自身が実行不可能な場合は除く．

[2] ここでは，従来の項と素性構造を合わせて，単に項と呼ぶ．

[3] 項と制約は，変数や素性構造のラベルを通じて結びつけられているので，独立させた状態で扱うことが多い．この表現は説明の便宜上のものである．

pを用いて，$(p \rightarrow \phi) \wedge (\neg p \rightarrow \psi)$と表す．よって，同様の手法により，互いに排他的な制約により選言を効率良く表現できる(Tsuda 1992)．異なる点は文脈としてPrologのプログラムが直接記述できるので，非常に強力な点である．

制約付き項に対する処理は単一化によって行なわれる．制約付き項の単一化器 unify は以下のように定義される．

unify($T_1|C_1$, $T_2|C_2$, $T_3|C_3$) $\Leftrightarrow$
funify(T_1,T_2,T_3) $\wedge$ append(C_1,C_2,C) $\wedge$ cunify(C,C_3)
ただし，funify/3は第1引数の項と第2引数の項を単一化したものが第3引数であるという関係，cunify/2は第1引数のゴール列を等価変換して簡略化したものが第2引数であるという関係を表す．

さて，この制約解消系の主要部分は，上記のcunify/2である．この部分でどのような処理を行なうかによって，制約として扱えるPrologプログラムのクラスが変わる．橋田らは，(橋田 1986; Hashida 1986)において，以下のモジュラーと呼ばれる標準形を基本にして，モジュラーでないゴール列をモジュラーなゴール列に変換する方式を提案している．

- モジュラー
  ゴール列$C_1, C_2, \ldots, C_m$は以下の全ての条件を満たす時，モジュラーであるという．
  1. C_iの全ての引数は変数$(1 \leq i \leq m)$である．
  2. どの変数も二個所以上の空虚でない引数位置には現れない．
  3. C_iはモジュラー定義述語から成る$(1 \leq i \leq m)$．
- 空虚な引数位置
  述語pの全ての定義節においてpの第i引数が変数である時，pの第i引数は空虚であるという．
- モジュラー述語
  述語pの全ての定義節の本体が，モジュラーまたはnilの場合，pはモジュラー述語であるという．

モジュラー化ができることを保証するためにcu-Prologでは，制約としてPure Prologのうちモジュラー述語もしくはモジュラーなプログラムに変換可能な述語のみが使用できる．

3 否定情報の取り扱い

文献(Tsuda *et al.* 1989)にも述べられているように，現在のcu-Prologでは，notなどの否定に関するオペレータを扱うことはできない．これは，カットオペレータを含む副作用のあるゴールをPrologプログラム変換では扱えないからである．そのため，失敗による否定(NAF: negation as failure)(Shepherdson 1984; Lloyd 1987)を直接実現することはできない．

しかし，内部変数を含まない述語の場合は，その定義からその否定を表す述語の定義をNAFを含まずに求める手続きが得られている．これは否定技法と呼ばれる(佐藤 1987)．例えば，この手続きにより，

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

から，

```
not_member(X, L) :- ∀ H,T.(L \= [H|T]).
not_member(X, [H|T]) :-
    X \= H, not_member(X,T).
```

が得られる．ここで，not-equal (単一化できない)を表す'\='および'∀'で限量された変数が入っていることに注意されたい[4]．このプログラムの示すように，否定技法による変換結果には，否定を表すnot-equalが最後まで残ってしまう．not-equalは二つの項が単一化できないことを表すが，その真偽は「それぞれの項が単一化できるときに失敗する」と定義することはできない．なぜならば，未束縛の変数がそのゴールの出現よりも後で起こる可能性があるからである．これを扱う一つの方法は，二つの項を常に監視して単一化できないか，あるいは，同一な項になった場合にその結果を反映させるというものである．しかし，この方法はプログラム変換に基づくシステムでは扱いづらく，また，二つの項の単一化を繰り返すために同じ部分の単一化を複数回行なうので効率が悪い．

