ICOT Technical Report: TR-0857

TR-0857

cu-Prolog for Constraint-Based Natural Language Processing

by

H. Tsuda

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5

Institute for New Generation Computer Technology

cu-Prolog for Constraint-Based Natural Language Processing

Hiroshi TSUDA
Institute for New Generation Computer Technology (ICOT)
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
Tel: 03-3456-3069 E-mail : tsuda@icot.or.jp

index terms:
Artificial Intelligence, Natural Language Processing, Logic Programming

Abstract

This paper introduces a constraint logic programming (CLP) language *cu-Prolog* as an implementation framework for constraint-based natural language processing.

Compared to other CLP languages, cu-Prolog has several unique features. Most CLP languages take algebraic equations or inequations as constraints. cu-Prolog, on the other hand, takes Prolog atomic formulas in terms of user-defined predicates. cu-Prolog, thus, can describe symbolic and combinatorial constraints occurring in the constraint-based grammar formalisms. As a constraint solver, cu-Prolog uses the unfold/fold transformation, which is well known as a program transformation technique, dynamically with some heuristics.

To treat the information partiality described with feature structures, cu-Prolog uses PST (Partially Specified Term) as its data structure.

Section 1 and 2 give an introduction to the constraint-based grammar formalisms on which this paper is based, and the outline of cu-Prolog is explained in section 3 with implementation issues described in section 4. Section 5 illustrates its linguistic application to disjunctive feature structure (DFS) and parsing constraint-based grammar formalisms such as Japanese Phrase Structure Grammar (JPSG). In either application, a disambiguation process is realized by transforming constraints, which gives a picture of constraint-based NLP.

1 Introduction

One of the main classification of contemporary natural language grammatical theories is whether their grammar descriptions are *rule-based* or *constraint-based* [2, 19]. [1] GPSG (Generalized Phrase Structure Grammar) and LFG (Lexical Functional Grammar) fall into the former category. The latter includes GB (Government and Binding) theory, HPSG (Head-driven PSG)[16, 17], and JPSG (Japanese PSG)[5]. By taking a constraint-based approach, more general and richer grammar formalisms are possible because morphology, syntax, semantics, and pragmatics are all uniformly treated as constraints. Also, declarative grammar description, one of the most important features of constraints, allows various flows of information during processing.

Consider their implementation environment. For rule-based grammars, many approaches have been attempted, such as FUG[13] and PATR-II[18]. As yet, however, no pathfinding work has been done on constraint-based grammars.

Our CLP language *cu-Prolog* [25, 24] aims to provide an implementation framework for constraint-based grammars. Unlike most CLP languages, cu-Prolog takes the Prolog atomic formulas of user-defined predicates as constraints.

cu-Prolog originated from the technique of *constrained unification* (or *conditioned unification* [8]) that is the unification of two constrained Prolog patterns. cu-Prolog adds constraints, given in terms of user-defined Prolog predicates, to Horn clauses (called *Constrained Horn Clause*). The

[1] Constraint-based approaches are also called *information-based* or *principle-based* approaches.

constraint solver of cu-Prolog uses the unfold/fold transformation[21], which is well known as a program transformation technique, dynamically with some heuristics. To describe information partiality in constraint-based grammars, cu-Prolog also provides PST(Partially Specified Term)[15] data structure.

This paper illustrates

- the outline of cu-Prolog,
- the treatment of disjunctive feature structures with constrained PST, and
- the JPSG parser - its most successful application - to illustrates constraint-based natural language processing.

2 Linguistic Constructions

As an introduction to computational linguistics, this section explains the some linguistic constructions occurring in constraint-based grammar formalisms.

2.1 Disjunctive Feature Structure (DFS)

Unification-based grammars utilize *feature structures* as basic structures for treating information partiality. A feature structure consists of a set of label/value pairs. In (1), *pos* and *sc* are called *features* and their values are *n* and a singleton set $< [pos : p] >$.

$$\begin{bmatrix} pos : n \\ sc : \langle \begin{bmatrix} pos : p \end{bmatrix} \rangle \end{bmatrix} \tag{1}$$

Morphological, syntactic, semantic, and pragmatic information are all uniformly stored in a feature structure.

Natural language descriptions require some framework to enable the handling of ambiguities such as polysemic words, homonyms, and so on. *Disjunctive feature structures (DFS)s* are commonly used to handle disjunction in feature structures[13]. DFSs consist of the following two structures.

Value disjunction A value disjunction specifies alternative values for a single feature. (2) states that the value of the *pos* feature is *n* or *v*, and the value of the *sc* feature is $<>$ (empty set) or $< [pos : p] >$.

$$\begin{bmatrix} pos : \{n, v\} \\ sc : \left\{ \begin{array}{l} \langle\rangle \\ \langle \begin{bmatrix} pos : p \end{bmatrix} \rangle \end{array} \right\} \end{bmatrix} \tag{2}$$

General disjunction A general disjunction specifies alternative groups of multiple features. In (3), feature *sem* is common, the rest being two-way ambiguous.

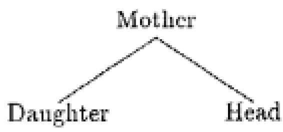
$$\begin{bmatrix} \left\{ \begin{array}{l} \begin{bmatrix} pos : n \end{bmatrix} \\ \begin{bmatrix} pos : v \\ vform : vs \\ sc : \langle \begin{bmatrix} pos : p \end{bmatrix} \rangle \end{bmatrix} \end{array} \right\} \\ sem : love(X, Y) \end{bmatrix} \tag{3}$$

One serious problem in treating DFSs is the computational complexity of their unification, because essentially NP-complete[12]. Some practical, efficient algorithms have been studied by [11, 4].

