ICOT Technical Memorandum: TM-0754

TM-0754

cu-Prolog and its application to JPSG parser

by
H. Tsuda, K. Hashida & H. Shirai

July, 1989

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

cu-Prolog and its application to JPSG parser

TSUDA, Hiroshi *
E-mail: tsuda%icot.jp@relay.cs.net

HASIDA, Kôiti
E-mail: hasida%icot.jp@relay.cs.net
Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku Tokyo, 108 Japan

SIRAI, Hidetosi †
Chukyo University
101-2 Yagoto honcho, Showa-ku Nagoya, 466 Japan
E-mail: sirai%ninchi.chukyo-u.jp@relay.cs.net

概要

本論文[1]では，条件付単一化（制約単一化）[14, 6] に基づき我々が作成した新しい制約論理型プログラミング言語 cu-Prolog を紹介し，cu-Prolog の適当な応用として JPSG(日本語句構造文法) による簡単な日本語パーザを取り上げる．

従来の多くの制約論理型プログラミング言語が制約に代数方程式を取るのに対し，cu-Prolog はユーザ定義の述語による Prolog の項を制約とする．このため，自然言語処理，特に単一化文法 [9] の処理に必要な記号的，組合せ的な制約を宣言的に記述できる．cu-Prolog の構文は，通常のホーン節に制約を加えた制約付ホーン節 (CAHC: Constraint Added Horn Clause) から成り，制約解消系は unfold/fold 変換により実行される．

This paper presents our new constraint logic programming language *cu-Prolog* and introduces a simple Japanese parser based on Japanese Phrase Structure Grammar (*JPSG*) as a suitable application of cu-Prolog.

cu-Prolog adopts *constraint unification* instead of the normal Prolog unification. In cu-Prolog, constraints in terms of user defined predicates can be directly added to the program clauses. Such a clause is called *Constraint Added Horn Clause (CAHC)*. Unlike conventional CLP systems, cu-Prolog deals with constraints about symbolic or combinatorial objects. For natural language processing, such constraints are more important than those on numerical or boolean objects. In comparison with normal Prolog, cu-Prolog has

*本論文執筆当時，東京大学大学院理学系研究科情報科学専攻
†当時，玉川大学情報工学科
[1]なお，この論文は本年 4 月に行われた ACL European Chapter に受理された論文 [11] に加筆訂正したものである．

more descriptive power, and is more declarative. It enables a natural implementation of JPSG and other unification based grammar formalisms.

1 Introduction

Prolog is frequently used in implementing natural language parsers or generators based on unification based grammars. This is because Prolog is also based on unification, and therefore has a declarative feature. One important characteristic of unification based grammar is also a declarative grammar formalization [9].

However, Prolog does not have sufficient power of expressing constraints because it executes every parts of its programs as procedures and because every variable of Prolog can be instantiated with any objects. Hence, the constraints in unification based grammar are forced to be implemented not declaratively but procedurally.

We developed a new constraint logic programming language *cu-Prolog* which is free from this defect of traditional Prolog [10]. In cu-Prolog, user defined constraints can be directly added to a program clause (constraint added Horn clause), and the constraint unification [14, 6] [2] is adopted instead of the normal unification. This paper discusses the outline of the cu-Prolog system, and presents a Japanese parser based on JPSG (Japanese Phrase Structure Grammar) [5] as a suitable application of cu-Prolog.

2 Constraint Added Horn Clause (CAHC)

Most of the constraint logic programming language systems (CAL [13], PrologIII [4], etc.) deal with constraints about algebraic equations, i.e., constraints about numerical domains, such as that of real numbers etc.

However, in the problems arising in Artificial Intelligence, constraints on symbolic or combinatorial objects are far more important than those on numerical objects. cu-Prolog handles constraints described in terms of sequence of atomic formulas of Prolog. The program clauses of cu-Prolog are following type, which we call *Constraint Added Horn Clauses* (CAHCs):

[Def] 1 (CAHC) Constraint Added Horn Clause *consist of the following two types of clauses:*

$$\overbrace{H}^{head} :- \overbrace{B_1, B_2, \ldots, B_n}^{body}; \overbrace{C_1, C_2, \ldots, C_m}^{constraint}.$$

$$\overbrace{H}^{head} ; \overbrace{C_1, C_2, \ldots, C_m}^{constraint}.$$

H is a head and $B_1, B_2, \ldots, B_n$ composes a body like normal Prolog.
$C_1, C_2, \ldots, C_m$ comprise a set of the constraints (or null) on the variables occurring in the head or body. $C_1, C_2, \ldots, C_m$ must be ,in the current implementation, modular *in the sense that it has the following canonical form.*

Seen from the declarative semantics, these two clauses are equivalent to the following two program clauses of normal Prolog.

