TR-579

Sentence Processing as Constraint Transformation

by
K. Hasida

July, 1990

© 1990, ICOT

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

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Sentence Processing as Constraint Transformation*

HASIDA, Kôiti
Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108 JAPAN
Tel: +81-3-456-3069. E-mail: hasida@icot.or.jp

Abstract

In most practical rather than idealized problem domains, due to partiality of information, some sort of constraint programming is required, rather than merely preferable as widely deemed. That is, information processing in such domains should be regarded as transformation of constraints.

A method, called Dependency Propagation (DP), of transforming logic program is accordingly proposed as a way of execution in constraint programming rather than compilation in procedural programming. As such, DP is meant to subsume various procedural methods as approximations. DP may first be regarded as a special case of unfold/fold transformation of logic programs, but later is extended by introducing transclausal variables and some related operations for local transformation. Also discussed is a general strategy for choosing the right operation depending upon the local computational context.

Sentence comprehension and production are hypothesized to be transformation of constraints by a general procedure. Standard algorithms for processing context free languages are demonstrated to emerge from sentence processing by DP under context-free grammars. It is also shown that compound structures are also dealt with efficiently along the same line.

1 Introduction

Procedural programming, where we stipulate the direction of information flow — as in, say, assignment `y := f(x)` — postulates that the programmer can restrict in advance the range of possibilities about which pieces of information are accessible and how informative they are compared with each other with respect to the problem under consideration. The task of a programmer here is to design a flow of information so as to investigate the accessible pieces of information in a decreasing order of informativeness.

Such a procedural approach often fails in daily situation as are considered in AI, however, due to *partiality of information*. Consider for instance how we understand natural language sentences. It is previously unknown which pieces of information are accessible and which of them are more informative as to the determination of what the

*Presented at The Logic Programming Conference'90, Tokyo. Also presented at ECAI'90, Stockholm, in a contracted version.

sentence means. Phonological information might be partly missing or unclear because of noise. Syntactic information may or may not contribute very much. It is not so informative when, for example, purely syntactic analysis gives rise to innumerable different parses of the sentence. Semantic information also may or may not be very informative. When you have little prejudice about the semantic content of what you hear, syntactic information would often contribute better than semantic information.

Partiality of information thus gives rise to a too diverse pattern of information flow to stipulate in advance in terms of procedural program. If you employed the procedural paradigm in such cases, you would end up with either a wrecklessly complicated program which fails to reflect the modularity of the relevant knowledge, or a program which is terribly inefficient, failing to implement the diverse flow of information, at the price of modular organization. Consider a computer system to comprehend natural language sentences. To reflect the modularity of the relevant constraints in the program is to make procedure modules for morphological analysis, syntactic analysis, semantic analysis, etc., and then line them up along this order. This would prevent, for instance, the reference to a piece of pragmatic information at the stage of syntactic analysis, contradicting the diversity of information flow.[1]

When we face partiality of information as in the domain of natural language, therefore, we need some sort of constraint programming, in order to implement a diverse flow of information and at the same time to have modular design of the system so that it should be tractable for the designer, be it human or nature. Namely, the information internalized in the system should be regarded as constraints rather than procedures. This conclusion being applied to every intermediate stage of computation, it follows that information processing should be transformation of constraints.

The rest of the paper discusses a method of constraint transformation. The problems we consider are those of artificial intelligence or cognitive science in general and natural language processing in particular. In such problems, one must deal with constraints on *combinatorial objects* such as parse trees and semantic networks. Constraints on numerical domain, finite domain, etc. [1, 2, 9, 11, 14, 18] play at best a minor role here. The transformation method we develop is an extension of unfold/fold transformation of logic program. The vital difference between the preceding works on program transformation and ours, however, is in the purpose. Former approaches have been interested in derivation of efficient programs. In contrast, we are interested in efficient transformation, because to us transformation is primarily execution rather than compilation. So a major objective of ours is to get efficient domain-specific algorithms emerge from domain-independent transformation strategies. In fact, we show that a general control scheme of constraint transformation applied to sentence parsing under context-free grammar gives rise to a process approximated by standard algorithms, and that more complex grammars may also be dealt with along the same line.

[1] Although we are able to write a fairly concise and efficient procedure for sorting without respecting the modularity of the underlying constraints, the difference between sorting and natural language processing is that the constraint relevant to the former is far simpler than in the latter, and the possible patterns of information flow therein is strongly restricted as mentioned earlier.

2 Dependency Propagation

A method of constraint transformation, called *dependency propagation*, is introduced below. The first two subsections describe three local transformation operations: *fusion*, *downward penetration* and *upward penetration*. Discussed after that is which operation to choose in which occasion.