2.2 Structural Principles

Unification-based grammars are phrase structures whose nodes are feature structures. Their grammar descriptions consist of a phrase structure and local constraints called *structural principles* in a phrase structure. Current constraint-based grammars such as HPSG and JPSG have general and few phrase structure and grammatical information is mainly described with structural principles.

JPSG[5] is a constraint-based grammar designed specifically for application to Japanese. It has been developed by the PSG working group at ICOT. JPSG has only one binary phrase structure.



This phrase structure is applicable to both the *complementation structure* and the *adjunction structure* of Japanese[2]. In the complementation structure, *Daughter* is a complement, and also acts as a modifier in the adjunction structure.

Structural principles are defined as constraints (relations) among the features in the local phrase structure. In the following, we explain some features and their constraints.

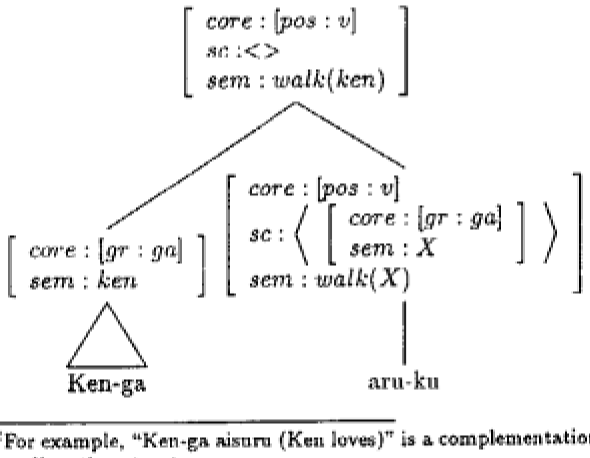
head features: Features such as *core*, which specifies core categories such as *pos* (part of speech) and *gr* (grammatical relation), and *sem* (semantics) are called *head features*. These conform to the *head feature principle*:

> The value of a head feature of the mother unifies with that of the head.

subcat features: Features *sc* (subcategorization) and *adjacent* are called *subcat features*. They take a set of feature structures that specify complement categories and conform to the *subcat feature principle*:

> In the complementation structure, the value of a subcat feature of the mother unifies with that of the head minus left daughter.

Below is a JPSG-like analysis of the Japanese sentence "Ken-ga aruku (Ken walks)." According to the subcat feature principle, variable X binds to *ken*.



[2]For example, "Ken-ga aisuru (Ken loves)" is a complementation structure, and "ooki-na yama (big mountain)" is an adjunction structure.

3 cu-Prolog

3.1 Conventional Approaches

Prolog is often used as an implementation language for unification-based grammars[18]. Its computational rules, however, are fixed and procedural, that is, always from left to right for AND processes, and from top to bottom for OR processes. Prolog programmers intentionally have to align goals such that they are solved efficiently. Prolog, therefore, is not well-suited for constraint-based grammars because it is impossible to stipulate in advance which type of linguistic constraints are to be processed, and in what order.

Some Prolog-like systems such as PrologII and CIL[15] employ the *bind-hook* mechanism that can delay some goals (constraints) until certain variables bind to ground terms. However, as the mechanism can only check frozen constraints only by executing them, it is not always efficient.

Most CLP languages, such as CLP(R)[10], PrologIII, and CAL, take the constraints of the algebraic domain with equations or inequations. Their constraint solvers are based on algebraic algorithms such as deriving Gröbner bases, solving equations, and so on. However, for AI applications and natural language processing systems especially, symbolic constraints are far more desirable than algebraic ones. cu-Prolog, on the other hand, can process symbolic and combinatorial constraints because its constraint domain is the Herbrand universe.

3.2 Constrained Horn Clause (CHC)

The basic component of cu-Prolog is the *Constrained Horn Clause (CHC)* [3].

[Def] 1 (CHC) *The* Constrained Horn Clause (CHC) *has the following form.*

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

H, B_i, and C_j are atomic formulas. The body *and* constraint *can be empty.* □

From the viewpoint of declarative semantics, the above is equivalent to the following Horn clause.

$$HEAD \ :\text{-}\ B_1, B_2, \ldots, B_n, C_1, C_2, \ldots, C_m.$$

3.3 Derivation Rule

cu-Prolog expands the derivation rule of Prolog by adding a constraint transformation operation.

$$\frac{\overbrace{A, \mathbf{K}; \mathbf{C}.}^{goal} \quad \overbrace{A' :\text{-} \mathbf{L}; \mathbf{D}.}^{program} \quad \overbrace{\theta = mgu(A, A')}^{substitution} \quad \overbrace{\mathbf{C}' = mf(\mathbf{C}\theta \cup \mathbf{D}\theta)}^{constraint\ transformation}}{\underbrace{\mathbf{L}\theta, \mathbf{K}\theta; \mathbf{C}'.}_{new\ goal}}$$

A and A' are heads. $\mathbf{K}$ and $\mathbf{L}$ are bodies, $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{C}'$ are constraints. $mgu(A, A')$ is the most general unifier between A and A'. $mf(\mathbf{Cstr})$ is a simplified (*modular*) constraint that is equivalent to **Cstr** (see subsections 3.5 and 3.6). As a computational rule, when the transformation of $\mathbf{C}\theta \cup \mathbf{D}\theta$ fails, the above derivation rule is not applied.