そこで，本システムでは，not-equalを持つ個々のゴールに対して，動的にプログラム節を生成し否定情報を扱うこととした．この方法では，プログラム変換の枠組に自然に取り込め，以下に述べるように定数部分において一度単一化できたものはそれ以降では考慮しないので効率が良い．さらに，プログラム節を排他的に構成することにより Unfoldingを行なった時にこのゴールに対する値の束縛が起こりジェネレータになりうるので，候補の絞り込みが早くなることが期待される．また，この枠組の上で，∀H,T(L \= [H|T]) のように，全称限量された変数を含む not-equal ゴールも扱える．本システムでは，全称限量を表現するゴールを特別に用意している．例えば，∀H,T(L \= [H|T]) に相当するゴールは forall([H,T],L \= [H|T]) と記述する．

以下にプログラム節の動的生成法を述べる．これは単一化アルゴリズム(Lloyd 1987)により得られた単一化子を用いて，動的に \= のプログラム節を生成する．Unfolding操作においては，この動的に生成されたプログラム節を用いる．全称限量されていない α \= β というゴールは，全称限量付きの場合の一例であるから，以下では，forall($[bv_1, \ldots, bv_n]$,α \= β) なるゴールについて考える．まず，$\{\alpha,\beta\}$ の単一化子θを単一化アルゴリズムにより求める．ただし，素性構

[4] Prologの内部変数は体部において'∃'で限量されている．

造を含む場合は素性構造自身も変数と同等に扱い，単一化子の表す代入の対象となるものと考える．

単一化子が存在しない時は，α \= β 自身が成功するので，

```
α\= β :- true.
```

というプログラム節が存在すると考えれば良い．

一方，$\theta = \{v_1/t_1, \ldots, v_m/t_m\}$ なる単一化子が得られた場合は以下のようにする．ここで，v_j は変数，t_j は項である．この単一化子が表す代入が不可能な時が α \= β が真になる時であることに注意すれば，

α \= β

$\Leftrightarrow$

v_1 \= $t_1 \vee v_2$ \= $t_2 \vee \ldots \vee v_m$ \= t_m

$\Leftrightarrow$

v_1 \= t_1

$\vee(v_1 = t_1 \wedge v_2$ \= $t_2)$

$\vee(v_1 = t_1 \wedge v_2 = t_2 \wedge v_3$ \= $t_3)$

$\vdots$

$\vee(v_1 = t_1 \wedge \ldots \wedge v_{m-1} = t_{m-1} \wedge v_m$ \= $t_m)$

が得られる．二番目の変形は選言の候補を互いに排他的にするためである．このとき，α ならびに β の構造において，最も内側の変数から順に $v_1, v_2, \ldots$ に対応させると，ある素性構造に関する \= が現われるときには，内側の変数には何らかの束縛が起こるため，全称限量のスコープが複数の選言に跨ることがなくなる．

ここで，変数 v_i のうち全称限量されているものに関しては，上記の v_l \= t_l は $\forall v_l(v_l$ \= $t_l)$ となるが，これは恒偽であるから，この項は上の式から取り除く．このようにして得られた α \= β の変形結果は，次のプログラム節と (閉世界仮説の下で) 等価であることがわかる．

```
forall([bv_1,...,bv_n],α \= β) :-
        forall([bv_a,...],v_1 \= t_1).
(forall([bv_1,...,bv_n],α \= β) :-
        forall([bv_b,...],v_2 \= t_2))θ_2.
        ⋮
(forall([bv_1,...,bv_n],α \= β) :-
        forall([bv_z,...],v_m \= t_m))θ_m.
```

ただし，θ_j は，$\{v_1/t_1, \ldots, v_{j-1}/t_{j-1}\}$ なる代入であり，各体部の全称限量変数は体部に生じているものだけとし，空 [] の場合は，forall(...) を外す．各プログラム節の体部は排他的になっているので，上記のプログラムによる展開と，forall([$bv_1, \ldots, bv_n$],α \= β) の実行は等価になっている．

θ がただ一つの代入 v/t しか持たず，v, t がそれぞれ，α,β の場合は上記の方法で得られるプログラム節が，トートロジとなる．この時だけはそのゴールの展開 (Unfolding) ができない例外として扱う．つまり，この例外が変数評価の遅延に相当する．これはその制約の真偽値が決まらない unknown を許す立場で，評価が可能になる時点まで遅延するということ意味を表す．しかし，これは，遅延という消極的な意味だけではなく，後に述べるように，否定的な表現の場合，その制約が残っていること自身に意味があることもある．

