PARSING WITHOUT PARSER

HASIDA, Kôiti TSUDA, Hiroshi*
Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN
Tel: +81-3-3456-3069
E-mail: hasida@icot.or.jp tsuda@icot.or.jp

ABSTRACT

In the domain of artificial intelligence, the pattern of information flow varies drastically from one context to another. To capture this diversity of information flow, a natural-language processing (NLP) system should consist of modules of constraints and one general constraint solver to process all of them; there should be no specialized procedure module such as a parser and a generator.

This paper presents how to implement such a constraint-based approach to NLP. *Dependency Propagation* (*DP*) is a constraint solver which transforms the program (=constraint) represented in terms of logic programs. *Constraint Unification* (*CU*) is a unification method incorporating DP. *cu-Prolog* is an extended Prolog which employs CU instead of the standard unification.

cu-Prolog can treat some lexical and grammatical knowledge as constraints on the structure of grammatical categories, enabling a very straightforward implementation of a parser using constraint-based grammars. By extending DP, one can deal efficiently with phrase structures in terms of constraints. Computation on category structures and phrase structures are naturally integrated in an extended DP. The computation strategies to do all this are totally attributed to a very abstract, task-independent principle: prefer computation using denser information. Efficient parsing is hence possible without any parser.

1 Introduction

The information-processing capacity of a cognitive agent is severely limited, whereas the world in which it finds itself contains a vast amount of information which might be relevant to its survival. A cognitive agent is thus destined to face *partiality of information*. That is, information processing by a cognitive agent is limited to a very small part of the potentially relevant information. In the domain of artificial intelligence in general and natural language processing in particular, therefore, the pattern of information flow varies very drastically from one context to another. This is necessary in order for a cognitive agent to have chance of access to the entire domain of potentially relevant information across various different contexts.

Due to this diversity of information flow, it is practically impossible to stipulate which pieces of information to process in which order. Consider the case of comprehension of natural language sentences, for instance. Parts of phonological information might be missing due to noise, and it may well be impossible to predict which part would be missing. Similarly, parts of syntactic information could be insufficient, giving rise to syntactic ambiguity. Semantic information also would be partially abundant or missing due to familiarity or ignorance to the topic, and so on. It is therefore utterly implausible to suppose that all phonological information is processed prior to syntactic information, or that syntactic information is processed before semantic information.

Accordingly, it is not at all a promising approach to AI or NLP to stipulate information flow totally, as procedural programs do. In particular, a hierarchical architecture consisting of modules of procedures fails to capture very complex, multi-directional, information flow in the domains such as NLP, because procedures stipulate what is input and what is output, severely restricting the global information flow across the entire system. This is what happens in the prevalent architecture of NLP systems consisting of a sequence of procedure modules such as, say, syntactic analyzer, semantic analyzer, pragmatic analyzer, generation planner, and surface generator.

The design of AI systems should abstract away information flow in accordance with its diversity.[1] This is where constraint paradigm [14] comes in. Since constraint, or declarative program, does not stipulate pro-

*The order is not significant.

[1]Of course, there are some aspects of cognitive process where information flow is rather restricted. Typical examples are found in low-level aspects of perception and motor control. Information flow may be stipulated to some adequate extent in the design of those subsystems. Nevertheless, diversity of information flow must be captured across different dimensions even in these cases, as is indicated by R. Brooks [1]; In his robot, although information flow in each module may be regarded as uni-directional and there is only a little interaction between different modules, input information is not restricted to flow all the way through every module before output information is tailored.

cessing order, it does not restrict information flow so severely as procedures do, and thus can capture the diversity of information flow.[2]

In the constraint paradigm, a NLP system involves modules of linguistic (syntactic, semantic, pragmatic, and so on.) and extralinguistic constraints. Whether there are different constraint solvers for different modules of constraints is not a light question, but we strongly suspect the answer is no. If yes, the communication between different modules would be too cumbersome to allow the massive interaction required in NLP. For instance, it would not be a very good idea to have a constraint solver specialized for processing syntactic information. Thus we employ a radical constraint-based viewpoint: just one very general constraint solver deals with all the different constraints, giving rise to diverse communication across them.[3] The task of NLP is hence divided into modules of constraints rather than modules of procedures as has been traditionally done. As a matter of course, a NLP system should include no parser.

In the rest of the paper, we will concentrate on parsing. Efficient parsing will be shown to emerge from our constraint solver, which is a general constraint transformation method employing several heuristics derived from the following very abstract, task-independent principle:

(1) Prefer computation using denser information.

That is, efficient parsing is attributed to this principle. This is regarded as an impressive demonstration of the feasibility of our constraint-based approach, because parsing is almost the only subproblem of NLP where there are endorsed efficient algorithms, mainly for dealing with phrase structures.

Our constraint solver, called *Dependency Propagation* [8, 7], deals with constraints in a combinatorial domain, unlike the constraint solvers embedded in most constraint logic programming (CLP) languages [2, 3, 9]. Section 2 describes how to parse ambiguous sentences with a CLP language, called *cu-Prolog*, which embeds an early version of DP. Although the ambiguity treated in Section 2 concerns only the structures of grammatical categories, Section 3 applies DP itself to parsing phrase-structure, typically formulated in terms of context-free grammars. It will be shown that efficient parsing procedures such as Earley's algorithm simply emerge from general processing strategies employed in a revised version of DP. Section 4 demonstrates that both types of information, about category structures and phrase structures, are processed efficiently in a naturally integrated manner by very general heuristics. Finally, Section 5 concludes the paper.