Note that the body part of CHC is processed procedurally with a fixed computation rule as Prolog. However, the constraint part is solved by constraint transformation with the heuristics as shown in subsection 3.6 and section 4. It is efficient to realize procedural processes such as parsing algorithms in the body, and unspecified processes such as linguistic constraints in the constraint part.

[3] Or *Constraint Added Horn Clause (CAHC)*.

3.4 PST

cu-Prolog adopts a PST[15] data structure that corresponds to the feature structure of unification-based grammars.

[Def] 2 **(Partially Specified Term (PST))** PST *is a set of label/value pairs, having the following form:*

$$\{l_1/t_1, l_2/t_2, \ldots\} .$$

l_i, *called* label, *is an atom and* $l_i \neq l_j (i \neq j)$. t_i, *called* value, *is a term.* □

An infinite PST structure such as $X = \{l/X\}$ is not allowed in cu-Prolog.

[Def] 3 **(Unification between PSTs)** *Let X,Y, and Z be PSTs. Z is the unification between X and Y when*

- $\exists l, l/x \in X, l/\_ \notin Y \rightarrow l/x \in Z$
- $\exists l, l/\_ \notin X, l/y \in Y \rightarrow l/y \in Z$
- $\exists l, l/x \in X, l/y \in Y \rightarrow l/unify(x, y) \in Z$

For example, {l/a,m/X} and {m/b,n/c} unify to give {l/a,m/b,n/c}.

[Def] 4 **(constrained PST)** *in the constraint part of CHC, a PST is introduced in an equal(unify) constraint, sometimes with other relevant constraints such as:*

$$\mathtt{X} = \mathrm{PST}, \mathtt{c_1(X)}, \mathtt{c_2(X)}, \ldots, \mathtt{c_n(X)}.$$

We call the above kind of description constrained PST. □

Note that X=PST corresponds to the unconditional conjunct of [11] and $\mathtt{c_1(X)}, \mathtt{c_2(X)}, \ldots, \mathtt{c_n(X)}$ the conditional conjuncts.

3.5 Modular: simplified form of constraint

A constraint in CHC has a simplified form called *modular*[8]. Modular is checked syntactically and used in the constraint transformer (3.6). Intuitively, a constraint is modular when all the arguments are different variables. For example, member(X, Y), member(U, V) is modular, and member(X, Y), member(Y, Z) or append(X, Y, [a, b, c, d]) are not modular. Modular constraints are satisfiable if each atomic formula is satisfiable [4]. In this subsection, we extend the notion of *modular* for constrained PST.

[Def] 5 **(component)** *The* component *of an argument of a predicate is a set of labels in PSTs to which the argument can bind. Here, an atom or a complex term is regarded as a PST of the label* []. □

Cmp(p,n) represents the component of the nth argument of a predicate p. Cmp(T) represents a set of labels of a PST T. In a constraint of the form X=t, variable X is regarded as being in the argument position whose component is Cmp(t).

Components can be computed by static analysis of the program [23], repeating the following procedure until there are no changes. The process always stops because the length of every component does not exceed the number of PST labels.

1. If there is a non-variable term T in the nth argument of predicate p in a head, add Cmp(T) to Cmp(p,n).
2. If there is a variable occurring both in the nth argument of predicate p in a head and mth argument of q in the body of the clause, add Cmp(q,m) to Cmp(p,n).

[4] Note that a modular constraint is not the canonical form of constraints

In the following example program, non-empty components are `Cmp(c0,1)={f,g,h}`, `Cmp(c2,1)={f,h}`, and `Cmp(c0,2)=Cmp(c1,2)={[]}`.

```
c0({f/b},X,Y):-c1(Y,X).
c0(X,b,_):-X={g/c},c2(X).
c1(X,X).
c1(X,[X|_]).
c2({h/a}).
c2({f/c}).
```

[Def] 6 (dependency) *A sequence of atomic formulas has* dependency *when*

1. *a variable occurs in two distinct places where their components have common labels,*
2. *a variable occurs in two distinct places where one component contains* [] *and another does not contain* [], *or*
3. *the binding of an argument whose component is not* ϕ. □

[Def] 7 (modular) *A sequence of atomic formulas is* modular *when it contains no dependency.* □

[Def] 8 (modularly defined) *A predicate is* modularly defined *when every body of its definition has no dependency.* □

User-defined predicates in a constraint part of CHC must be modularly defined. For example, member/2, append/3, or finite predicates are modularly defined. [5]

3.6 Constraint Transformation

The constraint solver ($mf(Cstr)$) transforms non-modular constraints into modular ones by defining new predicates. In the following, we refer to this solver as the *constraint transformer*. The constraint transformer dynamically utilizes *unfold/fold transformation* that preserves equivalence[21].

Section 4 explains implementation issues, including the heuristics of the constraint transformer.

3.6.1 Mechanism of constraint transformation

Unfold/fold transformation[21] is a well known program transformation technique. By applying the technique, we consider the transformation of a constraint $\Sigma = C_1, \ldots, C_n$.