ここで，幾つかの例をみてみよう．例えば，f(a, g(X))\=f(Z, g(b)) というゴールについては，次のプログラム節が得られ，これにより展開できる．

```
f(a, g(X)) \= f(Z, g(b)) :- a \= Z.
f(a, g(X)) \= f(a, g(b)) :- X \= b.
```

Z もしくは X の値が決まらないと，a \= Z もしくは X \= b は展開できないので，この時点での部分実行は停止するが，後に値が束縛されれば，それぞれのゴールは展開可能な状態になり，展開が進むことになる．ここで，二番目の節において，変数 Z が a に束縛されている点に注意されたい．これにより not-equal ゴールから他のゴール，主に他の not-equal との相互作用により簡略化が進むことが期待される．

素性構造の例を以下に挙げる．本システムにおける素性構造の表記については付録 B を参照されたい．動詞 'eat' は三人称・単数以外の場合に使用できるということを制約を用いて辞書項目に書くとすると，次のようになる．

```
[cat: vp,
 word: eat,
 agr:A#[number:N,person:P],
 subj:[agr:A]]
  | [forall([B], A \= B#[number: sg,
                         person: third])]
```

ここで，素性構造に対する全称限量の意味を定義しておく．素性構造は包摂関係 $\leq$[5] に関して束構造を構成する．ある素性構造は，それを top とする部分束上に位置するすべての素性構造と単一化可能であり，その素性構造群を代表するものとして位置付けられる．そこで，素性構造が全称限量されたときには，エルブラン領域を定義域とするのではなく，それが代表する素性構造群の集合を定義域とする全称限量として考える．すなわち，以下のようになる．

```
forall([B], A \= B#[number: sg,
                    person: third]) ⇔
  ∀ B ≤ [number:sg, person:third].(A \= B)
```

さて，先ほどの 'eat' における制約は先ほどと同じ手法で展開できて，以下のプログラム節が得られる．

```
forall([B], A#[number:N,person:P]
            \= B#[number:sg, person:third]) :-
      N \= sg.
forall([B], A#[number:sg, person:P]
            \= B#[number:sg, person:third]) :-
      P \= third.
```

[5] 情報の包含関係を表す半順序関係で，例えば，[a:v,b:w] $\leq$ [a:v] などが成り立つ．

ここで,

```
forall([B],A#[number:sg,person:third]
            \=B#[number:sg,person:third]) :-
    forall([B], A#[number:sg,person:third]
                \= B#[number:sg,person:third]).
```

は展開結果の一つの節になるのであるが，先ほどの全称限量の定義から体部は恒偽であるので削除される．

以上述べてきた方法を使うと，様々な否定的な表現をすることができる．以下に例を示す．

3.1 値の否定

素性がある値をとらないことを表す場合である．これは，通常の項ならば限量なしの not-equal 表現でできる．また，素性構造の場合は全称限量付きの not-equal により表現できる．先ほどの 'eat' の定義例がそれである．

3.2 値の同一性の否定

二つの素性の値が共参照しない (トークン同一でない) ことを表す場合である．素性構造を含むすべての項において，限量なしの not-equal で表現できる．例えば，「目的語が再帰代名詞でない限り，主語と目的語の指示対象が違う」という情報は以下のように表される．

```
[subj:[reference:X],
 obj:[reflective:-,reference:Y]]
   | [X \= Y]
```

3.3 ある素性の存在の否定

ある素性構造がある特定の素性を持たないことを表す場合である．これは，二つの全称限量変数を持つ not-equal で表現できる．例えば，素性構造 A が a という素性は持つが，b という素性は持たないということは以下のように記述される．

```
A#[a:X] | [forall([B,C#[b:B]],A\=C)]
```

制約が満足されている間はこの制約が解消されることはなく，常に制約としてのみ働く．

4 変換戦略

cu-Prolog におけるモジュラー化に基づく制約解消にこの not-equal を導入する場合には，モジュラー化の際の Unfolding するゴールの選択において，展開できないゴールを候補から外すようにすればよいであろう．ただし，not-equal 以外のプログラム節がモジュラーであったとしても，それによるゴールと not-equal を含むゴール列が常にモジュラー化可能であるかどうかは，調べる必要がある．