[2]Constraint is not the only approach to diversity of information flow. For instance, blackboard architecture is also regarded as aiming at the same thing. Coroutine implemented in languages such as CONNIVER [13] is another example. The reason why we employ constraint paradigm is twofold. First, it comes with intuitive declarative semantics. Second, it implements the diversity of information flow at finer-grained levels than might be captured in the other approaches.

[3]Thus we consider that General Problem Solver was basically on the right track. Its alleged failure was simply due to the immaturity of programming technologies.

2 Processing Category Structure

In [18], we introduced a symbolic CLP language cu-Prolog and showed how it applies to parsing based on JPSG (Japanese Phrase Structure Grammar) [5]. By treating grammatical principles and ambiguity concerning polysemy or homonymy straightforwardly in terms of constraints, syntactic, semantic and other types of ambiguity are processed in an integrated manner by *Constraint Unification* (*CU*). CU is the unifier employed in cu-Prolog, and is roughly regarded as the standard unification plus DP. cu-Prolog deals with various constraints on the structures of grammatical categories, without any special programming besides the encoding of the relevant constraints.

2.1 Dependency Propagation

For the sake of expository simplification, in this paper we restrict ourselves to Horn clauses, although DP is not actually so limited.

Dependency triggers constraint transformation in DP. Two occurrences of the same variable in a clause constitutes a *dependency* when both occurrences do not occupy any *vacuous argument place*. An argument place of an atomic formula is said to be *vacuous* when a variable filling that argument place is never instantiated by evaluating that atomic formula.

For instance, the first argument place of predicate `member` defined below is vacuous.

(2) a. `member(E,[E|_]).`
    b. `member(E,[_|S]) :- member(E,S).`

In the following clauses, (3) has no dependency, and (4) has a dependency because it is equivalent to (5).

(3) `:-member(a,X).`

(4) `:-member(X,[a,b,c]).`

(5) `:-member(X,Y),Y=[a,b,c].`

In DP, computation proceeds so as to eliminate dependency. Note that this is a more general control schema than Earley deduction [10], which executes the body of each clause in the fixed left-to-right order.

Basically, *fusion* replaces one or more literals with another, so as to eliminate dependency. Fusion is a sort of unfold/fold transformation for logic programs [15]. For example, `member(X,[a,b])` is replaced by `c0(X)`, where `c0` is a new predicate defined as follows.

(6) `c0(a).`
    `c0(b).`

That is, two atomic formulas, `member(X,Y)` and `Y=[a,b]`[4] have been fused to one atomic formula `c0(X)`.

The principle (1) provides us some heuristics for controlling fusion. For example, the elimination of dependency involving a variable binding, as in `p(a,X)`, should have higher priority than the elimination of dependency between two ordinary atomic formulas, as in `p(X), q(X)`. We will discuss heuristics further along this line later.

2.2 cu-Prolog

A program of cu-Prolog is a set of Constraint-Added Horn Clauses (CAHCs), A CAHC is a Horn Clause followed by constraints:

$$\overbrace{H}^{Head} :- \overbrace{B_1, B_2, \cdots, B_n}^{Body} ; \overbrace{C_1, C_2, \cdots, C_m}^{Constraint}.$$

The prolog part (the head plus the body) of a CAHC is processed procedurally just as in Prolog, whereas the constraint part is dynamically transformed with a sort of unfold/fold transformation during the execution of the former part. The following is the inference rule of cu-Prolog:

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

A and A' are atomic formulas. $\mathbf{K}$, $\mathbf{L}$, $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{C}'$ are sequences of atomic formulas. $mgu(A, A')$ is the most general unifier between A and A'.

$dp(\mathbf{C})$ is a modular constraint that is equivalent to $\mathbf{C}$. If $\mathbf{C}$ is inconsistent, the application of the above inference rule fails because $dp(\mathbf{C})$ does not exist.

The following holds if $\mathbf{C}_i$ and $\mathbf{C}_j$ share no variable:

(7) $dp(\mathbf{C}) = dp(\mathbf{C}_1), \cdots, dp(\mathbf{C}_n)$.

For example,

(8) *dp(*`member(X,[a,b,c]), member(X,[b,c,d]), app(U,V)`*)*

returns a new constraint `c0(X),app(U,V)`, where the definition of `c0` is

```
(9) c0(b).
    c0(c).
```

but

(10) *dp(*`member(X,[a,b,c]),member(X,[k,l,m])`*)*

is not defined.

[4] `Y=[a,b]` may be further regarded as a bundle of five atomic formulas: `Y=[A|Z]`, `A=a`, `Z=[B|W]`, `B=b` and `W=[]`.

2.3 JPSG parser in cu-Prolog

In cu-Prolog, unification-based grammar such as HPSG or JPSG can be implemented naturally by treating the constraints formulated in those theories almost as they are. Figure 1 shows an example session of the JPSG parser when it processes an ambiguous sentence.[5] Below we discuss two examples of CAHC in the JPSG parser in cu-Prolog [18].

