JPSG Parser on Constraint Logic Programming

TUDA, Hirosi *
Department of information science
Faculty of science
University of Tokyo 7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan
e-mail: a30728%tansei.cc.u-tokyo.junet@relay.cs.net

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

SIRAI, Hidetosi
Tamagawa University
6-1-1 Tamagawa gakuen, Machida-shi Tokyo, 194 Japan
e-mail: a88868%tansei.cc.u-tokyo.junet@relay.cs.net

Abstract

This paper presents a 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 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 [11].

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 [13]. In cu-Prolog, user defined constraints can be directly added to a program clause (constraint added Horn clause), and the constraint unification [12, 8] [1] is adopted instead of the nor-

*From this April, Fujitsu Corporation

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

mal unification. This paper discusses the outline of the cu-Prolog system, and presents a Japanese parser based on JPSG (Japanese Phrase Structure Grammar) [7] as a suitable application of cu-Prolog.

2 Constraint Added Horn Clause (CAHC)

Most of the constraint logic programming language systems (CAL [2], PrologIII [5], 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):

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

   (H is called the head, $B_1, B_2, \ldots, B_n$ is the body, $C_1, C_2, \ldots, C_m$ is the constraint. The body and the constraint can be empty.)

$C_1, C_2, \ldots, C_m$ comprise a set of constraints on the variables occurring in the rest of the clause. $C_1, C_2, \ldots, C_m$ must be, in the current implementation, *modular* in the sense that it has the following canonical form.

[Def.] 1 (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.] 2 (modularly defined) *Predicate p is* modularly defined, *when in every definition clause of p,* $P' :- D.$,

D is empty,

or

1. *every argument of D is variable,*
2. *no variable occurs in two distinct space, 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.

Seen from the declarative semantics, the program clause of cu-Prolog is equivalent to the following program clause of Prolog:

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

3 cu-Prolog

3.1 Constraint Unification

cu-Prolog employs Constraint Unification [12, 8] which is the usual Prolog unification plus constraint transformation (normalization).

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

$$Q, R; C. \ , \ Q' :- S; D.,$$

$$\frac{\theta = mgu(Q, Q'), B = mf(C\theta, D\theta)}{S\theta, R\theta; B}$$

(Q is an atomic formula. R, C, S, D, and B are sequences of atomic formulas. $mgu(Q, Q')$ is a most general unifier between Q and Q'.)

$mf(C_1, \ldots, C_m)$ is a modular constraint which is equivalent to $C_1, \ldots, C_m$. If $C_1, \ldots, C_m$ is inconsistent, $mf(C_1, \ldots, C_m)$ is not defined. 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 undefined.

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 a 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 a insufficient place.

As constraints are rewritten at every unification, cu-Prolog has more powerful descriptive ability than the bind-hook technique. For example, *freeze* in Prolog II[4] 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, X and Y are unifiable even after executing

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

and

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

In cu-Prolog,

$$f(X); member(X,[a,b]).^2$$

and

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

are not unifiable.

3.3 Constraint Transformation

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

Let $\mathcal{T}$ be 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 definition clauses of new predicates, and

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

$\mathcal{D}$ is initially

$$\{p(x_1,\ldots,x_m) :- C1,\ldots,C_n.\}$$

and other new predicates are included through the constraint normalization.

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

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

and $\mathcal{P}_n$ is modularly defined, where $\mathcal{P}_{i+1}$ is derived from $\mathcal{P}_i$ $(0 \le i < n)$ by one of the following three types of transformations.

1. *unfold transformation*
   Select one clause C from $\mathcal{P}_i$ and one atomic formula A from the body of C. Let $C_1,\ldots,C_n$ be all the clauses whose heads unify with A, and C'_j be the result of applying C_j to A of C $(j=1,\ldots,n)$. $\mathcal{P}_{i+1}$ is obtained by replacing C in $\mathcal{P}_i$ with $C'_1,\ldots,C'_n$.

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

$p1(a)$.
$p2(b)$.

2. *fold transformation*
   Let $C(A : -K\&L.)$ be a clause in $\mathcal{P}_i$, and $D(B : -K'.)$ be a clause in $\mathcal{D}$, and θ be $mgu(K, K')$ that meets the following conditions.

   (a) No variables occur in both K and L, and

   (b) C is not contained in $\mathcal{D}$.

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

3. *integration*
   Let C $(H : -B\&R.)$ be a clause in $\mathcal{P}_i$, where B is not modular and contains variables $x_1, \ldots, x_m$ and there are no common variables between B and R. Let p be a new m-ary predicate and the following clause E:

$$p(x_1, \ldots, x_m) : -B.$$

   be the definition of p. Then, $\mathcal{P}_{i+1}$ is obtained by replacing C in $\mathcal{P}_i$ with

$$H : -p(X_1, \ldots, x_m)\&R.$$

   and adding E. E is also added to $\mathcal{D}$.

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 [6].

The following example shows a transformation of

$$member(A, Z), append(X, Y, Z).$$

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 $\{member(A, Z), append(X, Y, Z)\}$. The new predicate $p1$ is defined as

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

and

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

Unfolding the first formula of D1's body, 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

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

By integration,

```
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

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

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

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

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

Unfold the second definition of D3, 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).
```

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

Folding T10 by D1 will generate

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

Accordingly

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

As a result,

$$member(A, Z), append(X, Y, Z)$$

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

$$\{\mathtt{T4}, \mathtt{T5'}, \mathtt{T6'}, \mathtt{T7}, \mathtt{T8}, \mathtt{T9}, \mathtt{T10'}\}.$$

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 system. Its precise computation speed is under evaluation, but is sufficient for practical use.