2.1 Basic Method

We represent constraints in terms of first-order logic programs on Herbrand universe, which is a domain of combinatorial objects. A program is a set of program clauses, as in Prolog. A program clause is a disjunction of *atomic constraints*. An atomic constraint is a literal, a possibly negated binding of a variable (such as `X≠f(Y)`), or a possibly negated equality between two variables. The arguments of an atomic constraint are all regarded as possibly instantiated variables. For instance, a binding `X=f(Y, Z)` is an atomic constraint, whose arguments are `X`, `Y` and `Z`, and equality `X=Y` is also an atomic constraint whose arguments are `X` and `Y`.

A program clause looks like:

```
(1) p(X,f(Y)) :- q(X,a), r(Y).
```

DEC-10 Prolog notation being adopted, names beginning with lowercase letters stand for constants (including functions and predicates) and those beginning with uppercase letters variables. Underscore ('_') is also used accordingly. Unlike Prolog, the order of atomic constraints in the body and whether a binding is embedded in a literal stipulated separately are irrelevant, so that (1) is identical to the following.

```
(2) p(X,Z) :- W=a, q(X,W), Z=f(Y), r(Y).
```

Just for the sake of expository simplicity, we assume that a program clause is a Horn clause, but our method is not so restricted essentially. The formulation of a program will be extended in the next section by allowing transclausal variables, so that a program is not exactly of a clausal form in the standard sense of the term.

By a *constraint* we mean a set of program clauses with one distinguished clause called the *top clause*, which is the representative of the entire constraint. Only the top clause is written as lacking the head. The body of a clause is called a *goal*, and the body of the top clause is called the *top goal*. As in Prolog again, the purpose of the computation is to show that the top goal is satisfiable,[2] in the sense that it has a minimal model as a set of ways of assigning finite Herbrand terms to the variables appearing in it. Thus the constraint (i.e., program) is transformed so as to have a more concise representation of the minimal model. A piece of computation is carried out upon a goal, and fails if it is detected to be unsatisfiable. A variable is instantiated when its value is known to be unique in the context of the goal.[3]

[2] More generally put, the purpose here is to tailor an abductive explanation [8] of why the top goal could be true. Due to partiality of information, or lack of knowledge in particular, this involves assumption of some parts of the constraint without any explicit reason to believe them to be true.

[3] Of course there is no algorithm to judge whether a goal is satisfiable, or whether the assignments to variables therein are uniquely determined in that context, for exactly the same reason why there is no corresponding algorithm for Prolog programs.

For instance, goal (3) may be transformed to (4). Here the definition of predicate member is (5). Predicate p, defined by (6), has been created during this transformation.

```
(3) :- member(X,[a,b,c]), member(X,[b,c,d]).
(4) :- p(X).
(5) member(E,[E|_]).
    member(E,[_|S]) :- member(E,S).
(6) p(b).
    p(c).
```

Let us formulate a general procedure for constraint transformation. Our procedure is a sort of unfold/fold transformation [15] controlled so as to yield satisfiable goals, by eliminating *dependency* in the program. We say there is a dependency when one variable appears at two different argument places of two (possibly the same) atomic constraints in the same goal. For example, the following goals each have a dependency.

```
(7) :- p(X), q(X,Y).
(8) :- p(X,X).
(9) :- p(f(X)).
```

(7) has a dependency concerning variable X. So does (8). The dependency in (9) is implicit; this goal is regarded as identical to the following, where the dependency concerning Y is explicit.

```
(10) :- p(Y), Y=f(X).
```

A *vacuous* dependency need not be eliminated. A dependency is said to be vacuous when one of the argument places is *vacuous*. An argument place is vacuous when it imposes no restriction on the instantiation of the variable. Thus, only the first argument place (i.e., the slot X occupies in X=f(Y,Z)) is not vacuous in a binding. The first argument of member as defined by (5) is vacuous, so that the dependency in the goal below is vacuous and hence invokes no computation.

```
(11) :- member(a,S).
```

Note that, as far as Horn clause program under the closed world assumption (CWA) is concerned, a goal is satisfiable if it contains only vacuous dependency and every predicate there is satisfiable. As a special case, the empty goal is satisfiable. A predicate is satisfiable if the body of at least one of its definition clauses is satisfiable. Let us say that a goal is *modular* when it contains only vacuous dependency and every predicate there is modular. We further say that a predicate is *modular* when the body of at least one of its definition clauses is modular.[4] Here we employ the MINIMAL definition of modularity. That is, nothing other than mentioned above is modular, so that modular goals and predicates are satisfiable in the standard sense that they have finite models. Predicate p which has only one definition clause as follows is hence not modular.