The first example concerns how to pack lexical ambiguity. The following is the lexical entry of a Japanese polysemic noun "hasi" that means bridge, chopsticks, or edge depending on contexts.

```
(11) lexicon(hasi,[...sem(TYPE,OBJ)]);
                  hasi_sem(TYPE,OBJ).
```

and predicate `hasi_sem` is defined as follows.

```
(12) hasi_sem(structure,bridge).
     hasi_sem(tool,chopsticks).
     hasi_sem(place,edge).
```

Constraint `hasi_sem(TYPE,OBJ)` represents various meanings of "hasi" and the ambiguity may be resolved during the parsing process when other constraints are imposed. Because such ambiguity is considered at one time, instead of divided into separate lexical entries, parsing process can be efficient.

In the second example, various feature principles of unification-based grammar are embedded in a phrase structure rule as constraints. The following clause shows the foot feature principle of JPSG: the foot feature value of the mother unifies with the union of those of her daughters.

```
(13) psr([ff(MS)],[ff(LDS)],[ff(RDS)]);
           union(LDS,RDS,MS).
```

`psr(Mother,Left_Daughter,Head)`
is a phrase structure rule followed by the constraint `union(LDS,RDS,MS)` which represents the foot feature principle. `MS`,`LDS`, and `RDS` are foot features of mother, left daughter, and right daughter respectively. The constraint is flexibly processed with the constraint transformation mechanism with a heuristic.

In traditional Prolog, these principles are supposed to be implemented in the following procedural way:

```
(14) psr([ff(MS)],[ff(LDS)],[ff(RDS)]) :-
           union(LDS,RDS,MS).
```

By applying this rule, `union(LDS,RDS,MS)` is executed immediately and parsing process may be inefficient when variables are not well instantiated. Note that it is practically impossible to stipulate the order to process linguistic constraints in advance.

[5] cu-Prolog is implemented in C language on UNIX 4.2/3BSD. This example is on SYMMETRY machine [19].

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

v[Form_675, AJN{Adj_677}, SC{SubCat_679}]:SEM_681---[suff_p]
 |
 |--v[vs2, SC{p[wo]}]:[love,ken,Obj0_415]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p]
 |   |   |
 |   |   |--n[n]:ken---[ken]
 |   |   |
 |   |   |__p[ga, AJA{n[n]}]:ken---[ga]
 |   |
 |   |__v[vs2, SC{p[ga], p[wo]}]:[love,ken,Obj0_415]---[ai]
 |
 |__v[Form_675, AJA{v[vs2,SC{p[wo]}]}, AJN{Adj_677}, SC{SubCat_679}]:SEM_681---[suru]

cat     cat(v, Form_675, [], Adj_677, SubCat_679, SEM_681)
cond     c7(Form_675, SubCat_679, Obj0_415, Adj_677, SEM_681)
True.
CPU time = 0.050 sec

_:-c7(F,SC,_,A,SEM).
  F = syusi  SC = [cat(p, wo, [], [], [], Obj00_30)]  A = []  SEM = [love,ken,Obj00_30];
  F = rentai  SC = []  A = [cat(n, n, [], [], [], inst(Obj00_38, Type3_36))]
  SEM = inst(Obj00_38, [and,Type3_36,[love,ken,Obj00_38]])
no.
CPU time = 0.017 sec
```

This is an example run of JPSG parser in cu-Prolog. The first line is a user's input. "Ken ga ai suru" has two readings: "Ken loves (someone)" and "(someone) whom Ken loves." The parser draws a parse tree and returns information (constraint) on the structure of the top node. In this example, the ambiguity of the sentence is captured as the two solutions of the piece of constraint `c7(F,SC,_,A,SEM)`. The first solution corresponds to "Ken loves (someone)." and the second solution "(someone) whom Ken loves."

Figure 1: Parsing an ambiguous sentence.

3 Processing Phrase Structure

The JPSG parser discussed in Section 2, however, cannot handle ambiguity on phrase structures because the parsing algorithm is written only in the Prolog part of CAHC. This section shows that chart parsing is naturally derived from a very general control strategy of an extended version of DP.

3.1 Context-Free Parsing by Fusion

Let us consider the following extremely simple context-free grammar.

(13) $P \rightarrow a$
$P \rightarrow PP$

The parsing of string $aa \cdots a$ under this grammar may be formulated in terms of the following constraint.[6]

```
(14) :- p(A^0,B), A^0=[a|A^1], ···, A^{n-1}=[a].
     p([a|X],X).
     p(X,Z) :- p(X,Y), p(Y,Z).
```

Note that the double occurrence of `Y` in the last clause does not count as a dependency, because the second argument place of `p` is vacuous. Thus the only dependency to eliminate now is that concerning A^0. Here, we replace $p(A^0,B)$ with $p_0(A^0)$, creating a new predicate p_0.

```
(15) :- p0(B), A^0=[a|A^1], ···, A^{n-1}=[a|A^n].
     p0(A^1).
     p0(Z) :- p(A^0,Y), p(Y,Z).
```

$p(A^0,Y)$ in the last clause is folded and we get

```
(16) p0(Z) :- p0(Y),p(Y,Z).
```

This clause has a dependency concerning `Y`. Then, the parsing process continues.

This transformation process is exempt from the infinite loop due to left recursion, unlike DCG of the standard type, because fusion includes some sort of tabulation technique [16]. If we had $A^0 = [b|A^1]$ instead of $A^0 = [a|A^1]$, for instance, we would have the following instead of (16).