[2] In these earlier papers, "constraint unification" was called "conditioned unification."

1. $H :- B_1, B_2, \ldots, B_n, C_1, C_2, \ldots, C_m.$

2. $H :- C_1, C_2, \ldots, C_m.$

Of course, program clauses of normal Prolog are the special cases of CAHCs.

[Def] 2 (modular) *A sequence of atomic formulas $C_1, C_2, \ldots, C_m$ is* modular *when*

1. *every arguments of C_i is variable, and*
2. *no variable occurs in two distinct places, and*
3. *the predicate of C_i is modularly defined ($1 \leq i \leq m$).*

[Def] 3 (modularly defined) *Predicate p is* modularly defined, *when in every definition clause of p ($P :- \mathbf{D}.$),*

D *is empty,*

or

1. *every argument of* **D** *is variable,*
2. *no variable occurs in two distinct places, and*
3. *every predicate occurring in* **D** *is $p()$ or modularly defined.*

For example,

$member(X, Y), member(U, V)$ is modular,
$member(X, Y), member(Y, Z)$ is not modular, and
$append(X, Y, [a, b, c, d])$ is not modular.

3 cu-Prolog

3.1 Constraint Unification

cu-Prolog employs Constraint Unification [14, 6] which is the usual Prolog unification plus constraint transformation (normalization).

Using constraint unification, the inference rule of cu-Prolog is as follows:

$$\frac{A, \mathbf{K}; \mathbf{C}. \ , \ A' :- \mathrm{L}; \mathbf{D}., \quad \theta = mgu(A, A'), \mathbf{C}' = mf(\mathbf{C}\theta, \mathbf{D}\theta)}{\mathbf{L}\theta, \mathbf{K}\theta; \mathbf{C}'}$$

(A is an atomic formula. **K**, **L**, **C**, **D**, and **C'** are sequences of atomic formulas. $mgu(A, A')$ is the most general unifier between A and A'.)

$mf(C_1, \ldots, C_m)$ is a modular constraint that is equivalent to $C_1, \ldots, C_m$. If $C_1, \ldots, C_m$ is inconsistent, $mf(C_1, \ldots, C_m)$ does not exist. In this case, the above inference rule is inapplicable.

For example,

$$mf(member(X, [a, b, c]), member(X, [b, c, d]))$$

returns a new constraint $c0(X)$, where the definition of $c0$ is

$c0(b)$.

$c0(c)$.

and

$$mf(member(X,[a,b,c]),member(X,[k,l,m]))$$

is not defined.

This transformation is done by repeating unfold/fold transformations as described later.

3.2 Comparison with conventional approaches

In normal Prolog, constraints are inserted in adequate places in the goal and processed as procedures. It is not desirable for a declarative programming language ,and the execution can be ineffective when constraints are inserted in insufficient places. On the contrary, in cu-Prolog, constraints are described and processed declaratively.

As constraints are rewritten at every unification, cu-Prolog has also more powerful descriptive ability than the bind-hook technique. For example, *freeze* in Prolog II[3] can impose constraints on one variable, so that when the variable is instantiated, the constraints are executed as a procedure. Freeze has, however, two disadvantages. First, *freeze* cannot impose a constraint on plural variables at one time. For example, it cannot express the following CAHC.

$$f(X),g(Y,Z);append(X,Y,Z).$$

Second, since the contradiction between constraints is not detected until the variable is instantiated, there is a possibility of executing useless computation in constraints deadlocking. For example, even after executing

$$freeze(X,member(X,[a,b]))$$

and

$$freeze(Y,member(Y,[u,v]))$$

X and Y are unifiable. In cu-Prolog,

$$f(X);member(X,[a,b]).^{3}$$

and

$$f(Y);member(Y,[u,v]).$$

are not unifiable.

[3] $member(X,[a,b])$ is not modular, but is equivalent to $p1(X)$,where

$p1(a)$.

$p1(b)$.

3.3 Constraint Transformation

This subsection explains the mechanism of constraint transformation in cu-Prolog.