[4]In the foregoing literature [4, 5, 17], modularity has been given a stronger definition; i.e., that a predicate is modular when ALL of its definition clauses have a modular body.

(12) `p(X) :- p(X).`

Elimination of dependency is a very good heuristic as long as Horn clause logic programs under CWA are concerned, and is still a useful heuristic in more general cases as well. We pick up clauses whose body contain dependency, and replace them with equivalent clauses without dependency, by *fusing* two atomic constraints into one for each dependency.[5] A new predicate is introduced here. For instance, (13) is transformed that way to (14).

(13) `p(X,Y,Z,W) :- q(X,Y), r(Y,Z), s(W).`

(14) `p(X,Y,Z,W) :- c0(X,Y,Z), s(W).`

Predicate `c0` has been created here, such that `c0(X, Y, Z)` is equivalent to `q(X, Y)` $\wedge$ `r(Y, Z)` for any `X`, `Y` and `Z`. That is, two atomic constraints `q(X, Y)` and `r(Y, Z)` have been fused into `c0(X, Y, Z)`. The definition clauses of `c0` might contain some dependency, probably due to the original dependency just eliminated. In fact, both clauses in `c0`'s definition (15) contain a dependency, provided that `q` is defined by (16).

(15) a. `c0(X,Y,Z) :- Y=f(X), r(Y,Z).`
b. `c0(g(W),Y,Z) :- q(W,Y), r(Y,Z).`

(16) a. `q(X,f(X)).`
b. `q(g(W),Y) :- q(W,Y).`

Transformation is thus recursively applied. (15a) and (15b) are transformed to (17a) and (17b), respectively.

(17) a. `c0(X,f(X),Z) :- c1(X,Z).`
b. `c0(g(W),Y,Z) :- c0(W,Y,Z).`

Note that `c1(X, Z)` has replaced `Y=f(X)` and `r(Y, Z)`. When fusing a literal and a binding, in general, the arguments of the resulting literal are the arguments of the old literal and the binding minus the first argument of the binding. Note also that, in the transformation of the second clause, the body has been folded into `c0(W, Y, Z)`, based on the equivalence mentioned above.

We refer to this method as *dependency propagation* (DP) [5, 6], in the sense that dependency propagate across the constraint as elimination of dependency gives rise to new dependency. A simple version of DP may be formulated as a special case of unfold/fold transformation of logic program, which has been shown to be sound [15] in the sense of the declarative semantics being unchanged.

2.2 Transclausal Variables

In the standard formulation of clausal form, each variable is local to one clause; i.e., universally quantified with that clause as the scope. A typical drawback of this is that no information about variables may be shared among different clauses. This raises computational complexity concerning both time and space: Communication of any such information thus necessitates explicit computation such as resolution or fusion, and the

[5] More than two atomic constraints may be fused at once in some implementation [17].

arity of predicates tends to be large. The method discussed so far here is problematic in this connection, because fusion tends to introduce predicates of greater arity, increasing computational complexity and decreasing readability.

A standard practice by which to settle this issue is to introduce global variables, as is the case with most programming languages. We will refer to what correspond here as *transclausal variables*, which are very similar to the global variables of PASCAL. For instance, program (18) should be interpreted as meaning (19), in the standard notation of first order logic.

```
(18) p(X¹) :- q.
     q :- r.
     r :- s(X¹).
     t(X¹) :- r.
```

$$
\begin{aligned}
(19)\ & \forall X\{p(X) \leftarrow q(X)\} \wedge \\
& \forall X\{q(X) \leftarrow r(X)\} \wedge \\
& \forall X\{r(X) \leftarrow s(X)\} \wedge \\
& \forall X\{t(X) \leftarrow r(X)\}
\end{aligned}
$$

A variable with a superscript, like X^1 in (18), is a transclausal variable; i.e., its different occurrences in different clauses are regarded as sharing the same memory location.

We say an atomic constraint α in a goal is an *ancestor* of atomic constraint β in another goal, when either they are the same or there is a clause γ :- A, such that γ is unifiable with α and the body of A contains an ancestor of β. Nowhere else α is an ancestor of β. Introduced accordingly are quasi-order relations among atomic constraints and clauses. That is, we say an atomic constraint α in a goal is *higher* than another atomic constraint β in another goal, or β is *lower* than α, when the goal to which α belongs contains an ancestor of β. Also, a clause Φ is *higher* than another clause Ψ, or Ψ is lower than Φ, when the body of Φ contains an ancestor of the atomic constraints in the body of Ψ.