```
(17) :- p0(B), A^0=[b|A^1], ··· .
     p0(Z) :- p0(Y), p(Y,Z).
```

Predicate p_0 lacks a finite proof, and hence is unsatisfiable under the minimal interpretation. This is detected by checking each predicate once when it is first given or created. Infinite loop is avoided in just the same manner also in a more complex case where every input symbol is a well-formed word but they are lined up in a wrong way.

In the current formulation, the computational complexity for processing context-free languages is exponential as to the sentence length. With respect to the above example, suppose that predicate r_i is such that for any assignment to variable X_i, there is a set of assignments to variables X^0 through X^{i-1} under which $r_i(X_i)$ is equivalent to the following:

(18) $p(A^0,X_0) \wedge p(X_0,X_1) \wedge \cdots \wedge p(X_{i-1},X_i)$

p_0 may be regarded as r_0. As it turns out, if a definition clause of r_i is (19) with $j = i$, then r_{i+1} will be created by fusion of $r_i(Y)$ and $p(Y, Z)$, whichever literal might be unfolded, and a definition clause of r_{i+1} will be (19) with $j = i + 1$.

(19) $r_j(Z)$:- $r_j(Y)$, $p(Y,Z)$.

Note that fusion of $r_j(Y)$ or $p(Y, Z)$ with any other literal never takes place, because `Y` is constrained nowhere else and the second argument place of `p` is vacuous. Since (29) is (19) with $j = 0$, it follows from induction on i that r_i is created during the current parsing for $0 < i < n$. A similar reasoning will prove that exponentially many corresponding predicates are created when the basic version of DP as described so far is applied to the following context-free grammar, because there are plural predicate symbols.

(20) $P \rightarrow a$ $\quad Q \rightarrow a$
$P \rightarrow PP$ $\quad Q \rightarrow PP$
$P \rightarrow PQ$ $\quad Q \rightarrow PQ$

3.2 Penetration

To remedy the inefficiency mentioned above, we revise DP by employing a different method of operation for constraint transformation. The new transformation operation we introduce here is *penetration*, which conveys information across clause boundaries.

For instance, consider clause (21), where predicate `p` is defined by (22).

```
(21) :- p(X,Y), p(X,Z), X=f(Y), Y=g(Z).
(22) p(f(A),A) :- s(A).
     p(C,a).
```

The information of `X=f(Y)` makes `X` penetrate through `p(X,Y)`, creating new predicate `q`, as follows:

```
(23) :- q(X,Y), p(X,Z), X=f(Y), Y=g(Z).
     q(X,A) :- X=f(A), q(A).
     q(X,a).
```

As indicated here, the first argument of `q` must always unify with `X` in the first clause, whereas its second argument has no such restriction.

[6] A^i represents the constant list of length `(n-i)` whose elements are "a"s.

In the following discussion, a penetrated variable is written with a superscript like X^1, and called a *transclausal variable*, which roughly corresponds to the global variable of programming languages such as Pascal and C. A transclausal variable may be treated as if it were a constant. Accordingly, a penetrated argument place are omitted for the sake of expository simplification. For instance, (23) may be rephrased as follows:

```
(24) :- q(Y), p(X¹,Z), X¹=f(Y), Y=g(Z).
     q(A) :- X¹=f(A), q(A).
     q(a).
```

Just as fusion, penetration has two cases: *unfolding* and *folding*. An unfolding, such as this case, introduces a new predicate, whereas a folding does not. Binding `X¹=f(B)` in the second clause unifies with `X¹=f(Y)`, and the resulting binding, `X¹=f(Y¹)`, is shared by the first and the second clause:

```
(25) :- q(Y), p(X¹,Z), X¹=f(Y¹), Y¹=g(Z).
     q(¹Y) :- X¹=f(Y¹), q(Y¹).
     q(a).
```

X^1 may penetrate through `p(X¹,Z)` as well:

```
(26) :- q(Y), q(Z), X¹=f(Y¹), Y¹=g(Z).
     q(Y¹) :- X¹=f(Y¹), q(Y¹).
     q(a).
```

This is a folding case of penetration.

A typical pattern of penetration is shown in Figure 2. `p(•,•)`s in the left-hand side of the figure all have the same sign, and those in the right-hand side all have the opposite sign. That is, either `p(•,•)`s in the left are all body literals and those in the right are all head literals, or vice versa. α represents a penetrating variable. We say that this penetration is *downward* in the former case, and *upward* in the latter. The penetration to get (23) and (26) is downward.

For $1 \leq i \leq n$, Ψ_i' is a duplication of Ψ_i except that `p(•,•)` has been replaced by `q(•,•)`. When Φ_i and Ψ_j are the same clause for some i and j, the situation will be more complicated in the sense that the duplication increases not only the right-hand half of the figure but also the left-hand half. The example shown in the next subsection includes some such cases.

As shown in the lower part of the figure, second or later penetration of α through the first argument of `p` is a folding, reusing `q` without introducing a new predicate. Corresponding unfolding and folding must be in the same direction: upward or downward. Otherwise the original combinations of clauses are not preserved. Suppose for instance that α is to penetrate through `p(•,•)` in Ψ_1 at the bottom stage in Figure 2. If we applied folding here, simply replacing this `p(•,•)` with a `q(•,•)`, the resulting configuration would lose the combination of Φ_3 and Ψ_1.

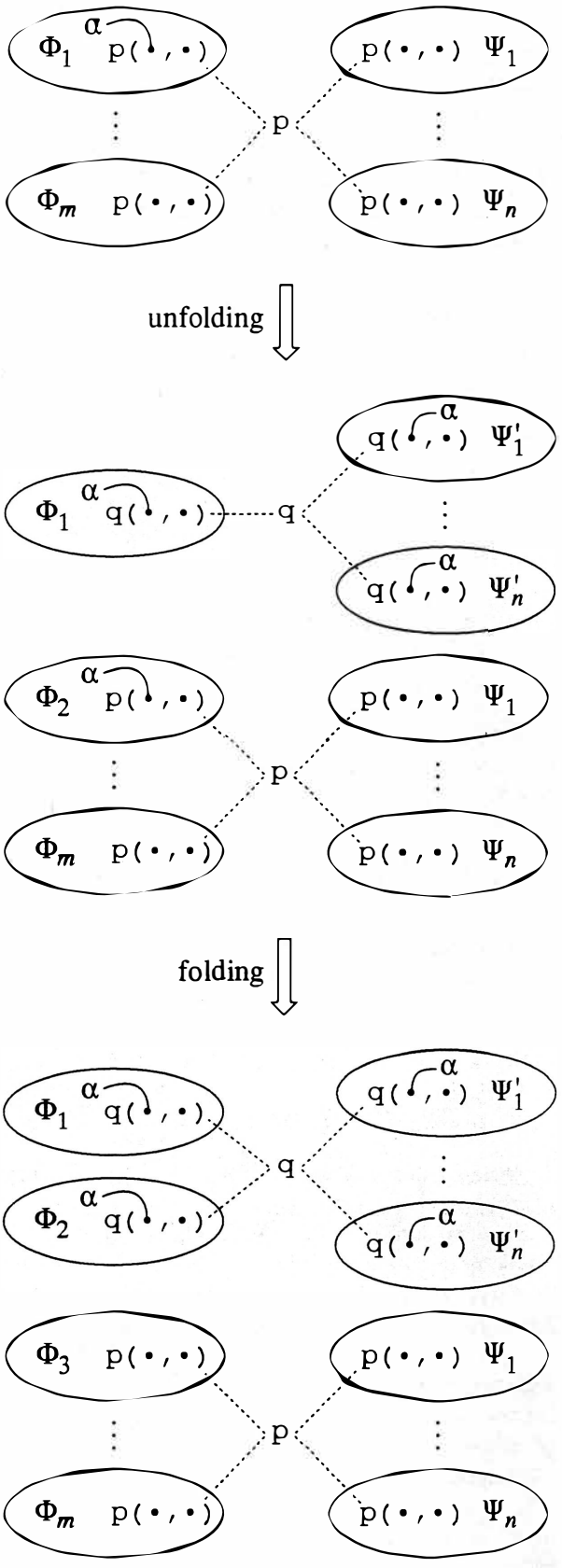

Figure 2: Penetration.

Like fusion, penetration is also triggered by dependency. In penetration, however, dependency may be transclausal. In (26), for instance, the dependency between `Y=g(Z)` and `q(Y)` could trigger penetration. This dependency is transclausal and involves a binding `Y=g(Z)`. In the case of upward penetration, the dependency in question involves a head literal.

To control computation, we must decide which dependency to trigger a penetration into which direction. The general principle (1) suggests the following heuristic in this respect.

(27) a. A dependency encompassing argument places with greater information quantity should more readily trigger a penetration.
b. The argument position with greater information quantity should be penetrated here.

(27b) guarantees that the resulting structure should have more homogeneous information distribution, increasing the entropy of the entire system.

For example, a binding in the top clause is considered to have much more information than bindings in the other clauses, in the sense that the atomic formulas in the top clause should primarily hold; if they do not, then we do not care whether the atomic formulas in the other clauses hold or not. The downward penetration occurring twice in the above example is motivated accordingly, because it is based on the information of `X=f(Y)` in the top clause.

3.3 Emergence of Chart Parsing

Now we demonstrate that Earley's algorithm naturally emerges from penetration controlled by the above heuristic. We consider the simple CFG example (13) again.

```
(13) :- p(A0,B), A0=[a|A1], ···, An-1=[a|An].
     p([a|X],X).
     p(X,Z) :- p(X,Y), p(Y,Z).
```

The following is obtained by downward penetration of A^0 through $p(A^0, B)$, which is unfolded.

```
(28) :- p0(B), A0=[a|A1], ···, An-1=[a|An].
     p0(A1).
     p0(Z) :- p(A0,Y), p(Y,Z).
```

The only relevant dependency here is the one concerning the first argument of $p(A^0, Y)$ in the bottom clause. This literal is hence folded and replaced with $p_0(Y)$, the entire clause being transformed as follows.

