Emergent Computation for Semantic Interpretation by Rule-Based Equivalent Transformation

Hiroshi Mabuchi[1], Kiyoshi Akama[2], and Hidekatsu Koike[2]

[1] Iwate Prefectural University, 152-52 Sugo Takizawa Iwate 020-0173 Japan
mabu@soft.iwate-pu.ac.jp
[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan
{akama, koke}@cims.hokudai.ac.jp

Abstract. We propose a method for emerging efficient processing for semantic interpretation. In this method, creation of many individual algorithms written to handle various situations is avoided, and the basic knowledge in an object domain and the situation in which a sentence was uttered as well as the given sentence are described as a declarative program. Semantic interpretation is then performed by equivalent transformation using correct rules. With this method, efficient processing matched to changing conditions emerges naturally in the computation process.

1 Introduction

The meaning of a sentence must be interpreted properly by using not only the given sentence, but also factors such as the situation in which the sentence was uttered and background knowledge in the domain of the topic. Attaining such a level of natural language understanding is not easy.

One of the problems when an algorithm, in which the computation order is fixed, is used is the need to assume a variety of cases and describe a number of separate algorithms that are suitable for each case. In order to separate cases properly, the situation in which a sentence was uttered and background knowledge must be considered in combination. However, since the number of possible combinations is extremely large, it is difficult to correctly describe them. Moreover, the maintenance required due to the addition of situations and the revision of background knowledge is costly.

Another problem is the efficiency of the computation. In natural language understanding, since a number of different kinds of constraints are related complexly, selection and flexible processing of an appropriate constraint are required at each stage of constraint elimination in order to handle the constraints efficiently. When attempting to execute this processing using an algorithm in which the computation order is fixed, an appropriate constraint that should be solved at each stage can not be selected from among all the possible constraints, and the volume of unnecessary computations becomes very large.

In order to solve these problems, we propose a new method for emerging efficient processing for semantic interpretation – creation of many individual algorithms written to handle various situations is avoided, and the basic knowledge in an object domain and the situation in which a sentence was uttered as well as the given sentence are described as a declarative program. Semantic interpretation is then performed by equivalent transformation using correct rules [7]. With this method, efficient processing matched to changing conditions emerges naturally in the computation process. A declarative program is a generalized logic program and can treat various data structures, including feature structures [3], strings, and multisets, as well as terms.

The framework of computation is referred to as Rule-Based Equivalent Transformation (RBET), which is based on a new model called the declarative computation model [10]. In RBET, a given problem is formalized as a problem which transforms one declarative program into another declarative program. A program is successively transformed into different programs by selecting and applying an appropriate rule from a number of equivalent transformation (ET) rules. The answer can be obtained from the final version of the program.

RBET does not fix computation order beforehand and aims to achieve flexible and efficient computation by selecting the rule to apply using controls, such as minimizing the number of clauses.

2 Method for Solving Problem

We first formalize a problem using declarative program. In order to represent an object, we consider the necessary data structure and decide a specialization system. A specialization system [10] is a structure consisting of logic objects and specializations, which are operations for logic objects. A logic object is an abstract concept of an object (an atomic formula or a term), and a specialization is an abstract concept of a substitution. Objects of a problem are represented by declarative programs on a specialization system.

Next, we construct the RBET to actually solve a formalized problem. We first decide a set of ET rules. The selection of a rule set has a great influence on the efficiency of computation. Unfolding rule [8] is an ET rule, and there exist many ET rules that are not unfolding rule. When the unfolding rule by which the number of clauses is increased are used, the efficiency of computation is often decreased. Therefore, introduction of efficient ET rules by which the number of clauses is not increased is important. Moreover, in order to transform a program at each stage, we decide the strategy to select an atom in the body and an applicable rule. Here, a logic object that appears in definite clause is often called an *atom*.

The present study efficiently solves problems as follows.

- By properly selecting a specialization system, we describe logic objects flexibly using knowledge representations such as feature structures.