Dependencies concerning transclausal variables may be transclausal. That is, a dependency concerning a variable occurs between two atomic constraints which refer to this variable as an argument and belong to two possibly different clauses.[6] Here, there must be a goal containing distinct ancestors of these atomic constraints; i.e., the two atomic constraints must be conjunctive with each other. In the following program, for instance, there occurs a dependency concerning X^1 between two atomic constraints p(X¹, Y) and X=a, because two of their distinct ancestors q(A) and r appear in the body of the first clause.

```
(20) s(A) :- q(A), r.
     q(Y) :- p(X¹,Y).
     r :- X¹=a.
     r :- t(X¹).
```

There is another dependency of the same sort between p(X¹, Y) and t(X¹) for the same reason, but the two occurrences of X¹) in X¹=a and t(X¹) constitute no dependency, because they share all the ancestor literals, such as r in the first clause.

[6] We assume, without loss of generality, that the two arguments of a possibly negated equality must be transclausal variables each of which appears in a clause higher than but not equal to the one containing that equality.

Let us revise DP to support transclausal variables. Several modifications are needed. First, the definition of modularity must be changed, corresponding to the above revised definition of dependency. Second, transformation strategy should be extended so as to deal with transclausal variables. Two operations are proposed below to handle transclausal variables.

For instance, variable Y in (21) may become transclausal when the dependency concerning it is eliminated. For instance, (22) is obtained after

For instance, suppose the dependency concerning Y in (21) is to be eliminated. Here let us make Y *penetrate downwards* through p(X, Y), to obtain (22), where Y has been rendered transclausal as Y^0.

(21) :- p(X,Y), q(Y,Z).

(22) :- c0(X), q(Y^0,Z).

If p is defined by (23), c0 will be defined by (24).

(23) a. p(1,a).
b. p(2,b).

(24) a. c0(1) :- Y^0=a.
b. c0(2) :- Y^0=b.

Now there have arisen two pieces of transclausal dependency concerning Y^0 here. One is between q(Y^0, Z) in (21) and Y^0=a in (24a), and the other is between q(Y^0, Z) in (21) and Y^0=b in (24b). To eliminate these pieces of dependency, let us *lower* q(Y^0, Z) to (24) and get Y^0 penetrating downwards, to obtain the following.

(25) :- c0(X).

(26) a. c0(1) :- Y^0=a, c1(Z).
b. c0(2) :- Y^0=b, c2(Z).

Predicates c1 and c2 are derived from q, in the context of (26a) and (26b), respectively, just as c0 was derived from p in the context of (21). The definition clauses of c1 and c2 probably refer to Y^0. Note here that no new predicate has been introduced from c0. Computation would proceed further so as to eliminate the remaining dependency.

As sketched above, downward penetration engenders transclausal occurrences of variables in lower clauses. *Upward penetration* introduces transclausal occurrences of variables in higher rather than lower clauses. Upward penetration is typically applied when a transclausal variable appears as an argument of the head of a definition clause for the predicate of the literal to be unfolded, as with the second clause in the following, for instance.

(27) a. p(X,Y) :- q(X,Y), r(X).
b. q(A^1,U) :- s(U).
c. q(V,W) :- u(W).

We assume that the literal q(X, Y) in (27a) is about to be processed in order to eliminate the dependency concerning X. Suppose further that the transclausal variable A^1 carries some information somewhere higher than (27a). Upward penetration of A^1 through q(A^1, U) gives rise to the following.

```
(28)  a. p(A¹,Y) :- c(Y), r(A¹).
      b. p(X,Y) :- q(X,Y), r(X).
      c. c(U) :- s(U).
      d. q(V,W) :- u(W).
```

Clause (27b) has been replaced with (28c). We have created (28a) by copying (27a) (−(28b)) and replacing `q(X,Y)` with `c(Y)` plus $X = A^1$ therein. A clause must be created in the same way from every clause whose body contains `q`. Note that there is no remaining dependency within the above part of constraint.

2.3 Choice of Local Operations

In summary, we have discussed three types of local transformation operations: fusion, downward penetration, and upward penetration. For each of them, there are two cases: unfolding and folding. Unfolding introduces a new predicate and folding reuses a predicate previously created by an unfolding of the corresponding type. Whenever possible, folding is preferred to unfolding. If folding cannot apply, an appropriate type of unfolding should be chosen out of the above three so as to maximize the chances of later folding. Let us assume that downward penetration is the default choice here, and consider how to choose between fusion and upward penetration depending upon the local computational context.