```
(29) p0(Z) :- p0(Y), p(Y,Z).
```

Now we have a non-vacuous dependency concerning `Y`, because p_0 says something substantial about the instantiation of its argument. The head $p_0(A^0)$ of the first definition clause of p_0 has transclausal variable A^0 as the argument. Since A^0 has been introduced in the top clause, upward penetration is applied here, so that the first definition clause of p_0 is replaced by $p_{0.1}$ .. and a new definition clause is introduced, as follows.

```
(30) p0.1 .
     p0(Z) :- p0.1, p(A1,Z).
     p0(Z) :- p0(Y), p(Y,Z).
```

The last clause of (28) has been replicated while $p_0(Y)$ therein has been replaced by $p_{0.1}$ plus $Y = A^1$, giving rise to the second clause in (30) above. Note that $p(Y)$ no longer imposes any restriction on the instantiation of `Y`. The dependency concerning `Y` in the third clause here is vacuous and left untouched for the time being.

A problem here, incidentally, is that another top clause as below is created.

```
(31) :- p0.1, B=A1, A0=[a|A1], ···,
                    An-1=[a|An].
```

To avoid two top clauses, we could introduce a new predicate `q` by which to mediate between the top clause and the locus of upward penetration:

```
(32) :- q, A0=[a|A1], ···, An-1=[a|An].
     q :- p0.1, B0=A1.
     q :- p0(B0).
```

Next, $p(A^1, Z)$ in the second clause of (30) is unfolded and a new predicate p_1 is created, A^1 penetrating downwards:

```
(33) p0(Z) :- p0.1, p1(Z).
     p1(A2).
     p1(Z) :- p1(Y), p(Y,Z).
```

Operation proceeds similarly, yielding the clauses below.

```
(34) p1,2 .
     p1(Z) :- p1,2, p2(Z).
     p0.2 :- p0,1, p1,2 .
     p0(Z) :- p0.2, p2(Z).
     p2(Z) :- p2(Y), p(Y,Z).
```

Shown below is what is finally obtained.

```
(35) :- q, A0=[a|A1], ···, An-1=[a|An].
     q :- p0(B0).
     q :- p0.i, B0=Ai. (0 < i ≤ n)
     pi(Z) :- pi.j, pj(Z). (0 ≤ i < j < n)
     pi(Z) :- pi(Y), p(Y,Z). (0 ≤ i < n)
     pi.i+1 . (0 ≤ i < n)
     pi.k :- pi.j, pj.k. (0 ≤ i < j < k < n)
```

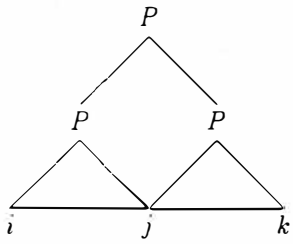

Figure 3: The meaning of $p_{i,k}$:- $p_{i,j}$, $p_{j,k}$.

3.4 Computational Complexity

Part of (35) amounts to a well-formed substring table, as in CYK algorithm, Earley's algorithm [4], chart parser, and so on. For instance, the existence of clause $p_{i,k}$:- $p_{i,j}$, $p_{j,k}$. means that, as illustrated in Fig. 3, the part of the given string from position i to position k has been parsed as having category P and is subdivided at position j into two parts, each having category P. Note that the computational complexity of the above process is $O(n^3)$ in terms of both space and time.

Moreover, the space complexity is reduced to $O(n^2)$ if we delete the literals irrelevant to instantiation of variables, which preserves the semantics of the constraints in the case of Horn programs. That is, the resulting structure would be:

(36) :- q, A^0=[a|A^1], $\cdots$, A^{n-1}=[a|A^n].
q :- $p_0(B^0)$.
q :- B^0=A^i. $(0 < i \leq n)$
p_i(Z) :- p_j(Z). $(0 \leq i < j < n)$
p_i(Z) :- p_i(Y), p(Y,Z). $(0 \leq i < n)$

Some sort of clauses listed here might be generated more than once in general cases where the grammar is less trivial than (13). For example, clause (37) may be derived from both (38) and (39).

(37) s_i(Z) :- vp_j(Z).

(38) s(X,Z) :- np(X,Y), vp(Y,Z).

(39) s(X,Z) :- np(X,Y), adv(Y,U), vp(Y,Z).

If (37) is generated twice, then of course we are able to collapse the two instances to one, so that the space complexity should be $O(n^2)$. Needless to say, this collapsing operation is totally domain-independent in its nature.

The process illustrated above corresponds best to Earley's algorithm. Our procedure may be generalized to employ more bottom-up control, so that the resulting process should be regarded as chart parsing in general, including left-corner parsing, and so on.

4 Integrated Processing

Section 2 treats linguistic constraints on category structures as constraint transformation, and Section 3 processed linguistic constraints on phrase structures. This section discusses how to handle various types of constraints mentioned in the previous two sections. Some heuristics will be needed to determine which constraint to process earlier than the others.

4.1 Heuristics

In the following discussion, we consider two types of linguistic constraints: constraints on category structure and those on phrase structure. For simplicity, the former constraints are represented only by predicate c, and the latter p. Accordingly, we introduce two types of dependency: *inter-dependency* and *intra-dependency*. Inter-dependency is a double occurrence of a variable in both types of constraints, such as X in p(X), c(X,Y). Intra-dependency arises with non-variable arguments or a variable that occurs only in one type of constraints such as c(a,X) or Y in c(a,Y),c(Y,b).

By applying the general heuristic (27) to this domain, we get the following heuristic:

- Eliminate intra-dependencies earlier than inter-dependencies.
- Eliminate intra-dependencies in category structure earlier than those in phrase structure.
- In eliminating inter-dependencies, the literal that has the fewer OR-alternatives should be unfolded (penetrated downward).

That is, constraints on category structures generally has more information quantity than those on phrase-structure, because the former are called by the latter. In the case of a dependency between two argument places of ordinary atomic formulas, moreover, penetration operation should take place at the one that has fewer alternatives of unfolding, because it is supposed to have more information quantity.

4.2 Example

The following is an ambiguous context free grammar that parses "I see a man with a telescope."

(40) $VP \rightarrow V\ NP$
$VP \rightarrow VP\ PP$
$NP \rightarrow NP\ NP$
$V \rightarrow$ see
$NP \rightarrow$ a man
$PP \rightarrow$ with a telescope

(41) is a parsing program in terms of this grammar.