- We use the ET rules, such as the rules based on unfolding, the candidate elimination rules, and the arithmetic computation rules, as well as unfolding rules. By restricting the domain of unfolding rule, many rules based on unfolding can be obtained from one unfolding rule. These rules make possible the detailed representation of processing. The possibility of efficient computation becomes higher the more ET rules there are.
- By first selecting the rule by which the number of clauses is not increased, the amount of computation in equivalent transformation is controlled and efficient computation can be performed.

3 Problem Setting of Natural Language Understanding

We define natural language understanding in the present paper as "outputting instructions that cause an appropriate performance when a sentence and situations are given as input data". Concretely, we assume a system that performs an appropriate action when an input sentence, such as "Capture a piece on file 4", is given, assuming the situation on a japanese chess (shogi) board. For example, assume that the sentence "Move Keima (knight) to (7,c)" is given in the situation of the shogi board shown in Fig.1.

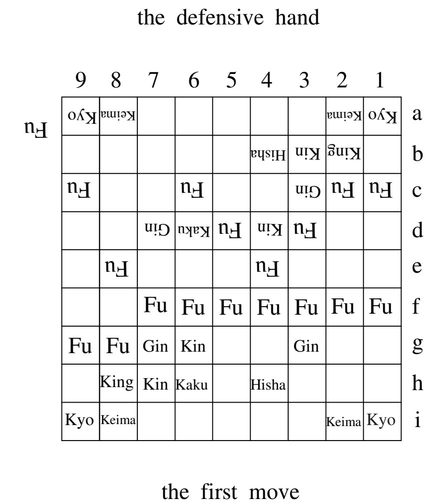

Fig. 1. Given Japanese Chess (Shogi) Board

Then, consider that the system outputs the instructions, "Move Keima on (8,a) to (7,c)", where the files of the shogi board are numbered 1, 2, 3, etc. The ranks are numbered a, b, c, etc, (in computing, a, b, and c correspond to 1, 2, and 3.). For example, (8,a) denotes the square at file 8 and rank a. In the given sentence, only the position to move to, (7,c), is given; however the information of

(8,a) must be supplied in the instructions that indicate the movement of a piece. For that purpose, the situation on a shogi board, the rules of shogi and other knowledge in the domain (e.g., common knowledge on the position relation), as well as a given sentence, must be used.

4 Formalization by Declarative Programs

4.1 Logic Objects

For example,

(capture <a:kind/Fu(Pawn), position/<p:file/*X>> <b:kind/King>)

is a logic object. Symbols starting from an asterisk are variables, and other symbols are constants. In the above logic object, the following three feature structures appear.

<a:kind/Fu, position/<p:file/*X>>, <p:file/*X>, <b:kind/King>

These have a form of <tag:attribute/value, ...>. A feature structure [3] means an object described by a set of attribute value pairs, and tag is a label adhering to the structure. Variable, constant, and feature structure are together called *logic object*. Herein, a feature structure is called a *feature structure object*, stressing the fact that the structure is a logic object.

For example, "Sente's Fu" can be represented by a feature structure object as <e1:kind/Fu, player/Sente>. Similarly, "Gote's Kin (Gold) on (3,b)" can be represented as follows.

<e2:kind/Kin, player/Gote, position/<r:file/3, rank/b>>,

where Sente means the first move, and Gote means the defensive hand. Since these are logic objects on pieces of shogi, these are called *piece objects*. <r:file/3, rank/b> is called a *position object*.

4.2 Specialization

We first define basic specialization. We adopt the operation that substitute an arbitrary object for a variable, such as {*position/*place} and {*pos/<r:file /3>}, as a basic specialization.

Logic objects are changed by this as follows.

*position $\rightarrow$ *place, *pos $\rightarrow$ <r:file/3>,

<e3:position/*pos> $\rightarrow$ <e3:position/<r:file/3>>.

As basic specializations concerning feature structure objects, we adopt two kinds of operations for the replacement of a tag and addition of an attribute/value. For example, feature structure objects are changed by these operations as follows.