Let $\mathcal{T}$ be a set of definition clauses of modularly defined constraints, Σ be a set of constraints $\{C_1, \ldots, C_n\}$ that contains variables $x_1, \ldots, x_m$, and p be a new m-ary predicate.

Let $\mathcal{D}$ be a set of definition clauses of new predicates, and

$$\mathcal{P}_0 = \mathcal{T} \cup \mathcal{D}$$

$\mathcal{D}$ is initially

$$\{p(x_1, \ldots, x_m) :- C_1, \ldots, C_n.\}$$

and other new predicates are added to $\mathcal{D}$ in the transformation process.

Then, $mf(\Sigma)$ returns $p(x_1, \ldots, x_m)$, if and only if there exists a sequence of program clauses

$$\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n$$

and every predicate in $\mathcal{P}_n$ is modularly defined. In the above sequence, $\mathcal{P}_{i+1}$ is derived from $\mathcal{P}_i$ $(0 \leq i < n)$ by one of the following three types of transformations.

1. *unfold transformation*
   Select one clause ζ from $\mathcal{P}_i$ and one atomic formula A from the body in ζ. Let $\zeta_1, \ldots, \zeta_n$ be all the clauses in $\mathcal{P}_i$ whose heads unify with A, and ζ'_j be the result of applying ζ_j to A of ζ $(j = 1, \ldots, n)$.
   $\mathcal{P}_{i+1}$ is obtained by replacing ζ in $\mathcal{P}_i$ with $\zeta'_1, \ldots, \zeta'_n$.

2. *fold transformation*
   Let $\zeta(A :- \mathbf{K}, \mathbf{L}.)$ be a clause in $\mathcal{P}_i$, and $\xi(B :- \mathbf{K}'.)$ be a clause in $\mathcal{D}$, and θ be the most general unifier between $\mathbf{K}$ and $\mathbf{K}'$ [4] that meets the following conditions:

   (a) $\mathbf{K}$ and $\mathbf{L}$ have no common variables, and

   (b) $\zeta \notin \mathcal{D}$.

   Then, $\mathcal{P}_{i+1}$ is obtained by replacing ζ in $\mathcal{P}_i$ with $A\theta :- B\theta, \mathbf{L}$.

3. *integration*
   Let $\zeta\ (A :- \mathbf{K}, \mathbf{L}.)$ be a clause in $\mathcal{P}_i$, where $\mathbf{K}$ is not modular and contains variables $x_1, \ldots, x_m$ and there are no common variables between $\mathbf{K}$ and $\mathbf{L}$. Let $p()$ be a new m-ary predicate whose definition is ξ :

   $$p(x_1, \ldots, x_m) :- \mathbf{K}.$$

   Then, $\mathcal{P}_{i+1}$ is obtained by replacing ζ in $\mathcal{P}_i$ with

   $$A :- p(x_1, \ldots, x_m), \mathbf{L}.$$

   and adding ξ to $\mathcal{P}_i$. ξ is also added to $\mathcal{D}$.

[4] For example, the most general unifier between $f(a, b, X), g(Y, c)$ and $g(d, U), f(V, b, W)$ is $\{U/c, V/a, X/W, Y/d\}$.

The third transformation can be seen as a special case of fold transformation. Hence, these three transformations preserve the semantics of programs because unfold/fold transformation has been proved as valid [12].

Example.

The following example shows a transformation of

$$\texttt{member(A,Z)}, \texttt{append(X,Y,Z)}$$

Here, $\mathcal{T}$ is { T1,T2,T3,T4 }, where

```
T1 = member(X,[X|Y]).
T2 = member(X,[Y|Z]):-member(X,Z).
T3 = append([],X,X).
T4 = append([A|X],Y,[A|Z]):-append(X,Y,Z).
```

and Σ is {member(A,Z),append(X,Y,Z)}.

Step1 : The new predicate p1 is defined as

```
D1 = p1(A,X,Y,Z):-member(A,Z),append(X,Y,Z).
```

and

$$\mathcal{P}_0 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{D1}\}$$

$$\mathcal{D} = \{\texttt{D1}\}$$

Step2 : Unfolding the first formula of D1's body (member(A,Z)), we get

```
T5 = p1(A,X,Y,[A|Z]):-append(X,Y,[A|Z]).
T6 = p1(A,X,Y,[B|Z]):-member(A,Z),append(X,Y,[B|Z]).
```