Let $\mathcal{T}$ be a set of program Horn clauses[6], $x_1, \ldots, x_m$ be variables in Σ, and p be a new m-ary predicate. Let $\mathcal{P}_i$ and $\mathcal{D}_i$ be sequences of sets of clauses that are initially defined as:

$$\begin{aligned} \mathcal{D}_0 &= \{p(x_1, \ldots, x_m) :\!- C_1, \ldots, C_n.\} \\ \mathcal{P}_0 &= \mathcal{T} \cup \mathcal{D}_0. \end{aligned}$$

The constraint transformer $mf(\Sigma)$ returns $p(x_1, \ldots, x_m)$, if and only if there exists a sequence $\mathcal{P}_0, \ldots, \mathcal{P}_l$ such that every clause in $\mathcal{P}_l$ is modular. $\mathcal{P}_{i+1}$ and $\mathcal{D}_{i+1}$ are derived from $\mathcal{P}_i$ and $\mathcal{D}_i$ by one of the following three types of transformations ($i = 0 \ldots l$).

1. <u>*unfolding*</u>

$$\frac{\mathcal{P}_i = \{H :\!- A \cdot \mathbf{R}\} \cup \mathcal{P}'_i, \quad \{A_j :\!- \mathbf{B}_j\} \subset \mathcal{P}_i, \quad A_j\theta_j = A\theta_j \;\; (j = 1 \ldots m)}{\mathcal{P}_{i+1} = \mathcal{P}'_i \cup \bigcup_{j=1}^{m} \{H\theta_j :\!- \mathbf{B}_j\theta_j, \mathbf{R}\theta_j\} \quad \mathcal{D}_{i+1} = \mathcal{D}_i}$$

Here, A is a selected atomic formula, A_j are atomic formulas , and $\mathbf{R}$ and $\mathbf{B}_j$ are sequences of atomic formulas.

[5][20] relaxes this definition as : a predicate is *M-solvable* when at least one of the body of its definition has no dependency.

[6]$\mathcal{T}$ does not contain CHCs.

2. *folding*

$$\frac{\mathcal{P}_i = \{H :- \mathbf{C} \cdot \mathbf{R}\} \cup \mathcal{P}'_i \quad \{A :- \mathbf{B}\} \subset \mathcal{D}_i, \quad \mathbf{B}\theta = \mathbf{C}}{\mathcal{P}_{i+1} = \mathcal{P}'_i \cup \{H :- A\theta, \mathbf{R}\} \quad \mathcal{D}_{i+1} = \mathcal{D}_i}$$

Here, **C** and **R** are selected such that they have no common variables.

3. *definition*

Let **B** be a sequence of non-modular atomic formulas containing variables $x_1, \ldots, x_n$, and q be a new n-ary predicate.

$$\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{q(x_1, \ldots, x_n) :- \mathbf{B}.\}$$
$$\mathcal{P}_{i+1} = \mathcal{P}_i$$

3.6.2 Example of Constraint Transformation

The following example demonstrates a transformation of $\Sigma =$ `member(A,Z),append(X,Y,Z)`.

Firstly, by introducing a new predicate `p1/4` as $D1$, we have:

$$\begin{aligned}
\mathcal{T} &= \{T1, T2, T3, T4\} \\
T1 &= \texttt{member(X, [X|Y]).} \\
T2 &= \texttt{member(X, [Y|Z]) :- member(X, Z).} \\
T3 &= \texttt{append([ ], X, X).} \\
T4 &= \texttt{append([A|X], Y, [A|Z]) :- append(X, Y, Z).} \\
D1 &= \texttt{p1(A, X, Y, Z) :- member(A, Z), append(X, Y, Z).} \\
\mathcal{D}_0 &= \{D1\} \\
\mathcal{P}_0 &= \mathcal{T} \cup \{D1\}.
\end{aligned}$$

Step 1: By unfolding of the first formula of D1's body (`member(A,Z)`), we get

$$\begin{aligned}
T5 &= \texttt{p1(A, X, Y, [A|Z]) :- append(X, Y, [A|Z]).} \\
T6 &= \texttt{p1(A, X, Y, [B|Z]) :- member(A, Z), append(X, Y, [B|Z]).} \\
\mathcal{P}_1 &= \mathcal{T} \cup \{T5, T6\}
\end{aligned}$$

Step 2: By defining new predicates `p2/4` and `p3/5` as $D2$ and $D3$, we get the following clauses.

$$\begin{aligned}
D2 &= \texttt{p2(X, Y, A, Z) :- append(X, Y, [A|Z]).} \\
D3 &= \texttt{p3(A, Z, X, Y, B) :- member(A, Z), append(X, Y, [B|Z]).} \\
T5' &= \texttt{p1(A, X, Y, [A|Z]) :- p2(X, Y, A, Z).} \\
T6' &= \texttt{p1(A, X, Y, [B|Z]) :- p3(A, Z, X, Y, B).} \\
\mathcal{D}_2 &= \{D1, D2, D3\} \\
\mathcal{P}_2 &= \mathcal{T} \cup \{T5', T6', D2, D3\}
\end{aligned}$$

Step 3: Unfolding $D2$ gives the following clauses.

$$\begin{aligned}
T7 &= \texttt{p2([ ], [A|Z], A, Z).} \\
T8 &= \texttt{p2([B|X], Y, A, Z) :- append(X, Y, Z).} \\
\mathcal{P}_3 &= \mathcal{T} \cup \{T5', T6', T7, T8, D3\}
\end{aligned}$$

Step 4: Unfolding the second formula of $D3$'s body (`append(X,Y,[B|Z])`) gives

$$\begin{aligned}
T9 &= \texttt{p3(A, Z, [ ], [B|Z], B) :- member(A, Z).} \\
T10 &= \texttt{p3(A, Z, [B|X], Y, B) :- member(A, Z), append(X, Y, Z).} \\
\mathcal{P}_4 &= \mathcal{T} \cup \{T5', T6', T7, T8, T9, T10\}.
\end{aligned}$$

Step 5: Folding $T10$ by $D1$ generates $T10'$ and finally we get the following clauses.

$$T10' = \texttt{p3(A,Z,[B|X],Y,B) :- p1(A,X,Y,Z)}.$$
$$\mathcal{P}_5 = T \cup \{T5', T6', T7, T8, T9, T10'\}.$$

Every clause in $\mathcal{P}_5$ is modular. As a result, `member(A,Z),append(X,Y,Z)` has been transformed into `p1(A,X,Y,Z)`, preserving equivalence, and new predicates p1/4, p2/4, and p3/5 have been defined by $T5', T6', T7, T8, T9$, and $T10'$.