我々は，モジュラー化とは別の変換戦略を用いて，制約の簡単化を行なっている．モジュラー化方式では使用できるプログラムのクラスが限定されるため，利用者が常にプログラム記述に注意を払う必要があるからである．我々の変換戦略は以下に述べる「一意展開」の考え方に基づいている．制約を表すゴール列の解が複数存在する場合には，まだその制約を解くだけの情報が集まっていないことを表す．一方，制約を表すゴール列の解が一つだけしか存在しない場合には，その制約を解くのに十分な情報が集まっていることを表し，簡単化できる．この判断をゴール列全体に対して行なうことはゴール列全体を展開することに他ならず，Prolog インタプリタの動作そのものである．しかし，この判断を，プログラムの完全なる実行ではなく，プログラム変換の概念により一部分に適用することもできる．つまり，ゴール列全体にではなく，その一部からなるゴール列に限定し，さらに，その部分ゴール列の完全な実行ではなく，数ステップの推論に限定した部分実行に限定して考えることができる．

まず，上記ゴール列をプログラム変換で考えるに当たって，そのゴール列を体部に持ち，そのゴール列に現われるすべての変数を頭部の引数として持つプログラム節 (定義節) を新たに定義する．そのプログラム節の定義の下では，そのゴール列を制約とすることと，上記プログラム節の頭部と同じゴールを唯一持つ新たなゴール列を制約とすることとは等価である．よって，ゴール列をその新しいゴール列におきかえ，定義したプログラム節に対してプログラム変換を適用することは，元のゴール列に対して変換操作を施していることに他ならない．

このプログラム節の下，先ほどの部分ゴール列に対する部分実行における判断は次のようになる．プログラム節の体部のゴールの一部に対して，各々の Unfolding 操作がプログラム節の数を 2 以上にしない時には，曖昧性を増すことなく簡単化できる．このような Unfolding 操作による展開を「一意展開」と呼ぶ．

例えば，`p(a,X) :- q(X). p(b,X) :- r(X). s(b,X) :- q(X). s(c,X) :- r(X). q(1). q(m). q(n). r(m). r(n).` に対して，制約 `[p(a,A)]` は，`c1(A) :- p(a,A).` なる `c1/1` により，`[c1(A)]` と置き換えられるが，`p(a,A)` と単一化可能な頭部を持つプログラム節は 1 通りなので，一意展開できて，`c1(A) :- q(A).` である．また，同様に，`[p(A,B),s(A,B)]` では，`c2(A,B) :- p(A,B),s(A,B).` という新述語に対して，`p(b,X),s(b,X)` をそれぞれ頭部に持つプログラム節の組あわせだけが展開可能なので，`c2(b,B) :- r(B),q(B).` となる．

上記のように，一意展開可能かどうかの判定は，基本的には，実際の Unfolding を行なわなくとも，頭部の情報だけで行なえる．しかし，各々の節の適用条件が頭部に現われていないプログラムではこの判定だけでは不十分である．特に，`\=` を用いて節の適用条件を記述する場合は，その条件は頭部に現われえない．そ

こで，Unfoldingを実際に1回行ない，その展開されたプログラム節の下で別のゴールの頭部による判定を行なう．また，展開できない場合でも，あるゴールと単一化可能な頭部を一般化したものをゴールに単一化させ，候補の絞り込みを行なう．

一意展開の過程はプログラム節からゴールへの値の束縛が主であるが，これ以外に，ゴール節からプログラム節の頭部への値の束縛も考慮すべきである．その束縛により解の候補が絞り込まれる可能性があるからである．

よって，変換戦略は以下の手順で行なわれ，新しい制約はステップ1における定義節の頭部である．

1. 制約列を体部に持つ定義節を定義する．
2. その体部の各々のゴールについて，そのゴールと単一化可能なプログラム節の頭部を一般化したものを単一化させる．
3. 体部の各々のゴールについて，そのゴールからプログラム節の頭部への値の束縛がある場合には
   (a) その値で特殊化したプログラム節を新たに作成する[6]．
   (b) そのプログラム節の各々の節について本変換戦略を再帰的に適用する．
4. 一意展開を行なう．