Fusion is preferred when the pattern in question – the combination of the two argument places – appears frequently. In some cases, the current constraint might contain many instances of the same pattern. Suppose that the dependency between `p(X, Y)` and `q(Y, Z)` is to be eliminated, for instance. If you find that in the present constraint there are many of the same pattern of dependency between the second argument place of `p` and the first argument place of `q`, then probably fusion is the best choice. Another type of cases where fusion is preferable is that in which the same pattern of dependency is expected to arise later. For instance, the pattern of dependency between `member(A, B)` and `append(B, C, D)` will recur when the two literals are unfolded, where `member` is defined by (5) and `append` is defined as follows.

```
(29) append([],Y,Y).
     append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

This recurrence is detected by looking at the two relevant argument places. In the second definition clause of `member`, the second argument is instantiated to a list and its CDR part is constrained again as the second argument of `member`. The first argument behaves similarly for `append`.

Upward penetration should be fired when, as mentioned regarding (27), an argument of the head of a clause is a transclausal variable introduced somewhere higher. This argument place should be pertaining to a dependency one level higher. Perhaps there would be other occasions where upward penetration is preferable, but we do not investigate them any more here.

3 Parsing and Generation

This section demonstrates that efficient computation comes out of the above transformation strategy. It is shown in particular that standard algorithms for context-free languages

emerge, mainly due to upward penetration. Constraints more complicated than context-free grammars are also processed efficiently by extending the present framework.

3.1 Phrase Structure Synthesis

In natural language processing, sentence parsing is almost the only subdomain for which there are established algorithms; i.e., parsers for context free languages. It is hence a very good demonstration of the efficiency of constraint paradigm, if a process approximated in terms of such an algorithm emerges as a part of constraint transformation.

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

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

Sentence parsing under this grammar may be formulated in terms of the following constraint.

(31)
```
:- p(A^0,B), A^0=[a|A^1], ···, A^{n-1}=[a|A^n].
p([a|X],X).
p(X,Z) :- p(X,Y), p(Y,Z).
```

Note that the dependency concerning Y in the last clause is vacuous, because the second argument place of p is vacuous. Thus the only dependency to eliminate now is the one between $p(A^0, B)$ and $A^0=[a|A^1]$. So we obtain the following by downward penetration of A^0 through $p(A^0, B)$, which is unfolded.

(32)
```
:- p_0(B), A^0=[a|A^1], ···, A^{n-1}=[a|A^n].
p_0(A^1).
p_0(Z) :- p(A^0,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.

(33)
```
p_0(Z) :- p_0(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.

(34)
```
p_{0.1}.
p_0(Z) :- p_{0.1}, p(A^1,Z).
p_0(Z) :- p_0(Y), p(Y,Z).
```

The last clause of (32) 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 (34) 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.

(35) :- $p_{0,1}$, B=A^1, A^0=[a|A^1], ···, A^{n-1}=[a|A^n].

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

(36) :- q, A^0=[a|A^1], ···, A^{n-1}=[a|A^n].
q :- $p_{0,1}$, B^0=A^1.
q :- p_0(B^0).

Next, p(A^1, Z) in the second clause of (34) is unfolded and a new predicate p_1 is created, A^1 penetrating downwards:

(37) p_0(Z) :- $p_{0,1}$, p_1(Z).
p_1(A^2).
p_1(Z) :- p_1(Y), p(Y,Z).

Operation proceeds similarly, yielding the clauses below.

(38) $p_{1,2}$.
p_1(Z) :- $p_{1,2}$, p_2(Z).
$p_{0,2}$:- $p_{0,1}$, $p_{1,2}$.
p_0(Z) :- $p_{0,2}$, p_2(Z).
p_2(Z) :- p_2(Y), p(Y,Z).

Shown below is what is finally obtained.

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

Control issues are irrelevant here, because (39) is obtained by exhaustive elimination of dependency and involves no nondeterminism.

Part of (39) amounts to a well-formed substring table, as in CYK algorithm, Earley's algorithm [3], chart parser, and so on. For instance, the existence of clause $p_{i,k}$:- $p_{i,j}$, $p_{j,k}$. means that the part of the given string from position i to position k has been parsed as having category P and 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:

(40) :- q, A^0=[a|A^1], ···, 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). $(0 \leq i < n)$
$p_{i,j}$. $(0 \leq i < j < n)$

Although this example assumes a very simple grammar, it is quite straightforward to extend it to context-free grammars in general. The process illustrated above corresponds best to Earley's algorithm. Our procedure may be generalized to employ bottom-up control, so that the resulting process should be regarded as chart parsing, left-corner parsing, etc. Note further that incremental parsing is captured simply by considering that n increases as more input words are supplied by the input device.

The above parsing process never fall in an infinite loop due to left recursion, unlike DCG of the standard type. If we had $\mathtt{A^0 = [b|A^1]}$ instead of $\mathtt{A^0 = [a|A^1]}$, for instance, we would have the following instead of (34).

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

Predicate $\mathtt{p_0}$ 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.

Sentence generation is formulated similarly to (31) as follows.