So

$$\mathcal{P}_1 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}\}$$

Step3 : By integration (p2 and p3 are new predicates) ,

```
T5' = p1(A,X,Y,[A|Z]):-p2(X,Y,A,Z).
T6' = p1(A,X,Y,[B|Z]):-p3(A,Z,X,Y,B).
D2  = p2(X,Y,A,Z):-append(X,Y,[A|Z]).
D3  = p3(A,Z,X,Y,B):-member(A,Z),append(X,Y,[B|Z]).
```

and

$$\mathcal{P}_2 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}', \texttt{T6}', \texttt{D2}, \texttt{D3}\}$$

$$\mathcal{D} = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$$

Step4 : By unfolding D2,

```
T7 = p2([],[A|Z],A,Z).
T8 = p2([B|X],Y,A,Z):-append(X,Y,Z).
```

These clauses comprise the modular definition of p2. Thus

$$\mathcal{P}_3 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{D3}\}.$$

Step5 : Unfold the second formula of D3's body (append(X,Y,[B|Z])), and we have

```
T9  = p3(A,Z,[ ],[B|Z],B):-member(A,Z).
T10 = p3(A,Z,[B|X],Y,B):-member(A,Z),append(X,Y,Z).
```

$$\mathcal{P}_4 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}\}.$$

Step6 : Folding T10 by D1 will generate

```
T10' = p3(A,Z,[B|X],Y,B):-p1(A,X,Y,Z).
```

Accordingly

$$\mathcal{P}_5 = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}'\}.$$

Every predicate in $\mathcal{P}_5$ is modularly defined. As a result,

$$\texttt{member(A,Z)}, \texttt{append(X,Y,Z)}$$

has been transformed to p1(A,X,Y,Z) preserving equivalence, and the following new clauses have been defined.

$$\{\texttt{T5}', \texttt{T6}', \texttt{T7}, \texttt{T8}, \texttt{T9}, \texttt{T10}'\}.$$

3.4 Implementation

The source code of cu-Prolog is, at present (Ver 2.0), composed of 4,500 lines of language C on UNIX or MS-DOS.

For the effective implementation of the constraint transformation, some heuristics are necessary. In the current implementation, as an object term of the unfold transformation, one atomic formula is selected in the following order:

1. The atomic formula of the finite predicate.
2. The atomic formula that has constants or [ ] (nil) in its arguments.
3. The atomic formula that has lists in its argument.
4. The atomic formula that has plural dependent variables.
5. Any atomic formula.

Here,

[Def] 4 (finite predicate) *A predicate $p()$ is* finite, *when the body of every definition clause of $p()$ is*

1. *nil*, or
2. *composed of finite predicates*

Above selection rule is effective in the following examples.

Example 1 *Transformation of $m(X,Y), p(X)$, where*

$m(X,[\,]).$
$m([A|B],X) :- m(B,X).$
$p([\,]).$
$p([a]).$

If we always select the leftmost atomic formula, that causes an infinite loop because $m(\,)$ is always unfolded. However, with the above rule, because predicate m is infinite and p is finite, $p(X)$ is unfolded for the first time, and the transformation easily stops.

Example 2 *Transformation of*

$$fuse(A,B,C), fuse(C,D,E), fuse(E,F,G)$$

where

$fuse([\,],[\,],[\,]).$
$fuse([A|X],Y,[A|Z]) :- fuse(X,Y,Z).$
$fuse(X,[A|Y],[A|Z]) :- fuse(X,Y,Z).$
$fuse([A|X],[A|Y],Z) :- fuse(X,Y,Z).$

The leftmost selection causes an infinite loop. Moreover, in this example, unfolding from the formula except $fuse(C,D,E)$ also causes an infinite loop. Because $fuse(C,D,E)$ alone has two dependent variables (C and E), it is selected by the selection rule 4 and the transformation succeeds.

Figure 1 demonstrates constraint transformation routine of cu-Prolog.

4 A JPSG parser

As an application of cu-Prolog, a natural language parser based on the unification based grammar has been considered first of all. Since constraints can be added directly to the program clauses representing a lexical entry or a phrase structure rule, the grammar is implemented more naturally and declaratively than with ordinary Prolog. Here we describe a simple Japanese parser of JPSG in cu-Prolog. CAHC plays an important role in two respects.