Implementation of the effective constraint transformation shown in above subsection requires some heuristics in the application of three transformation. Especially, in unfold transformation, one atomic formula A is selected in the following heuristic rules

1. The atomic formula of the finite predicate.
2. The atomic formula that has constants or [ ] in its arguments.
3. The atomic formula that has lists in its argument.
4. The atomic formula that has plural dependencies.

Here,

[Def.] 3 (finite predicate) *A predicate p is* finite, *when the body of every definition clause of p is*

1. *nil,* or
2. *expressed by finite predicates*

Figure 1 demonstrates constraint transformation.

4 A JPSG parser

As an application of cu-Prolog, a natural language parser based on unification based grammar has been considered first of all. Since constraints can be added directly to the program clause 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.

$$lexicon([hasi|X], X, [\ldots semSEM]);$$

$$hasi\_sem(SEM).$$

where *hasi_sem* is defined as follows.

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 [7], 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:

$$psr([slashMS], [slashLDS], [slashRDS]);$$

$$union(LDS, RDS, MS).$$

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

```
_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,wo,ni]),member(X,[no,wo,ni]).

solution = c0(X)
c1(wo).
c1(ni).
c0(X0):-c1(X0).

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

solution = c7(A, Z, X, Y)
c8(X2, X2, X0, Y1, Y3):-append(X0, Y1, Y3).
c8(X2, Y3, X0, Y1, Z4):-c7(X2, Z4, X0, Y1).
c7(A0, X1, [], X1):-member(A0, X1).
c7(A0, [A1|Z4], [A1|X2], Y3):-c8(A0, A1, X2, Y3, Z4).
```

The first four lines are definitions of *member* and *append*. The lines that begin with "@" are user's input atomic formulas (constraints). The system returns the constraint ($c0(X)$) that is equivalent to the input constraint, and its definitions.

Figure 1: Demonstration of the constraint transformation routine

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 [10].

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[3] are naturally implemented in terms of an extended version of cu-Prolog [9]. 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

This study owes much to our colleagues in the JPSG Working group at ICOT. The implementation of cu-Prolog is supported by ICOT and the Ministry of International Trade and Industry in Japan.

References

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

[2] A. AIBA. Seiyaku Ronri Programming (Constraint Logic Programming). *bit*, 20(1):89–97, 1988. (in Japanese).

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

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

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

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

v[Form_764, AJN{Adj_768}, SC{SubCat_772}]:SEM_776---[suff_p]
 |
 |--v[vs2, SC{Sc_752}]:[love,Sbj_120,Obj_124]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p]
 |   |   |
 |   |   |--n[n]:ken---[ken]
 |   |   |
 |   |   |__p[ga, AJA{n[n]}]:ken---[ga]
 |   |
 |   |__v[vs2, SC{p[ga], Sc_752}]:[love,Sbj_120,Obj_124]---[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_752}]:[love,Sbj_120,Obj_124]---[ai]
 |
 |__v[Form_764, AJA{v[vs2,SC{Sc_752}]}, AJN{Adj_768},
  SC{SubCat_772}]:SEM_776---[suru]

cat     cat(v, Form_764, [], Adj_768, SubCat_772, SEM_776)
cond    [c2(Sc_752, Obj_124, Sbj_120, Form_764, SubCat_772, Adj_768, SEM_776)]
True.

_:-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.

Figure 2: Demonstration of our JPSG parser

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

n[n]:Semantics_824---[adjunct_p]
 |
 |--v[Form_796, AJN{n[n]}, SC{_820}]:Semantics_824---[suff_p]
 |   |
 |   |--v[vs2, SC{Sc_376}]:[love,Sbj_152,Obj_156]---[ai]
 |   |
 |   |__v[Form_796, AJA{v[vs2,SC{Sc_376}]}, AJN{n[n]}, SC{_820}]:Semantics_824---[suru]
 |
 |__n[n]:inst(Obj_932, [people,Obj_932])---[hito]

cat     cat(n, n, [], [], [], Semantics_824)
cond     [c6(Sc_376, Obj_156, Sbj_152, Form_796, _820, Obj_932, Semantics_824)]
True.

_:-c6(_,_,_,_,_,_,_,Sem).
  Sem = inst(Obj0_136, [and,[people,Obj0_136],[love,Sbj1_140,Obj0_136]])
  Sem = inst(Sbj0_136, [and,[people,Sbj0_136],[love,Sbj0_136,Obj1_140]])
```

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

Figure 3: Example of ambiguity

[6] K. FURUKAWA and F. MIZOGUTI, editors. *Program Henkan (Program Transformation). Tisiki Johoshori Series No.7*, Kyoritu, Tokyo, 1987. (in Japanese).

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

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

[9] K. HASIDA. A Constraint-Based View of Language. In *Proceedings of Workshop on Situation Theory and its Applicàtion*, 1989. (to appear).

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

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

[12] H. SIRAI and K. HASIDA. Zyookentuki Tanituka (Conditioned Unification). *Computer Software*, 3(4):28–38, 1986. (in Japanese).

[13] H. TUDA. *A JPSG Parser in Constraint Logic Programming.* Master's thesis, Department of Information Science, University of Tokyo, 1989. (to appear).