<e4:kind/Fu> $\rightarrow$ <e5:kind/Fu>.

<e4:kind/Fu> $\rightarrow$ <e4:kind/Fu,player/Sente>.

<r:file/3> $\rightarrow$ <r:file/3,rank/b>.

A specialization is defined as an arbitrary finite sequence of the above-defined three kinds of basic specializations. The action of a specialization is defined as the action when basic specializations, which are elements of the specialization, are applied in order.

For example, a sequence $\theta\sigma$ obtained from a basic specialization θ that replaces e4 with e5 and a basic specialization σ that adds player/Sente is an example of specializations. For example, the following feature structure object is changed by $\theta\sigma$ as <e4:kind/Fu> → <e5:kind/Fu,player/Sente>.

4.3 Unification

A unification is decided based on the definition of object and specialization.
For example,
<x:kind/Fu, position/*p> and <y:player/Sente, position/<r:file/7>> are changed by the unification as follows.

```
<z:kind/Fu, player/Sente, position/<r:file/7>>
```

When both objects have the same attribute, if their attribute values can not be unified, the unification fails. For example, the unification of <x:kind/Fu> and <y:kind/King> fails.

4.4 Definite Clause

For example,
(capture <a:kind/Fu> *piece) and (foo <a:kind/Fu> *pieceList) are logic objects; therefore

```
(capture <a:kind/Fu> *piece) ← (foo <a:kind/Fu> *pieceList).
```

is a definite clause on a specialization system. In this clause, <a:kind/Fu> appears twice. Thus when the same feature structure object appears two or more times in a clause, attribute value pairs and colons can be omitted as follows.

```
(capture <a:kind/Fu> *piece) ← (foo <a> *pieceList).
```

4.5 Representation of Meaning in Sentence

We herein show how to represent the meaning of sentence.

For example, the meaning of the sentence "Capture a piece on file 4 by Sente's Fu." is represented as follows.

```
H1 (capture <a:kind/Fu> *piece) ←
B1      (player <a> Sente),
B2      (file *piece 4),
B3      (can_capture <a> *piece).
```

If this clause is read as follows, then the clause can be seen to correspond to the given sentence.

The player of **a** (Fu) is Sente (B1), a piece is located on file 4 (B2), and if the piece can be captured by **a** (B3), then capture the piece by **a** (H1).

4.6 Representation of Rules in Shogi

The two predicates "player" and "file" can be given by the following clauses, respectively.

```
(player <X:player/*player> *player) ←.
```

<X:..> is the piece of *player.

```
(file <Y:position/<p:file/*file>> *file)) ←.
```

The file of the piece <Y:..> is *file.

The predicate "can_capture" can be given as follows.

```
(can_capture *A *B) ←
   (enemy_piece *A *B),
   (position *B *pos),
   (exist *A),
   (rules_for_piece_movement *A *pos),
   (no_piece_on_the_path *A *pos),
   (on_the_board *pos),
   (exist *B).
```

This clause means that when *A and *B are enemy pieces, *B is located on *pos, there exists *A on a board, *A can move to *pos under the rules of shogi, there is no piece on the path to *pos, *pos is on the board, and *B exists on the board, *B can be captured with *A.

The predicate "exist" is defined as follows.

```
(exist *piece) ←
   (situation board *board),
   (member *piece *board).
```

This clause means that when the situation on a board is represented by *board, which is the list of all the pieces on the board, and *piece is a member of *board, there exists *piece on the board.

4.7 Representation of Situation on a Board

The predicate "situation" can be given by two fact clauses of the following form.

```
(situation board [list of all pieces on a board]) ←.
(situation pieces_in_hand [list of all pieces in hand]) ←.
```

For example, since there exist 39 pieces on the board in the example shown in Fig.1, the situation on the board is represented as follows.

```
(situation  board
    (<k1: kind/Fu,player/Sente,position/<file/1,rank/f>>
     <k2: kind/Kyo,player/Sente,position/<file/1,rank/i>>
          :
     <k38: kind/Fu,player/Gote,position/<file/9,rank/c>>
     <k39: kind/Kyo,player/Gote,position/<file/9,rank/a>>)

) ←.
```

5 Equivalent Transformation Rules

We show several ET rules that are not unfolding rules.