```
(42) :- p(A^0,B), A^0=[_|A^1], A^1=[_|A^2], ···, A^{n-1}=[_|A^n].
     p([a|X],X).
     p(X,Z) :- p(X,Y), p(Y,Z).
```

This also gives rise to (39). Although this is a too simple example, it suggests that incremental generation will be implemented just the same way as incremental parsing, with n increasing as more words are required by the output device.

The polynomial complexity bound mentioned above is due to transclausal variables, or upward penetration in particular. If we restrict ourselves to local variables and fusion operation, the computational complexity for processing context-free languages becomes exponential. Let us consider the former example of parsing again. Starting from (31), we first proceed just the same as above as far as (33), provided that $\mathtt{A}^i$'s are treated as constants; otherwise the computation would be more complicated. Suppose that predicate $\mathtt{r}_i$ is such that $\mathtt{r}_i(\mathtt{X}_i)$ is equivalent to the following for any assignment to variables.

```
(43) p(A^0,X_0) ∧ p(X_0,X_1) ∧ ··· ∧ p(X_{i-1},X_i)
```

$\mathtt{p}_0$ may be regarded as $\mathtt{r}_0$. As easily seen, if a definition clause of $\mathtt{r}_i$ is (44) with $j = i$, then $\mathtt{r}_{i+1}$ will be created by fusion of $\mathtt{r}_i(\mathtt{Y})$ and $\mathtt{p(Y, Z)}$, whichever literal might be unfolded, and a definition clause of $\mathtt{r}_{i+1}$ will be (44) with $j = i + 1$.

```
(44) r_j(Z) :- r_j(Y), p(Y,Z).
```

Note that fusion of $\mathtt{r}_j(\mathtt{Y})$ or $\mathtt{p(Y, Z)}$ with any other literal never takes place, because Y is constrained nowhere else and the second argument place of p is vacuous. Since (33) is (44) with $j = 0$, it follows from induction on i that $\mathtt{r}_i$ is created during the current parsing for $0 < i < n$. A similar reasoning will show that exponentially many corresponding predicates are created when the basic version of DP with only fusion is applied to the following context-free grammar.

(45) $P \rightarrow a \qquad Q \rightarrow a$
$P \rightarrow PP \qquad Q \rightarrow PP$
$P \rightarrow PQ \qquad Q \rightarrow PQ$

The efficient computation shown above has emerged out of a domain-independent control strategy. In this connection, Shieber [12] 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 it is not restricted to parsing or generation in 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, etc., rather than monadic symbols. In such a more general case, the standard parsing algorithms are regarded as partially approximating DP in sentence comprehension.

3.2 Manipulating Compound Structures

The above parsing process is given the surface string as input and synthesizes the rest of the sentence structure. It should hence be a straightforward business to generalize this so as to tailor more complex sentence structures. On the other hand, the generation process mentioned above receives the output request and synthesize the sentence structure entirely. This is unlike the actual sentence generation, in which a semantic or pragmatic structure is largely given and the rest of the sentence structure is worked out.

Below we consider grammars much more serious than context-free, grammatical categories being complex structures instead of monadic symbols. Now dependency may arise concerning not only the surface string but also the internal structures of categories, which could involve semantic or pragmatic structures. More realistic sentence generation is thus implemented, though the following discussion does not make any explicit mention about parsing or generation.

Mainly three issues are considered here. First, it is shown that folding takes place often enough, which means that local pieces of computation are largely shared, contributing to the efficiency of the entire task. Second, in this connection, DP will be shown to have a nice termination property regarding sentence processing, in the sense that it avoids infinite loops in many cases. Last, we consider how to control the order of computation or partial computation, and discuss that DP may be extended so as to naturally incorporate some control strategies proposed elsewhere.

Take for instance the following top clause.

```
(46) :- c(category(H,[]),STR0,STR1), ···.
```

Now we are going to eliminate the dependency. Suppose the definition clauses for `c` include the clause below.