4 Implementation

This section presents some implementation issues, with particular emphasis on the constraint transformer.

4.1 Constraint Transformer

4.1.1 Constraint Transformation Strategy

The constraint transformer consists of the following three clause pools.

- `DEFINITION` stores the derivation clauses of new predicates,
- `NON-MODULAR` stores non-modular Horn clauses, and
- `MODULAR` stores modular Horn clauses.

`DEFINITION` realizes $\mathcal{D}_i$ and `NON-MODULAR` and `MODULAR` correspond to $\mathcal{P}_i$.

The constraint transformer repeats the following procedures until `DEFINITION` and `NON-MODULAR` are both empty.

1. If `DEFINITION` is not empty, remove one clause from `DEFINITION` and try **unfolding**.
2. If `DEFINITION` is empty but `NON-MODULAR` is not empty, remove one clause N from `NON-MODULAR`. If N's head is modular, try **unfolding**. If not, attempt **folding** or **definition** on N's body.

Actually, according to fixing the transformation strategy, some constraints cannot be transformed into modular ones, although such a situation is rare for actual linguistic constraints[22]. To avoid the situation, there are following choices:

- to adjust heuristics,
- to confine user predicates in finite or linear[22] predicates, or
- to relax the definition of modularly-defined such as M-solvable [20].

4.1.2 Heuristics

One of the outstanding features of the constraint transformer is the use of heuristics in the unfold/fold transformation.

An unfolding literal can be selected arbitrarily. The constraint transformer computes the *activation value* ε of each atomic formula for the first time, and unfolds the atomic formula of the highest value.

$$\begin{aligned}
Const &= \text{Number of arguments that bind to constants}\\
Vnum &= \text{Total number of variable occurrences in the formula}\\
Funct &= \text{Number of arguments that bind to complex terms}\\
Rec &= \text{1 for recursive predicate and 0 for finite predicate}\\
Defs &= \text{Number of definition clauses of the predicate}\\
Units &= \text{Number of unit clauses in the predicate definition}\\
Facts &= \text{If the predicate is defined only by unit clauses then 1, otherwise 0}\\
\text{activation_value } \varepsilon &= 3 * Const + Vnum + 2 * Funct - 2 * Rec - Defs + Units + 3 * Facts
\end{aligned}$$

Each factor of the activation value is defined so as to include some empirical heuristics used in [24]. There may, however, be more effective heuristics with more factors or with a non-linear formula[6].

4.2 cu-PrologIII

cu-Prolog has been implemented in C language of UNIX4.2/3BSD and on the Apple Macintosh[20]. The UNIX version of cu-Prolog (the current version is cu-PrologIII), is registered as ICOT Free Software. Anonymous FTP from `ftp.icot.or.jp` is available (the file name is `kbms-clp/unix/cuprolog.tar.Z`

5 Linguistic applications

This section demonstrates the linguistic application of cu-Prolog; DFS unification and JPSG parser.

5.1 Constraint-based NLP

In cu-Prolog, both DFSs and structural principles are treated as constraints in CHC. Moreover, constraints are accumulated to reduce the value range of variables. In other words, a disambiguation process is automatically realized by constraint transformation. This gives a picture of *constraint-based natural language processing.*

Most traditional approaches, on the other hand, are procedural and backtrack-based. That is, a parser returns one answer then backtracks to return the other answer. Alternatively, phonological, syntactic, semantic, and pragmatic processes are applied, one by one.

5.2 DFS unification

cu-Prolog requires no special device to embody the unification between two DFSs, that is, two constrained PSTs. The unification between constrained PST's is done by performing PST unification, followed by the transformation of the relevant constraints.

Consider the following example [4] of DFS unification between

$$\left[\mathtt{a} : \left\{ \begin{bmatrix} \mathtt{b} : + \\ \mathtt{c} : - \end{bmatrix}, \begin{bmatrix} \mathtt{b} : - \\ \mathtt{c} : + \end{bmatrix} \right\} \right] \text{ and } \begin{bmatrix} \mathtt{a} : [\ \mathtt{b} : \mathtt{V}\ ] \\ \mathtt{d} : \mathtt{V} \end{bmatrix}$$

. These DFSs are encoded as two constrained PSTs, `X={a/U},s(U)` and `Y={a/{b/V},d/V}`, where

```
s({b/+,c/-}).       % definition of s/1
s({b/-,c/+}).
```

PST unification between `X` and `Y` gives

```
X=Y={a/U,d/V}, U={b/V}, s(U).
```

There is a dependency in terms of a label `b`, because `Cmp(s,1)={b,c}`.

By defining a new predicate `c1/2` as follows, [7] `U={b/V},s(U)` becomes equivalent to `U={b/V},c1(U,V)`.

```
c1({c/-},+).
c1({c/+},-).
```

Note that the result `X=Y={a/U,d/V},U={b/V},c1(U,V)` does not have any dependency because `Cmp(c1,1)={c}`.