5.1 Rule Based on Unfolding for "exist" Atom

In RBET, the rules based on unfolding are used before other rules. As a result, detailed processing becomes possible and efficient computation can be performed.

For example, from the definition of the predicate "exist" shown in section 4.6, the rule "exi", which restricts the object of the transformation to only the "exist" atom, can be obtained. The rule "exi" is applied only to the "exist" atom, whose argument is one, and causes the same transformation result as the unfolding rule. When this rule is applied, the number of clauses in program does not change, therefore, the cost of this rule is 0. In contrast, the cost of the unfolding rule which is the total mapping is ∞.

5.2 Rule Based on Unfolding for Member Atom

A predicate member is defined as

```
(member *a (*a . *rest)) ←.
(member *a (*b . *rest)) ← (member *a *rest).
```

From this definition, the rule "mem", which unfolds the member atom, can be easily obtained. This rule replaces the atom having the form of (member A (F . R)) with the atom in the body. The operation is performed for two clauses of member, and a maximum of 2 new clauses are generated from a selected clause.

That is, when this rule is applied, the number of clauses in program increases by a maximum of 1. Therefore, the cost of this rule is 1.

5.3 Single Candidate Rule

When the conditions are imposed on the argument as well as on the predicate's name, in many cases the processing that the rule performs becomes easier. For example, when the second argument of a selected member atom is a list consisting of one element, the first argument must be equal to the element. Therefore, when a member atom of a selected clause has the form of (member X (E)), a new clause can be obtained by eliminating the member atom from the selected clause and by specializing the selected clause by the unification of X and E. Even if this rule is applied, the number of clauses in program does not change. Therefore, the cost of this rule is 0.

5.4 Candidate Elimination Rule

The candidate elimination rule limits the candidates in the candidate list of a selected member atom, `(member <object> [candidate list])`, to only the candidates that can unify with `<object>`. Therefore, `(member <object> [candidate` list]) is changed to (member <object> [new candidate list]), where new candidate list consists of the candidates that can unify with <object>. When new candidate list is the same as the original candidate list, the candidate elimination rule is inapplicable. This is the step for avoiding risk in the infinite applications of the transformation rule. The cost of this rule is clearly 0.

5.5 Arithmetic Computation Rule

Predicates for arithmetic computation are necessary in order to compute the movement of pieces. For example, (minus X Y Z) means that $X - Y = Z$.

An arithmetic computation rule is used to transform an atom of arithmetic computation. For example, when the atom of (minus *x 2 5) exists in the body of the selected clause, the computation is performed by the arithmetic computation rule. As a result, the minus atom is removed from the selected clause and *x is unified with 7. The cost of this rule is 0.

The following is another example of arithmetic computation rules. If the selected atom has the form of (minus X Y 0), X must be equal to Y. Therefore, this minus atom is removed from the selected clause, and X is unified with Y. Ths cost of this rule is also 0.

6 Semantic Interpretation by Equivalent Transformation

As an example, we use the sentence "Capture a piece on file 4 by Sente's Fu". The situation on the shogi board is as shown in Fig.1.

6.1 Outline of Transformation Process

In equivalent transformations of the present paper, an atom in a body is selected from only clauses (let P_{rule} be a set of these all clauses) in which the head is an atom using a predicate – a predicate (e.g., "capture") in the head of clauses used to represent the meaning of the sentence.

A set of all clauses that represent the rules in shogi, situations on a board, etc. is unchanged throughout the process of equivalent transformations. When all of the clauses in P_{rule} become fact clauses by equivalent transformation, a problem can be solved. Transformations are automatically performed under the control that minimizes the number of clauses.