```
(47) c(category(HEAD,CATLIST),X,Z) :-
          c(CAT,X,Y),
          c(category(HEAD,[CAT|CATLIST]),Y,Z).
```

`CATLIST` may be regarded as the value of **subcat** feature or **slash** feature in HPSG [10]. Similarly, `HEAD` is the bundle of head features and the like, which may refer to the semantic

or pragmatic structure. (47) contains the same pattern of dependency as in (46), a dependency between the first argument of c and the first argument of a binding with functor category. Those pieces of dependency are hence eliminated by fusion, so that (46) is replaced with (48) and (47) gives rise to (49).

```
(48) :- c0(H,[],STR0,STR1) ···.

(49) c0(HEAD,CATLIST,X,Z) :-
          c(CAT,X,Y),
          c0(HEAD,[CAT|CATLIST],Y,Z).
```

If H were further constrained by another atomic constraint in the top clause, then we would eliminate the resulting dependency by downward panetration to obtain the following.

```
(50) :- c1([],STR0,STR1) ···.

(51) c1(CATLIST,X,Z) :-
          c(CAT,X,Y),
          c1([CAT|CATLIST],Y,Z).
```

Now H has become a transclausal variable, probably appearing in another definition clause of c1. H's absence in (51) demonstrates that penetration eliminates dependency by eliminating the occurrence of the relevant variable from a relevant clause.

As for (48), we might have eliminated the dependency with respect to the binding concerning []. The reason we eliminated the dependency concerning H instead is that it enables the above folding immediately, as should be evident from (49). This choice can be a part of general strategy independent of the problem domain. Note that this folding was made possible because we used fusion when obtaining (48) and (49); otherwise we would not have the same predicate in those two clauses.

If there is another atomic constraint on STR0 in (46), as is the case with comprehension, we could have eliminated the dependency concerning that constraint instead of that concerning the first argument of c. Namely, the following could have been worked out from (46) and (47), by letting STR0 penetrate down through.

```
(52) :- d0(category(H,[]),STR1), ···.

(53) d0(category(HEAD,CATLIST),Z) :-
          d0(CAT,Y),
          c(category(HEAD,[CAT|CATLIST]),Y,Z).
```

This would block the folding corresponding to the one we did to obtain (49). If the dependency about STR0 got eliminated in (50) and (51) instead, then we would have:

```
(54) :- e0(H,[],STR1) ···.

(55) e0(HEAD,CATLIST,Z) :-
          c(CAT,STR0,Y),
          c0(HEAD,[CAT|CATLIST],Y,Z).
```

For the sake of folding about STR0 in (55), we could substitute c(CAT, STR0, Y) successively as follows.

```
(56) e0(HEAD,CATLIST,Z) :-
          c0(HEAD0,CATLIST0,STR0,Y),
          CAT=category(HEAD0,CATLIST0),
          c0(HEAD,[CAT|CATLIST],Y,Z).

(57) e0(HEAD,CATLIST,Z) :-
          e0(HEAD0,CATLIST,Y),
          CAT=category(HEAD0,CATLIST0),
          c0(HEAD,[CAT|CATLIST],Y,Z).
```

The first substitution here is valid, provided that `c` eventually instantiates its first argument to `category(_,_)`, which could be detected when we worked out the definition of `c0`. This new operation might be called *fission*, in contrast with fusion. Fission may be executed as soon as detected possible in the corresponding fusion. Furthermore, the fusion and the fission in the present example could be finished at the stage of precompilation, previous to parsing or generation.

Next let us investigate the termination property. In the previous subsection, we have already seen that DP is not trapped in an infinite loop due to left recursion which DCG of the ordinary sort falls in. DP is shown to avoid infinite loops in other important cases of sentence processing as well. In the case of Prolog, an infinite loop may arise when a clauses like (47) is recursively applied on the second literal in its body. In DP, we are able to escape such a trap. The dependency in (49) might be eliminated and give rise to the following.

```
(58) c0(HEAD,CATLIST,X,Z) :-
          c(CAT,X,Y),
          f1(HEAD,CAT,CATLIST,Y,Z).