First, CAHC is used in the lexicon of homonyms or polysemic words. For example, a Japanese noun "hasi" is 3-way ambiguous, it means a bridge, chopsticks, or an edge. This polysemic word can be subsumed in the following single lexical entry.

$$lexicon([hasi|X], X, [\ldots sem(SEM)]); hasi\_sem(SEM).$$

where *hasi_sem* is defined as follows.

```
_member(X,[X|Y]).
_member(X,[Y|Z]):-member(X,Z).
_append([],X,X).
_append([A|X],Y,[A|Z]):-append(X,Y,Z).

_@ member(X,[ga,no,wo,ni,kara,made,sae]),member(X,[to,he,ni,kara,sura,ga]).

solution = c0(X)
c3(ni).
c3(kara).
c0(ga).
c0(X0):-c3(X0).
CPU time = 0.017 sec

_@ member(A,Z),append(X,Y,Z).

solution = c14(A, Z, X, Y)
c15(X2, X2, X0, Y1, Y3):-append(X0, Y1, Y3).
c15(X2, Y3, X0, Y1, Z4):-c14(X2, Z4, X0, Y1).
c14(A0, X1, [], X1):-member(A0, X1).
c14(A0, [A1|Z4], [A1|X2], Y3):-c15(A0, A1, X2, Y3, Z4).
CPU time = 0.000 sec
```

The first four lines are definitions of *member* and *append*. The lines that begin with "@" are user's input atomic formulas (constraints). cu-Prolog returns the constraint ($c0(X)$) that is equivalent to the input constraint, and its definitions. (The cpu time is counted by 60ths of a second).

図 1: Demonstration of the constraint transformation routine

$hasi\_sem(bridge)$.
$hasi\_sem(chopsticks)$.
$hasi\_sem(edge)$.

The value of the semantic feature is a variable (SEM), and the constraint on SEM is $hasi\_sem(SEM)$. Note that predicate *hasi_sem* is modularly defined. According to CAHC, such ambiguity may be considered at one time, instead of being divided in separate lexical entries. Japanese has such an ambiguity is also shown in conjugation, post positions, etc. They can be treated in this manner.

Second, a phrase structure rule is written naturally in a CAHC. In JPSG [5], FFP(FOOT Feature Principle) is:

> The value of a FOOT feature of the mother unifies with the union of those of her daughters.

This principle is embedded in a phrase structure rule as follows:

$$psr([slash(MS)],[slash(LDS)],[slash(RDS)]);union(LDS,RDS,MS).$$

However, this cannot be described in this manner in traditional Prolog.

Figure 2 shows a simple demonstration of our JPSG parser, and Figure 3 shows an example of treating ambiguity as constraint. The current parser treats a few feature and has little lexicon. However, the expansion is easy. It parses about ten to twenty words sentences within a second on VAX8600. Since JPSG is a declarative grammar formalism and cu-Prolog describes JPSG also

declaratively, the parser needs parsing algorithms independently. In the current implementation, we adopt the left corner parsing algorithm [1]. Furthermore, we would even be able to abandon parsing algorithm altogether [8].

```
_:-p([ken,ga,naomi,wo,ai,suru]).

v[Form_928, AJN{Adj_930}, SC{SubCat_932}]:SEM_934---[suff_p]
 |
 |--v[vs2, SC{Sc_922}]:[love,Sbj_589,Obj_591]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p]
 |   |   |
 |   |   |--n[n]:ken---[ken]
 |   |   |
 |   |   |__p[ga, AJA{n[n]}]:ken---[ga]
 |   |
 |   |__v[vs2, SC{p[ga], Sc_922}]:[love,Sbj_589,Obj_591]---[subcat_p]
 |       |
 |       |--p[wo]:naomi---[adjacent_p]
 |       |   |
 |       |   |--n[n]:naomi---[naomi]
 |       |   |
 |       |   |__p[wo, AJA{n[n]}]:naomi---[wo]
 |       |
 |       |__v[vs2, SC{p[wo], p[ga], Sc_922}]:[love,Sbj_589,Obj_591]---[ai]
 |
 |__v[Form_928, AJA{v[vs2,SC{Sc_922}]}, AJN{Adj_930}, SC{SubCat_932}]:SEM_934---[suru]

cat     cat(v, Form_928, [], Adj_930, SubCat_932, SEM_934)
cond     [c2(Sc_922, Obj_591, Sbj_589, Form_928, SubCat_932, Adj_930, SEM_934)]
True.
CPU time = 0.050 sec

_:-c2(_,_,_,F,SC,ADJ,SEM).
  F = syusi  SC = []  ADJ = []  SEM = [love,ken,naomi];
```

The first line is a user's input. "Ken ga Naomi wo ai suru" means "Ken loves Naomi."
Then, the parser returns the parse tree and the category and constraint ($c2(\ )$) of the top node. User solves the constraint to get the actual value of the variables.