The clause M1, which represents the meaning of a given sentence, is equivalently transformed into clause M10 as follows: Here, "Unfold X" means to select an atom of X in a body and transform the atom by a rule based on unfolding.

```
M1 (capture <a:kind/Fu> *piece) ←
        (player <a> Sente),
        (file *piece 4),
        (can_capture <a> *piece).
```

```
     ↓[Unfold 'player' and 'file']

M2 (capture <a:kind/Fu,player/Sente> <x:position/<p:file/4>>) ←
        (can_capture <a> <x> ).

     ↓[Unfold 'can_capture']

M3 (capture <a:kind/Fu,player/Sente> <x:position/<p:file/4>>) ←
        (enemy_piece <a> <x>),
        (position <x> *pos),
        (exist <a>),
        (rules_for_piece_movement <a> *pos),
        (no_piece_on_the_path <a> *pos),
        (on_the_board *pos),
        (exist <x>).

     ↓[Unfold 'enemy_piece' and 'position']

M4 (capture <a:kind/Fu,player/Sente>
           <x:position/<p:file/4>,player/Gote>) ←
        (exist <a>),
        (rules_for_piece_movement <a> <p>),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).

     ↓[Unfold '(exist <a>)' and 'situation']

M5 (capture <a:kind/Fu,player/Sente>
           <x:position/<p:file/4>,player/Gote>) ←
        (member <a> [list of all the pieces on the board]),
        (rules_for_piece_movement <a> <p>),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).

     ↓[candidate elimination rule]

M6 (capture <a:kind/Fu,player/Sente>
               <x:position/<p:file/4>,player/Gote>) ←
        (member <a> [list of nine Sente's Fu]),
        (rules_for_piece_movement <a> <p>),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).
```

```
     ↓[Unfold 'rules_for_piece_movement' etc.]
     ↓[arithmetic computation rule]

M7 (capture <a:kind/Fu,player/Sente,position/<q:file/4,rank/*y>>
              <x:position/<p:file/4,rank/*z>,player/Gote>)←
        (member <a> [list of nine Sente's Fu]),
        (minus *y *z 1),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).

     ↓[candidate elimination rule]

M8 (capture <a:kind/Fu,player/Sente,position/<q:file/4,rank/*y>>
              <x:position/<p:file/4,rank/*z>,player/Gote>)←
        (member <a>
        (<k7:kind/Fu,player/Sente,position/<r:file/4,rank/f>>)),
        (minus *y *z 1),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).

     ↓[single candidate rule]

M9 (capture <a:kind/Fu,player/Sente,position/<q:file/4,rank/f>>
          <x:position/<p:file/4,rank/*z>,player/Gote>)←
        (minus 6 *z 1),
        (no_piece_on_the_path <a> <p>),
        (on_the_board <p>),
        (exist <x>).

     ↓[arithmetic computation rule]
     ↓[Unfold all the atoms in the body]

M10 (capture <a:kind/Fu,player/Sente,position/<q:file/4,rank/f>>
           <x:kind/Fu,position/<p:file/4,rank/e>,player/Gote>) ←.
```

6.2 Candidate Elimination

In `M5`, since the member atom first appears, the candidate elimination rule can be applied. Among the elements of the candidate list, in the member atom of `M5`, for example, `<k1:kind/Fu,player/Sente,position/<file/1,rank/f>>` is unifiable with `<a:kind/Fu,player/Sente>`, and therefore is left as a candidate, whereas

```
<k2:kind/Kyo,player/Sente,position/<file/1,rank/i>>
```

is not unifiable with <a:kind/Fu,player/Sente>, and therefore is removed from the candidates by the candidate elimination rule. As a result, the member atom is changed as follows.

```
(member <a:kind/Fu,player/Sente>
        [list of all pieces on the board])
                 ↓
(member <a:kind/Fu,player/Sente> [list of nine Sente's Fu])
```

Therefore, M5 is transformed into M6.