As mentioned in subsection 3.4, a constrained PST corresponds to Kasper's treatment of DFS[11]. In [11], DFS unification consists of three procedures: definite component unification, compatibility checking, and exhaustive consistency checking. PST unification corresponds to the

[7] By means of the unfold transformation, `c1/2` is defined as `c1({b/+,c/-},+).` and `c1({b/-,c/+},-).` Subsequently, omit `b` from the component of the first argument of `c1`.

first procedure, and the following constraint transformation corresponds to the second and third procedures. In the worst case, the unification requires the exponential time of the number of disjunctions, but in reality our approach requires polynomial time, as Kasper's does. The cu-Prolog approach is superior to Kasper's in the following points:

- checking is done by unfolding only dependent PSTs, and
- the unfolding formula is selected by applying heuristics as shown in Section 4, and
- constrained PST can treat disjunction names [3] which specify the value combination of different features and disjunction among different feature structures[23].

Figure 1 is an example of DFS unification in [11] in cu-PrologIII. It demonstrates the unification between

$$\begin{bmatrix} subj : \begin{bmatrix} lex : yall \\ person : 2 \\ number : pl \end{bmatrix} \end{bmatrix} \tag{4}$$

and

$$\begin{bmatrix} rank : clause \\ subj : [case : nom] \end{bmatrix} \wedge \left(\begin{bmatrix} voice : passive \\ transitivity : trans \\ [< subj >, < goal >] \end{bmatrix} \vee \begin{bmatrix} voice : active \\ [< subj >, < actor >] \end{bmatrix} \right) \tag{5}$$
$$\wedge \left(\begin{bmatrix} transitivity : intrans \\ actor : [person : 3] \end{bmatrix} \vee \begin{bmatrix} transitivity : trans \\ goal : [person : 3] \end{bmatrix} \right)$$
$$\wedge \left(\begin{bmatrix} number : sing \\ subj : [number : sing] \end{bmatrix} \vee \begin{bmatrix} number : pl \\ subj : [number : pl] \end{bmatrix} \right).$$

Here, $[< subj >, < goal >]$ indicates that the value of feature *subj* is equal to the value of *goal*.

5.3 Encoding Lexical Ambiguity

As an example of utilizing DFS, consider the lexicons of homonym or polysemic words. If the lexicon of an ambiguous word is separated into multiple entries in terms of the difference, the parsing process may be inefficient in that it sometimes backtracks to consult the lexicon. In constraint-based NLP, such ambiguity is packed as a constraint in a lexicon.

Below is a sample lexicon of the Japanese auxiliary verb "reru." "reru" follows a verb whose inflection type is `vs` or `vs1`. If the adjacent verb is transitive, "reru" indicates plain passive. If the verb is intransitive, "reru" indicates affective passive [8]. These combinations are represented by adding constraints of `reru_form/1` and `reru_sem/4` to one lexical entry.

```
%% lexical entry of ''reru''
lex(reru,{sc/SC, sem/Sem, adjacent/{pos/v,infl/I,sc/VSC,sem/Sem}});
     reru_form(I),                   % inflection (constraint)
     reru_sem(VSC,Vsem,SC,Sem).      % combination of subcat and sem (constraint)

%%%%%% definition of constraints  %%%%%%
reru_form(vs).  % inflection type of the adjacent verb
refu_form(vs1).

% constraint for intransitive (affective) passive
reru_sem([{form/ga,sem/Sbj}],Sem,[{form/ga,sem/A},{form/ni,sem/Sbj}],affected(A,Sem)).
% constraint for transitive (normal) passive
reru_sem([{form/ga,sem/Sbj},{form/wo,sem/Obj}],Sem,[{form/ga,sem/Obj},{form/ni,sem/Sbj}],Sem).
```

Although the lexicon is ambiguous, however, many kinds of constraints are automatically accumulated for solving during parsing. The disambiguation process in parsing is naturally realized by the constraint transformation in cu-Prolog.

[8] For example, "Ken ga ame ni fu-ra-reru" (Ken is affected by the rain.)

5.4 Encoding Structural Principle

As mentioned in Section 2, the structural principles of JPSG and HPSG are relations among features of three categories in a local phrase structure. Intuitively, structure principles are encoded as constraints in a phrase structure rule with CHC as:

$$psr(M,D,H); sp_1(M,D,H),\ldots,sp_n(M,D,H).$$

Here, $psr/3$ is a phrase structure rule and each $sp_i/3$ $(i = 1\ldots n)$ indicates structure principles. In cu-Prolog, these structural principles are evaluated flexibly with heuristics. In Prolog, however, the above phrase structure rule is represented as:

$$psr(M,D,H) :- sp_1(M,D,H),\ldots,sp_n(M,D,H).$$

Each principle is always evaluated sequentially. Prolog, therefore, is not well-suited to constraint based grammars because it is impossible to stipulate in advance which kind of linguistic constraints must be processed, and in what order.

The following example demonstrates the *foot feature principle* of JPSG[5]:

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

By defining `ffp/3` as follows, the principle is represented as constraint `ffp(M,D,H)`.

```
ffp({foot/MF},{foot/DF},{foot/HF}) :- union(DF,HF,MF).
```

Figure 2 is an example of the JPSG parser in cu-PrologIII that parses the ambiguous Japanese sentence "Ken ga ai suru"(Ken loves.) For an ambiguous sentence, the parser returns the corresponding feature structure with constraints.