上記過程の中で，現在のところステップ3においてゴールがプログラム節の頭部を特殊化する場合はすべて，ステップ3a，3bを行なっている．しかし，これは一部の制約記述において，上記変換過程が停止しないことがわかっている．例えば，append(X,X,Y)などでは，値の伝播が無限につづいてしまう．これに対しては，現在のところ，探索の深さによる制御を行なっているが，本質的には別の制御が必要があろう．

5 否定表現に関連する研究

我々の方法は論理プログラムの枠組により制約を扱っているので，単に素性構造上で否定知識を扱える以上の記述能力があるが，素性構造に対する否定的な制約の記述力だけをみても，十分な記述力がある．素性構造における否定表現という観点から関連研究を幾つか述べる．

素性構造は部分情報を表すので，否定を表す記述を任意の時点で解釈できないという問題がある．これを扱う方法としては，主に以下の2通りがある．

- 否定記述を正しく解釈できる時点まで解釈を遅延する．
- 否定記述を制約として扱い，制約を付加した素性構造自身を扱う．

[6]新述語の作成ならびにUnfolding,Folding操作により得られる．

両者は完全に分離されているわけではなく，前者において遅延された否定記述を特に制約と見れば，後者となる．その意味において，我々の方法は，中間の立場に相当する．つまり，十分評価できない情報を制約として保存している．

Dawar and Vijay-Shanker (1989)は Rounds-Kasper Logicによる素性構造記述式[7]に否定記号を付加し，記述式の解釈の値として真，偽の他に未定を採り入れることにより，否定記述の評価を遅延させている．しかし，遅延された部分はそのまま保存されるので，我々の方式とは異なり，増進的ではない．

Smolka and Treinen (to appear)の方法も一種の遅延方式である．この方法は，彼等独自の素性記述式に全称限量付きのnot equalを加えている点において我々の方式と似ている．彼らの方法は素性記述を幾つかの基礎的な素性構造操作に分解し操作的に解釈する．すべての操作が成功した場合に素性記述が満足されるとする．否定の付加された素性記述の場合は，現在の状態をスタックに保存し，その否定内の素性記述を解釈する．その解釈の最中に単一化が変数への代入や新たな素性の設定など素性構造への破壊的な操作とならない場合は，そのまま否定記述は評価可能である．一方，破壊的な操作となる場合にはそれら破壊的操作となる部分の(部分的)単一化の履歴をすべて保存しておく．後に，その破壊的操作の対象となった素性に代入などが行なわれた場合には，保存しておいた部分的単一化を再評価する．否定に関する情報は，単一化が破壊的操作にならなくなるまで，保存され，繰り返し，再評価される．この方法では，破壊的操作の部分だけを保存し，それだけを再評価するために，増進的に評価が行なわれる．また，破壊的操作となる単一化操作は，素性構造の単一化結果として得られる単一化子の個々の代入に相当するので保存される情報は我々の方式と同等である．しかし，この方法では，複数の否定記述に関して独立して扱われるため，否定記述同士の相互作用により否定記述が簡略化される場合が扱えない．例えば，

```
forall([A#[f1:a,f2:b]], [f1:X,f2:Y] \= A),
X \= a
```

は我々の方法では，一度の変換により，`X \= a`に縮退できる．

Kogure (1992)の方法は，制約を素性構造に与える方式である．この方法では，素性構造にその素性構造と同一にならない素性構造のリストと現われ得ない素性名のリストを付加することにより否定的制約を表現している．この方式は単一化時に効率良く否定制約を扱うことができる．しかし，われわれのプログラム変換の枠組にこの手法を導入することは可能であるが，制約として陽に全称量化子付きのnot equalを表現できないため，否定技法に用いることはできない．また，

[7]素性構造に対する制約として捉えることができる．以下，記述式と略記する．

ある素性名の素性は存在するが，その値は特定のある値以外である，ということを記述するためには，利用者が，そのための型を明示するか，個々のアトムの非同一性を表す制約の選言に明示的に展開し，選言の扱える単一化器を用意する必要がある．

6 おわりに

本稿では，自然言語処理など記号処理の分野において強力な記述力を発揮するPrologプログラム変換に基づく論理型制約解消システムに，全称限量可能なnot-equalをプログラム変換の枠組に沿う形で実現することにより，ある程度の否定知識を表現できるように拡張した．また，適用可能なプログラムのクラスに制限のない「ゴール列の一意展開」の概念により制約の簡単化を行なう方式を導入した．