6.3 Preservation of Candidates

In M6, although the number of candidates in Sente's Fu is reduced, it has not yet been specified which piece on the board "Sente's Fu" is. However, at this step, since further reduction of the number of candidates is impossible, application of the candidate elimination rule is impossible.

One of the applicable rules at this point is the rule based on unfolding for the predicate "rules_for_piece_movement". After the application of this rule, other rules based on unfolding and the arithmetic computation rule can be further applied.

As a result, the object is changed as follows.

```
<a:kind/Fu,player/Sente> ⟹
  <a:kind/Fu,player/Sente,position/<q:file/4,rank/*y>>
<x:position/<p:file/4>,player/Gote> ⟹
  <x:position/<p:file/4,rank/*z>,player/Gote>
```

This change causes a change in the first argument of the member atom. Thus, while other atoms are being selected and transformed, the member atom holds the candidate of Sente's Fu and fills the role of receiving information on the change of the object.

6.4 Change of Candidate

In M7, as the object of the first argument in the member atom was changed, the application of the candidate elimination rule becomes possible again. As a result, the number of candidates is reduced as follows.

```
(member <a:kind/Fu,player/Sente,position/<q:file/4,rank/*y>>
        [list of nine Sente's Fu])
                 ↓
(member <a:kind/Fu,player/Sente,position/<q:file/4,rank/*y>>
        (<k7:kind/Fu,player/Sente,position/<r:file/4,rank/f>>))
```

That is, "Sente's Fu on file 4" is seen to be "Sente's Fu on (4,f)" among nine Sente's Fu.

6.5 Decision of Object

In `M8`, only one candidate exists in the member atom. In this case, the single candidate rule can be applied. The member atom in `M8` is eliminated by executing

```
(= <a:kind/Fu,player/Sente,position/<p:file/4,rank/*y>>
   <k7:kind/Fu,player/Sente,position/<r:file/4,rank/f>>).
```

This causes the effect of substituting f for `*y`. Thus, `M8` is transformed into `M9`. Moreover, all the atoms of the body in `M9` are eliminated by the rules based on unfolding and the arithmetic computation rule. As a result, `M10` can finally be obtained. From `M10`, the following instructions can be obtained:

"Capture Gote's Fu on (4,e) by Sente's Fu on (4,f)".

6.6 Emerged Information Processing

When we observe the process obtained by RBET, the following skillful inference appears to be performed precisely according to each situation.

1. piece $\to_{capture}$ piece
2. Sente's Fu $\to_{capture}$ piece on file 4
3. Sente's Fu $\to_{capture}$ Gote's piece on file 4
4. Sente's Fu on file 4 $\to_{capture}$ Gote's piece on file 4
5. Sente's Fu on (4,f) $\to_{capture}$ Gote's piece on file 4
6. Sente's Fu on (4,f) $\to_{capture}$ Gote's piece on (4,e)
7. Sente's Fu on (4,f) $\to_{capture}$ Gote's Fu on (4,e)

1. Based on the meaning of "capture," one piece captures another piece.
2. By referring to the meaning of the given sentence, Sente's Fu captures a piece on file 4.
3. Based on a rule in shogi that a player can only capture an enemy piece, a piece on file 4 must be Gote's piece.
4. Based on a rule in shogi, Fu is not allowed to move to another file. Therefore, Sente's Fu must be on file 4 in order to capture an enemy piece on file 4.
5. Based on the situation on the board, Sente's Fu on file 4 is only one and is on (4,f).
6. Based on a rule in shogi, Fu can move only one square straight upward at a time. Therefore, the enemy piece captured by Fu is on (4,e).
7. Based on the situation on the board, the enemy piece on (4,e) is Fu.

That the possible movements are gradually reduced to one during this process is indicated by the change in the number of possible movements.

a. At stage 1, there are 39 pieces on a given board. Therefore, the number of possible movements is $39 \times 39 = 1521$.
b. At stage 2, Sente has nine Fu pieces and the number of pieces on file 4 is five; therefore, the number of possible movements is reduced to $9 \times 5 = 45$.

c. At stage 3, the number of Gote pieces on file 4 is three; therefore, the number of possible movements is 9×3=27.
d. At stages 4 and 5, Sente has only one Fu on file 4; therefore, the number of possible movements is 1×3=3.
e. At stages 6 and 7, the number of pieces on (4,e) is only one; therefore, the number of possible movements is 1×1=1 and the movement is decided.