```
(41) p(X,Z,C) :- p(X,Y,LC),p(Y,Z,RC),
                 c(LC,RC,C).
     c(v,np,vp).
     c(np,pp,np).
     c(vp,pp,vp).
     p([see|W],W,v).
     p([a,man|W],W,np).
     p([with,a,telescope|W],W,pp).
```

Predicate p represents phrase structure constraint and predicate c represents constraint on category structure.[7]

```
(42) :-p(A0,B,C), A0=[see|A1], A1=[a,man|A2],
                  A2=[with,a,telescope|A3],A3=[ ].
```

(42) is a question clause. This example shows that two meanings of "I see a man with a telescope" are derived from this program by the constraint transformation with the heuristic mentioned previously.

The dependency to be processed is in terms of A^0 in (42) because LC and RC in (40) do not have dependencies on account of vacuous argument places. Then, apply downward penetration in terms of A^0 to (42). $p_0(B,C)$ is equivalent to $p(A^0,B,C)$.

```
(43) :-p0(B,C).
(44) p0(A1,v).
(45) p0(B,C) :- p(A0,Y,LC),p(Y,B,RC),
                c(LC,RC,C).
```

The first body literal of (45) can be folded and we get

```
(46) p0(B,Cat) :- p0(Y,LC),p(Y,B,RC),
                  c(LC,RC,Cat).
```

Apply upward penetration to (44). Here $p_{0,1}$ is equivalent to $p_0(A^1,v)$.

```
(47) :-p0,1.
(48) p0,1.
(49) p0(B,Cat):-p0,1,p(A1,B,RC),c(v,RC,Cat).
```

Unfold the category constraint of (49).[8]

[7] From unification-based point of view, suppose each category has the form [pos/X] and c() represents the pos feature principle:

> The combination of the values of pos feature of mother, left daughter, and right daughter category is (vp,n,np), (np,np,pp), or (vp,vp,pp).

[8] Let c_0(Cat) be c(v,RC,Cat) and you apply downward penetration to (49), obtaining

$$p_0(B,Cat) :- p_{0,1},p(A^1,B,RC^1),c_0(Cat).$$

However, c_0 has only one definition clause:

$$c_0(vp):-RC^1=np.$$

So c_0 is reduced and you get (50).

```
(50) p0(B,vp):-p0,1,p(A1,B,np).
```

Now the remaining clauses are (43), (47), (48), (46) and (50). Apply downward penetration in terms of A^1 to (50). $p_1(B)$ is equivalent to $p(A^1,B,np)$.

```
(51) p0(B,vp):-p0,1,p1(B).
(52) p1(A2).
(53) p1(Z):-p(A1,Y,np),p(Y,Z,RC),c(np,RC,np).
```

Unfold the category structure constraint of (53).

```
(54) p1(Z):-p(A1,Y,np),p(Y,Z,pp).
```

The first body of (54) can be folded and we get

```
(55) p1(Z):-p1(Y),p(Y,Z,pp).
```

Upward penetration in (52). $p_{1,2}=p_1(A^2)$

```
(56) p0(A2,vp):-p0,1,p1,2.
(57) p1(Z):-p1,2,p1(A2,Z,pp).
(58) p1,2.
```

Upward penetration in (56). $p_{0,2}$ is equivalent to $p_0(A^2,vp)$.

```
(59) p0(B,Cat):-p0,2,p(A2,B,RC),c(vp,RC,Cat).
(60) p0,2:-p0,1,p1,2.
```

Unfold the category constraint of (59).

```
(61) p0(B,vp):-p0,2,p(A2,B,pp).
```

Here, the remaining clauses are (43), (47), (48), (58), (60), (46), (51), (55), (57) and (61). Apply downward penetration of A^2 in (61). $p_2(B)$ is equivalent to $p(A^2,B,pp)$.

```
(62) p0(B,vp):-p0,2,p2(B).
(63) p2(A3).
(64) p2(B):-p(A2,Y,LC),p(Y,B,RC),c(LC,RC,pp).
```

Unfolding of the category constraint of (64) fails. Fold (57).

```
(65) p1(Z):-p1,2,p2(Z).
```

Upward penetration in (63). $p_{2,3}$ is equivalent to $p_2(A^3)$.