本稿で述べたnot-equalの実現方法は，十分効率のよい方法であり，さまざまな場合を記述できるが，変換戦略に関しては議論が十分ではない．例えば，制約に変数を含まない場合に先の過程が常に正しい解を出すことは明らかであるが，変数への部分的な束縛があったときに，実際には解が一つしかないいかなる場合についても，先に述べた過程で正しく一意展開の解を見つけられるかどうかは，調査すべき事項である．また，一意展開においてその手続きの終了が一意展開の失敗時点であることから，若干のオーバヘッドが存在するという問題点がある．以上の点は今後の検討課題である．

参考文献

Dawar, Anuj and Vijay-Shanker, K. 1989. A three-valued interpretation of negation in feature structure descriptions. In *Proceedings of 27th Annual Meeting of the Association for Computational Linguistics*, pp. 18–24. ACL, June.

Gunji, Takao. 1987. *Japanese Phrase Structure Grammar*. Studies in Natural Language and Linguistic Theory. D.Reidel Publishing Company, Dordrecht, Holland.

Hashida, Koiti. 1986. Conditioned unification for natural language processing. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING 86)*, pp. 85–87.

Kogure, Kiyoshi. 1992. A treatment of negative descriptions of typed feature structures. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING 92)*, pp. 380–386, August.

Lloyd, J.W. 1987. 論理プログラミングの基礎．ソフトウェア サイエンス シリーズ．産業図書，東京．(邦訳: 佐藤，森下)．

Sells, Peter. 1985. *Lecture on Contemporary Syntactic Theories: An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammer, and Lexical-Functional Grammer*, Vol. 3 of *CSLI Lecture Notes*. CSLI.

Shepherdson, John C. 1984. Negation as failure: A comparison of clark's completed data base and reiter's closed world assumption. *The Journal of Logic Programming*, Vol. 1, No. 1, pp. 51–79.

Shieber, Stuart M. 1986. *An Introduction to Unification-Based Approaches to Grammer*, Vol. 4 of *CSLI Lecture Notes*. CSLI, Stanford, CA.

Smolka, Gert and Treinen, Ralf. (to appear). Records for logic programming. *Journal of Logic Programming*.

Tsuda, Hiroshi, Hashida, Koiti, and Sirai, Hidetosi. 1989. cu-Prolog and its application to JPSG parser. In *Proceedings of the Logic Programming Conference '89*, pp. 155–164.

Tsuda, Hiroshi. 1992. cu-Prolog for constraint-based grammar. In *Proceedings of the International Conference on Fifth Generation Conputer Systems '92*, pp. 347–356.

小暮潔. 1993a. 素性構造 (1). 人工知能学会誌, Vol. 8, No. 2, pp. 184–191, 3月.

小暮潔. 1993b. 素性構造 (2). 人工知能学会誌, Vol. 8, No. 3, pp. 305–311, 5月.

佐藤泰介, 玉木久夫. 1983. Prologに於けるプログラム変換. In *Proceedings of the Logic Programming Conference '83*. ICOT, 6月.

古川康一, 溝口文雄 (編). 1987. プログラム変換, 知識情報処理シリーズ, 第7巻, 第6章, pp. 103–119. 共立出版. (佐藤: 合成問題への新しいアプローチ).

橋田浩一, 白井英俊. 1986. 条件付単一化. コンピュータ・ソフトウェア, Vol. 3, No. 4, pp. 28–38, 10月.

松本裕治, 伝康晴, 宇津呂武仁. 1993. 構文解析システム SAX 使用説明書 version 2.0. 京都大学 工学部 長尾研究室, 奈良先端科学技術大学院大学 松本研究室, 4月.

森辰則, 中川裕志. 1994. 日本語マニュアル文からの知識抽出 — ゼロ代名詞照応問題を中心に —. 情報処理学会第48回 (平成6年前期) 全国大会講演論文集, pp. 3–177 – 3–178. 情報処理学会, 3月.

中川裕志, 中村直人. 1985. Prolog等価変換エディタと等価変換戦略. 情報処理学会論文誌, Vol. 26, No. 5, pp. 905–912.