In this example, the movement must be selected from among several possible movements. In such a case, the number of clauses in the program is increased only by the number of possible movements, but in the system of the present paper, the number of clauses in the program is not increased from start to finish. This control of the increase in the number of clauses enhances the efficiency of computation.

One of reasons that such efficiency is obtained is that RBET allows the use of a variety of rules. If the number of rules used in RBET is limited, the number of clauses increases.

7 Related Works

Various systems that identify various meanings have been described, including the system proposed by Winograd [11] and that of Mellish [6]. The latter is an improvement of the former to provide greater flexibility. As described in section 6.6, a processing similar to that in the latter system is realized using member atoms. However, the present method avoids creation of many individual algorithms written to handle various situations and realizes the same processing as that of Mellish using the following system constuction method: (1) represent the objects by declarative programs, (2) introduce basic and general-purpose ET rules, and (3) transform the declarative program by the control that minimizes the number of clauses.

Studies dealing with natural language processing by program transformation include those by Hashida [4] and by Den et al. [1]. The main differences between the present study and these studies are as follows:

- The previous studies deal primarily with syntactic analysis and syntactic generation, whereas the focus of our study is semantic interpretation.
- In the previous studies, a logic program is used as an object of transformation, whereas in our study a declarative program is used. Therefore, in one example of the present paper, feature structure objects can be treated.
- These previous studies use a small number of transformation schemata, which are independent of the selected atom, such as unfold/fold transformation [8]. Whereas the present study uses a number of specific ET rules for each atom. Therefore, efficient ET rules that use the characteristic of atoms can be utilized, and the control that minimizes the number of clauses becomes possible.

8 Conclusions

We proposes a method for emerging efficient processing for semantic interpretation. The present method avoids creation of many individual algorithms written to handle various situations and does not specify closely the order of computation. In spite of this, the result is that efficient processing matched to changing conditions emerged naturally in the computation process.

The application of the present method is not confined to semantic interpretation of natural language. In natural language processing, for example, unified processing of various constraints, such as syntactic constraints, semantic constraints, and extra-linguistic constraints, is also possible.

References

1. Den, Y., Matsumoto, Y., and Nagao, M.: Cognitive Constraint Programming and Integrated Natural Language Processing – on the Integration of Syntax Analysis and syntactical Generation. Computer Software, Vol.8. No.6 (1991) 38–50
2. Jaffar, J. and Lassez, J.L.: Constraint Logic Programming. Technical Report, Department of Computer Science, Monash University (1986)
3. Carpenter, B.: The Logic of Typed Feature Structures. Cambridge Tracts in Theoretical Computer Science 32, Cambridge University Press (1992)
4. Hashida, K.: Constraint and Language. Computer Software, Vol.6. No.4 (1989) 16–29
5. Hashida, K.: Integration of Parsing and Generation of Natural language Snetences. Computer Software, Vol.33. No.7 (1992) 790–800
6. Mellish, C.: Computer Interpretation of Natural Language Descriptions. Ellis Horwood (1985)
7. Tamaki, H.: Program Transformation of Logic Programs. in 'Program Transformation', Kyouritsu Shuppan (1987)
8. Tamaki, H. and Sato, T.: Unfold/fold Transformation of Logic Programs. Proc. of 2nd ILPC (1984) 127–138
9. Akama, K.: Set Bound Variables and its Application to Natural Language Processing. Artificial Intelligence, Vol.4. No.2 (1989) 177–184
10. Akama, K.: Declarative Computation Model. Information Processing, SIG-programming, 12-10-PRG (1993) 87–94
11. Winograd, T.: Understanding Natural Language. Academic Press (1972)