(59) f1(HEAD,CAT0,CATLIST,X,Z) :-
          c(CAT,X,Y),
          f1(HEAD,CAT,[CAT0|CATLIST],Y,Z).
```

The infinite loop is avoided by unfolding only predicates which are known to be solvable, essentially along the line of Tamaki and Sato's tabulation technique [16]. The dependency about the third argument of `f1` in the body of (59) is hence left untouched until `f1` turns out to be satisfiable. Since (59) does not contribute to the satisfiability of `f1`, it turns out that `f1` is satisfiable only when another definition clause is worked out whose body is satisfiable. If there comes out no such clause, `f1` is known to be unsatisfiable and thus (59) is deleted, provided that CWA is employed. Note that such a control is incorporated into DP with a very small cost for marking each predicate with its satisfiability status.

DP may hence be regarded as having a very nice termination property, though of course it does not always terminate, because the constraints can simulate arbitrary Turing machines. To ensure termination, we should relax the transformation procedure so that not every dependency be eliminated, at the price of overlooking inconsistencies. Such partiality of processing is essential in the domain of AI. The control of partial processing is thus a very important issue in every theory of computation for such domains.

So let us finally consider how to control partial processing or, almost equivalently, processing order. In this connection, Hasida [5] has proposed head-driven propagation (HDP for short), which gives priority to elimination of dependency involving the head in

grammatical sense. In (47), for instance, the dependency relating to the second literal of the body is probably eliminated prior to the other dependency, because that literal is the head in the local tree represented by this clause. This means that the information in the head is exploited more readily than that in the non-head category. HDP is a good heuristic because the reference to the head is of a great help in determining the grammatical status, syntactic or semantic, of the non-head category currently under consideration. Shieber's semantic-head-driven generation [13] is a similar strategy, except that it is biased towards semantic information and primarily restricted to generation.

Although the details are beyond the scope of the current line of discussion, HDP naturally emerges if we extend the present framework by incorporating spreading activation [7, 19, 20]. The general idea is that closely related nodes strongly activate each other. Here we look upon a constraint as a network. The nodes in this network are variables, atomic constraints, clauses and predicates, and the links are references among them. For instance, the node corresponding to a variable is linked with the node corresponding to a literal which refers to that variable as one of its arguments. Two nodes are closely related when there are many short paths between them. In the above clauses describing a local branching tree, the connection between the first literal (the head of the clause and the mother in the local tree) and the third literal (the head daughter in the local tree) is actually stronger than the one between the first two and that between the last two. In (47), that connection includes three different paths running through `HEAD`, `CATLIST` and `Z`, whereas the latter two involve one and two paths. In general, the connection between the mother category and the head daughter is thus locally the strongest, This means that the head daughter tends to get activated more than the other daughter, so that more information tends to be exploited out of the head than out of the other category, which is HDP.

4 Final Remarks

A method of transforming logic programs called dependency propagation has been described. We have emphasized that sentence processing is carried out efficiently by a domain-independent transformation strategy. A tabular method of parsing and generating sentences has shown to naturally emerge as a corollary of our general control strategy. It has been also demonstrated that folding often takes place in manipulation of compound structures. All of this exemplifies that constraint-based methods may be as efficient as procedural methods, perhaps especially in the domain of daily rather than artificial problems.

We have discussed little about heuristic control of partial transformation. What DP as described above tends to do is an exhaustive search, which is almost always stupid in the domain where partiality of information is encountered. Heuristics based on assumption cost [8] and spreading activation are discussed elsewhere [6], but further investigation is widely open about this issue.

Acknowledgements

The author received useful comments on earlier versions of the present work from many colleagues. In particular, thanks go to NAKANO Mikio, NISHIKAWA Noriko, TUDA

Hirosi, HAGIWARA Kaoru, and Prof. TANAKA Hozumi. Suggestions by the unidentified reviewers were also helpful.

References

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

[2] 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.

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

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

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

[6] Hasida, K. (1989) 'A Constraint Based View of Language,' manuscript presented at the STASS Asilomar Conference.

[7] Hasida, K., Ishizaki, S., and Isahara, H. (1987) 'A Connectionist Approach to the Generation of Abstracts,' in Kempen, Gerard (ed.), *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics*, Martinus Nijhoff Publishers (Kluwer Academic Publishers), pp. 149-156.

[8] Hobbs, J., Stickel, M., Martin, P., and Edwards, D. (1988) 'Interpretation as Abduction,' *Proceedings of the 26th Annual Meeting of ACL*, pp. 95-103.

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

[11] Sakai, K. and Sato, Y. (1988) *Boolean Gröbner Bases*, ICOT Technical Memo No. 488.

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

[13] Shieber, S.M., Noord, G., and Moore, R.C. (1989) 'A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms,' *Proceedings of the 27th Annual Meeting of ACL*, pp. 7-17.

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

[18] Waltz, D.L. (1975) 'Understanding line Drawings of Scenes with Shadows,' in Winston, P.H. (*ed.*) *The Psychology of Computer Vision*, McGraw-Hill.

[19] Waltz, D.L., and Pollack, J. (1985) 'Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation,' *Cognitive Science*, Vol. 9, pp. 51-74.

[20] Ward, N. (1988) 'Issues in Word Choice,' *Proceedings of the 12th COLING*, pp. 726-731.