A Prologプログラムの変換

文献(佐藤 1983; 中川 1985)を基に，Prologプログラムの等価変換についてfold/unfoldを中心に簡単に述べる．

最初のプログラムをS_0，変換過程で導入される定義節の集合の初期状態として$D_0 = \{\}$とする．いま，$\langle S_n, D_n\rangle$が与えられているとする．変換は，Definition(新しい述語の定義の導入)，Unfolding(ゴールのすべての可能な展開)，Folding(節のゴール列の新述語による置き換え)，Deletion (呼び出しが成功しない節の削除)，Merge (同一ゴールの併合)，Laws(特別な知識によるプログラムの等価変換)，Case Split (節の場合分け)，などの変換規則のうち，一つを選択し，$\langle S_{n+1}, D_{n+1}\rangle$を得る作業である．ここでは，Unfolding,Foldingについてその定義を述べる．

A.1 Unfolding

S_n中のある節

$$A_0 \text{ :- } A_1,\ldots,A_m.$$

および，その本体のゴールA_iを適当に選ぶ．A_iと単一化できる頭部を持つS_0中の節を，

$$\{B_1 \text{ :- } \beta_1,\ldots,B_k \text{ :- } \beta_k\}$$

とする[8]．A_iとB_jの最汎単一化子をΘ_jとすれば[9]，

$$\begin{aligned} S_{n+1} &= S_n - \{A_0 \text{ :- } A_1,\ldots,A_m.\} \\ &\quad \bigcup_{1\le j\le k} \{(A_0 \text{ :- } A_1,\ldots,A_{i-1},\beta_j,A_{i+1},\ldots,A_m)\Theta_j\} \\ D_{n+1} &= D_n \end{aligned}$$

S_nに付け加えられた各節を展開された節とよぶ．

A.2 Folding

S_nから，節A_0 :- $\alpha_1\ \alpha_2\ \alpha_3$ D_nから，定義節$P(\vec{X})$:- $\beta(\vec{X},\vec{Y})$.を適当に選ぶ．$\vec{X}$,$\vec{Y}$は変数群であり，$\vec{Y}$の各変数は，$\vec{X}$の各変数と相異なる．このとき，選択した節の適当なrenaming A_0' :- $\alpha_1'\ \alpha_2'\ \alpha_3'$ について，**Folding条件**

- A_0 :- $\alpha_1\ \alpha_2\ \alpha_3$は，展開された節である．
- ある代入Θが存在して，$\alpha_2' = \beta(\vec{X},\vec{Y})\Theta$．ただし，$\vec{Y}$の各変数$y_j$について，$y_j\Theta = y_j$，かつ，$P(\vec{X})\Theta$，$A_0'$，$\alpha_1'$，$\alpha_3'$中に$\vec{Y}$の変数が現れない．

を満たしているならば，

$$\begin{aligned} S_{n+1} &= S_n - \{A_0 \text{ :- } \alpha_1\ \alpha_2\ \alpha_3\} \\ &\quad \cup\{A_0' \text{ :- } \alpha_1'\ P(\vec{X})\Theta,\ \alpha_3'\} \\ D_{n+1} &= D_n \end{aligned}$$

とする．A_0' :- $\alpha_1'\ P(\vec{X})\Theta$, α_3'も展開された節である．

[8]ただし，各β_iはゴール列．

[9]変数の適当なrenamingは暗に仮定される．

B 素性構造の記述

本システムにおける素性構造は<素性構造>に示すようにマトリックス記法を用いている．これは，構文解析システムSAX(松本 1993)に付属している素性構造の単一化器で扱えるものと同じである．

```
<項> ::=  <変数> | <素性構造> | <複合項>
<変数> ::=  Prologの変数
<複合項> ::=  Prologの複合項
<素性構造> ::=  <タグ>#<素性リスト> | <素性リスト> | <タグ>
<タグ> ::=  Prologの変数
<素性リスト> ::=  <素性－素性値ペア>のリスト
<素性－素性値ペア> ::= <素性名>:<項>
<素性名> ::=  Prologのアトム
```

この中で，<素性リスト>が素性構造の実際のデータを表現する．<タグ>は共参照を表現するために用いる．例えば，人称・数の一致を表現するには，

```
[cat: vp,
 agr: X#[number: sg,
         person: third]
 subj:[agr:X]]
```