```
(66) p0(A3,vp):-p0,2,p2,3.
(67) p2,3.
(68) p1(A3):-p1,2,p2,3.
```

Upward penetration in (66). $p_{0,3} \equiv p_0(A^3,vp)$.

```
(69) p0(B,Cat):-p0,3,p(A3,B,RC),c(vp,RC,Cat).
(70) p0,3.
```

Unfolding of the category constraint in (69) fails. Upward penetration in (70). $p_{1,3} \equiv p_1(A^3)$.

```
(71) p0(A3,vp):-p0,1,p1,3.
(72) p1(Z):-p1,3,p(A3,Z,pp).
(73) p1,3:-p1,2,p2,3.
```

(66) and (71) represent the two readings of "see a man with a telescope."

5 Concluding Remarks

In this paper, we have shown that various parsing techniques are subsumed in a general procedure of constraint transformation, whose control heuristic is attributed to an abstract, task-independent principle (1). Thus our conclusion is that no parser at all is needed in natural language processing, It is both desirable, as is discussed first in the paper, and possible, as we have so far demonstrated, for an NLP system to have no particular module for parsing sentences, just as a car has no particular part for driving towards the east or turning to the left.

Our approach will capture sentence generation as well, if we employ a more adequate control heuristic, which could also be derived from (1). In this connection, Shieber [12], among others, has also proposed a computational architecture by which to unify sentence parsing and generation, but his method is primarily specific to phrase-structure synthesis. A significant merit of our approach is that, as shown above, it is not in any way restricted to parsing or generation of context-free languages. Also, no additional mechanism is required to extend the underlying grammatical formalism so that grammatical categories may be complex feature bundles, as is the case with GPSG, LFG, HPSG, and so on.

At any rate, heuristics play the most important role in our approach. As this paper only gave an intuitive rationale on some heuristics in terms of information quantity, more formal account of them is yet to be worked out. A promising direction seems to be to define some sort of potential energy over constraints, which should capture information density, providing not only processing control but also preference of conclusion. Introducing hierarchies in the constraint is regarded as along the same line.

References

[1] Brooks, R. (1988) *Intelligence without Representation*, technical report, AI Laboratory, MIT.

[2] Colmerauer, A. (1987) *An Introduction to Prolog III*, unpublished manuscript.

[3] Dincbas, M., Simonis, H. and Van Hentenryck, P. (1988) 'Solving a Cutting-Stock Problem in Constraint Logic Programming,' *Proceedings of the 5th International Conference of Logic Programming*, pp. 42–58.

[4] Earley, J. (1970) 'An Efficient Context-Free Parsing Algorithm,' *Communications of ACM*, Vol. 13, pp. 94–102.

[5] Gunji, T. (1986) 'Japanese Phrase Structure Grammar', Reidel, Dordrecht,1986.

[6] Hasida, K. (1986) 'Conditioned Unification for Natural Language Processing,' *Proceedings of the 11th COLING*.

[7] Hasida, K. and Ishizaki, S. (1987) 'Dependency Propagation: A Unified Theory of Sentence Comprehension and Generation,' *Proceedings of the 10th IJCAI*, pp. 664–670.

[8] Hasida, K. (1990) 'Sentence Processing as Constraint Transformation,' *Proceedings of ECAI'90*.

[9] Jaffar, J. and Lassez, J. (1988) 'From Unification to Constraints,' *Logic Programming'87*, Lecture Notes in Computer Science, No. 315, pp. 1–18.

[10] Pereira, F.C.N. and Warren, D.H.D. (1983) 'Parsing as Deduction,' *Proceedings of ACL'83*, pp. 137–144.

[11] Pollard, C. and Sag, I.A. (1987) *Information-Based Syntax and Semantics. Volume 1*, CSLI Lecture Notes No. 13.

[12] Shieber, S.M. (1988) 'A Uniform Architecture for Parsing and Generation,' *Proceedings of the 12th COLING*, pp. 614–619.

[13] Sussman, G. and McDermott, D.V. (1972) *CONNIVER Reference Manual*, Memo 259, AI Laboratory, MIT.

[14] Sussman, G. and Steele, G., Jr. (1980) 'Constraints – A Language for Expressing Almost-Hierarchical Descriptions,' *Artificial Intelligence*, Vol. 14.

[15] Tamaki, H. and Sato, T. (1983) 'Unfold/Fold Transformation of Logic Programs,' *Proceedings of the Second International Conference on Logic Programming*, pp. 127–138.

[16] Tamaki, H. and Sato, T. (1984) 'OLD Resolution with Tabulation,' *Proceedings of the Third International Conference on Logic Programming*, pp. 84–98.

[17] Tsuda, H. and Hasida, K. (1990) 'Parsing as Constraint Transformation — an extension of cu-Prolog' *Proceedings of the Seoul International Conference on Natural Language Processing*, pp. 325–331.

[18] Tsuda, H., Hasida, K., and Sirai, H. (1989) 'JPSG Parser on Constraint Logic Programming,' *Proceedings of the European Chapter of ACL'89*, pp. 95–102.

[19] Tsuda, H., Hasida, K., Yasukawa,H. and Sirai, H. (1990) 'cu-Prolog V2 system', *ICOT TM-952*.