6 Concluding Remarks

This paper outlined cu-Prolog, then covered the treatment of DFS and parsing JPSG to realize constraint-based NLP.

We would like to stress that every feature mentioned in this paper was uniformly processed in the same framework as a constraint transformation. In comparison with many conventional approaches, our approaches, including Hasida's DP (Dependency Propagation/Dynamic constraint Processing) [6, 7], provide a far more general and flexible framework for NLP.

DP is an extension framework of the constraint unification that treats clausal-form logic programs by constraint transformation. DP adopts concepts of the dynamics; potential energy is defined to programs and inferences are controlled so as to minimize the energy[7]. Compared with DP, cu-Prolog mixes procedural programming and constraints by CHC, and can be seen as being a more practical approach.

Subsequently, we hope to add constraint hierarchy to cu-Prolog. In the current framework, every constraint is equally satisfied, such that if the constraint is over-constrained the transformation fails. However, constraints occurring in a grammar description sometimes contradict each other and have preferences or hierarchies. Such cases would easily occur if we were to consider various heterogeneous linguistic constraints.

For example, [14] postulates two constraints, semantic and syntactic preferences, in WH-questions such as:

- (Sementic preference): The preference of indirect object (IO) taken by the verb "give" is *higher_animate(people)* > *animate* > *inanimate*.
- (Syntactic preference):
  - prefer: NEXT-as-IO: The noun next to the verb is IO.
  - not-prefer: WH-comp-as-IO: The complement of the WH-clause is IO.

Cost-based abduction[9] adds numerical costs and weights to literals to derive the least cost abduction as the best explanation.

What is the framework for treating such constraint relaxation or optimization? A cue in the field of CLP is a hierarchical constraint logic programming (HCLP) [1] proposed as an extension of CLP. In HCLP, every constraint is labeled with its strength (hierarchy), with constraints being processed from the stronger to the weaker ones. HCLP also provides comparators, that may differ in the application, to compare the appropriateness of solutions.

Acknowledgment

The author thanks Hidetosi Sirai of Chukyo University for his cooperation in implementing cu-Prolog. Thanks are also due to Kazumasa Yokota, Kôiti Hasida, Satoshi Tojo, Hideki Yasukawa, and other ICOT researchers for their valuable comments. For their contributions to JPSG, the author thanks Prof. Takao Gunji and other PSG working group members.

References

[1] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proc. of 6th International Conference of Logic Programming*, pages 149–164, 1989.

[2] B. Carpenter, C. Pollard, and A. Franz. The Specification and Implementation of Constraint-Based Unification Grammar. In *Proc. of 2nd International Workshop on Parsing Technologies*, pages 143–153. Sigparse ACL, February 1991.

[3] J. Dörre and A. Eisele. Feature Logic with Disjunctive Unification. In *Proc. of COLING-90 Vol.2*, pages 100–105, August 1990.

[4] A. Eisele and J. Dörre. Unification of Disjunctive Feature Descriptions. In *Proc. of 26th ACL Annual Meeting*, pages 286–294, June 1988.

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

[6] K. Hasida. Common Heuristics for Parsing, Generation, and Whatever. In T. Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*. Kluwer Academic Publishers, 1991.

[7] K. Hasida, K. Nagao, and T. Miyata. Joint Utterance: Intrasentential Speaker/Hearer Switch as an Emergent Phenomenon. In *Proc. of IJCAI93*, pages 1193–1199, Chambery, 1993.

[8] K. Hasida and H. Sirai. Jyokentsuki Tan'itsu-ka (Conditioned Unification). *Computer Software*, 3(4):28–38, 1986. (in Japanese).

[9] J. R. Hobbs, M. Stickel, P. Martin, and D. Edwards. Interpretation as Abduction. In *Proc. of the 26th ACL Annual Meeting*, pages 95–103, 1988.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM POPL Conference*, pages 111–119, Munich, 1987.

[11] R. T. Kasper. A Unification Method for Disjunctive Feature Descriptions. In *25th ACL Annual Meeting*, pages 235–242, July 1987.

[12] R. T. Kasper and W. C. Rounds. A Logical Semantics for Feature Structure. In *Proc. of 24th ACL Annual Meeting*, pages 257–266, 1986.

[13] M. Kay. Parsing in Functional Unification Grammar. In D. R. Dowty, L. Karttunen, and A. M. Zwicky, editors, *Natural Language Parsing*, chapter 7, pages 251–278. Cambridge university press, 1985.

[14] M. P. Marcus. *A Theory of Syntactic Recognition for Natural Language*. MIT Press. Cambridge:Mass. 1980.

[15] K. Mukai. Partially Specified Term in Logic Programming for Linguistic Analysis. In *Proc. of the International Conference of Fifth Generation Computer Systems*, pages 479–488. ICOT. OHMSHA, Springer-Verlag. 1988.

[16] C. Pollard and I. A. Sag. *Information-Based Syntax and Semantics. Vol.1 Fundamentals.* CSLI Lecture Notes Series No.13. Stanford:CSLI, 1987.

[17] C. Pollard and I. A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, 1993. (to appear).

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

[19] S. M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, A Bradford Book, 1992.

[20] H. Sirai. A Guide to MacCUP. unpublished, 1991. (available by anonymous FTP from csli.stanford.edu (pub/MacCup)).

[21] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In *Proc. of 2nd International Conference on Logic Programming*, pages 127–137, 1983.