図 2: Demonstration of our JPSG parser

5 Final Remarks

The further study of cu-Prolog has many prospects. For example, to expand descriptive ability of constraints, the negative operator or the universal quantifier can be added. The constraint-based, alias partial, aspects of Situation Semantics[2] are naturally implemented in terms of an extended version of cu-Prolog [7]. For practical applications in Artificial Intelligence in general and natural language processing in particular, one needs a mechanism for carrying out computation partially, instead of totally as described above, where constraint transformation halts only when the constraint in question is entirely modular. So the most difficult problem one must tackle concerns itself with heuristics about how to control computation.

Acknowledgments

```
_:-p([ai,suru,hito]).

n[n]:Semantics_430---[adjunct_p]
 |
 |--v[Form_416, AJN{n[n]}, SC{_428}]:Semantics_430---[suff_p]
 |   |
 |   |--v[vs2, SC{Sc_198}]:[love,Sbj_81,Obj_83]---[ai]
 |   |
 |   |__v[Form_416, AJA{v[vs2,SC{Sc_198}]}, AJN{n[n]}, SC{_428}]:Semantics_430---[suru]
 |
 |__n[n]:inst(Obj_487, [people,Obj_487])---[hito]

cat     cat(n, n, [], [], [], Semantics_430)
cond     [c6(Sc_198, Obj_83, Sbj_81, Form_416, _428, Obj_487, Semantics_430)]
True.
CPU time = 0.017 sec
_:-c6(_,_,_,_,_,_,Sem).
  Sem = inst(Obj0_72, [and,[people,Obj0_72],[love,Sbj1_74,Obj0_72]]);
  Sem = inst(Sbj0_72, [and,[people,Sbj0_72],[love,Sbj0_72,Obj1_74]]);
```

This is a parse tree of "ai suru Ken" that has two meaning: "Ken whom someone loves" or "Ken who loves someone". These ambiguity is shown in two solution of the constraint.

図 3: Example of ambiguity

This study owes much to our colleagues in the JPSG Working Group (chairman : GUNJI Takao) at ICOT, Prof. YAMADA Hisao, Mr.ONO Yoshihiko, and NAKANO Mikio. The implementation of cu-Prolog was supported by ICOT and the Ministry of International Trade and Industry in Japan.

参考文献

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing.* Prentice-Hall, 1972.

[2] J. Barwise and J. Perry. *Situation and Attitudes.* MIT Press, Cambridge, Mass, 1983.

[3] A. Colmerauer. Prolog II Reference Manual and Theoretical Model. Technical report, ER-ACRANS 363, Groupe d'-Intelligence Artifielle, Universite Aix-Marseille II, October 1982.

[4] A. Colmerauer. Prolog III. *BYTE*, August 1987.

[5] T. GUNJI. *Japanese Phrase Structure Grammar.* Reidel, Dordrecht, 1986.

[6] K. HASIDA. Conditioned Unification for Natural Language Processing. In *Proceedings of the 11th COLING*, pages 85–87, 1986.

[7] K. HASIDA. A Constraint-Based View of Language. In *Proceedings of Workshop on Situation Theory and its Application*, 1989.

[8] K. HASIDA and S. ISIZAKI. Dependency Propagation: A Unified Theory of Sentence Cmprihension and Generation. In *Proceedings of IJCAI*, 1987.

[9] S. M. Shieber. *An Introduction to Unification-Based Approach to Grammar.* CSLI Lecture Notes Series No.4. Stanford:CSLI, 1986.

[10] H. TUDA. A JPSG Parser in Constraint Logic Programming. Master's thesis, Department of Information Science, University of Tokyo, 1989.

[11] H. TUDA, K. HASIDA, and H. SIRAI. JPSG Parser on Constraint Logic Programming. In *Proc. of 4th ACL European Chapter*, pages 95–102, 1989.

[12] 古川康一, 溝口文雄 共編. プログラム変換. 知識情報処理シリーズ 7. 共立出版, 東京, 1987.

[13] 相場亨. 制約論理プログラミング. *bit*, 20(1):89–97, 1988.

[14] 白井英俊, 橋田浩一. 条件付単一化. コンピュータソフトウェア, 3(4):28–38, 1986.