[22] Y. Tomioka. Computability of Modularization of Constaraints. *Computer Software*, 9(6):58–68, 1992. (in Japanese).

[23] H. Tsuda. Disjunctive feature structure in cu-prolog. In *8th Conf. Proc. Japan Soc. Softw. Sc. Japan.*, pages 505–508, 1991. (in Japanese).

[24] H. Tsuda, K. Hasida, and H. Sirai. cu-Prolog and its application to a JPSG parser. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Logic Programming '89*, pages 134–143. Springer-Verlag LNAI-485, 1989.

[25] H. Tsuda, K. Hasida, and H. Sirai. JPSG Parser on Constraint Logic Programming. In *Proc. of 4th ACL European Chapter*, pages 95–102, 1989.

```
%% definition of the unconditional conjuncts (user's input)
cc1({voice/passive,trans/trans,subj/X,goal/X}).
cc1({voice/active, subj/X,actor/X}).
cc2({trans/intrans, actor/{person/third}}).
cc2({trans/trans, goal/{person/third}}).
cc3({numb/sing, subj/{numb/sing}}).
cc3({numb/pl, subj/{numb/pl}}).

%% Disjunctive Feature Structure unification (user's input)
@ U={rank/clause, subj/{case/nom}},cc1(U),cc2(U),cc3(U),
    U={subj/{lex/yall,person/second,numb/pl}}.

%% answer: equivalent constraint
solution = c0(U_0, {subj/{case/nom}, rank/clause}, {subj/{person/second, numb/pl, lex/yall}})

%% definitions of a new predicate (c0)
c0(_p1, _p1, _p1) :- cc2(_p1), cc1(_p1);
   _p1={subj/{person/second, numb/pl, case/nom, lex/yall}, numb/pl, rank/clause}.

CPU time = 0.150 sec (Constraints Handling = 0.000 sec)

>:-c0(X,_,_).           % solve the new constraint
success.                  % X is the final answer of the unification.
  X = {voice/active, trans/trans, subj/{person/second, numb/pl, case/nom, lex/yall},
     goal/{person/third}, actor/{person/second, numb/pl, case/nom, lex/yall},
     numb/pl, rank/clause};
```

This is a demonstration of DFS unification using the constraint transformer. The first 7 lines define disjunctions in (5) in terms of user-defined predicates. In cu-PrologIII, a constraint that follows "@" at the top level is transformed into modular one. In this case, it specifies the unification between (5) and (4). To this input, the constraint transformer returns equivalent modular constraint and definition clauses of newly defined predicates. The result of the unification, which is a non-disjunctive FS in this case, is given as the binding of X in the last 3 lines.

Figure 1: DFS unification

```
_:-p([ken,ga,ai,suru]).        % user's input of ''Ken ga ai-suru.''

       %%% parse tree
{sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v}, sc/V1_2024,
 refl/[], slash/V3_2026, psl/[], ajn/[], ajc/[]}---[suff_p]
 |
 |--{sem/[love,V7_2030,V6_2029], core/{pos/v}, sc/V0_2023, refl/[],
     slash/V2_2025, psl/[], ajn/[], ajc/[]}---[subcat_p]
 |    |
 |    |--{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],
          psl/[], ajn/[], ajc/[]}---[adjacent_p]
 |    |    |
 |    |    |--{sem/ken, core/{form/n, pos/n}, sc/[], refl/[], slash/[],
               psl/[], ajn/[], ajc/[]}---[ken]
 |    |    |
 |    |    |__{sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[],psl/[], ajn/[],
              ajc/[{sem/ken, core/{pos/n}, sc/[], refl/ReflAC_70}]}---[ga]
 |    |
 |    |__{sem/[love,V7_2030,V6_2029], core/{form/vs2, pos/v}}---[ai]
 |
 |__{sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v}, sc/[], refl/[],
     slash/[], psl/[], ajn/[], ajc/[{sem/[love,V7_2030,V6_2029],
     core/{form/vs2, pos/v}, sc/[], refl/ReflAC_1493}]}---[suru]

category= {sem/[love,V7_2030,V6_2029], core/{form/Form_1381, pos/v},
      sc/V1_2024, refl/[], slash/V3_2026, psl/[], ajn/[], ajc/[]}  %category

constraint= c40(V0_2023, V1_2024, V2_2025, V3_2026, V4_2027, V5_2028,
       {sem/ken, core/{form/ga, pos/p}, sc/[], refl/[], slash/[], psl/[],
       ajn/[], ajc/[]}, V6_2029, {sem/V6_2029, core/{form/wo, pos/p}}, V7_2030,
       {sem/V7_2030, core/{form/ga, pos/p}}),
   syu_ren(Form_1381)  %constraint about the category
true.
CPU time = 2.217 sec (Constraints Handling = 1.950 sec)

_:-c40(V1, _, _, V3, _, _,_, V6,_ , V7,_).        %solve constraint
  V1 = []  V3 = [{sem/V0_4}]  V6 = V0_4  V7 = ken;      % solution 1
  V1 = [{sem/V0_4, core/{form/wo, pos/p}}]  V3 = []  V6 = V0_4  V7 = ken;   % solution 2
no.
CPU time = 0.017 sec (Constraints Handling = 0.000 sec)
```

The parsing of "Ken ga ai-suru" that has two meanings: "Ken loves (someone)" or "(someone) whom Ken loves." The parser draws a corresponding parse tree and returns the category of the top node with constraints. In this example, the ambiguity of the sentence is indicated in the two solutions of the constraint c40.

Figure 2: JPSG parser: disambiguation