Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp141-146)

Componentwise Program Construction: The Equivalent Transformation Computation Model versus the Logic Programming Model

KIYOSHI AKAMA
Information Initiative Center
Hokkaido University
Sapporo, Hokkaido, 060-0811
JAPAN

EKAWIT NANTAJEEWARAWAT
Sirindhorn Intl. Inst. of Tech.
Thammasat University
Rangsit, Pathumthani, 12121
THAILAND

HIDEKATSU KOIKE
Faculty of Social Information
Sapporo Gakuin University
Ebetsu, Hokkaido, 069-8555
JAPAN

Abstract: In the equivalent transformation (ET) computation model, a program is a set of procedural rewriting rules for answer-preserving transformation of problems with respect to given background knowledge. In this paper, we discuss an approach to program construction by creating and accumulating individually correct and efficient program components one by one, referred to as componentwise program construction. Basic requirements for componentwise program construction are identified, based on which we show that the ET model possesses several desirable properties for this program construction approach, in particular, compared with the logic programming model. In addition, we compare the expressive power of program components in the ET model and that of components in the logic programming model by viewing computation in the latter model as problem transformation using only one specific class of rewriting rules, i.e., single-head general unfolding-based rules, and then demonstrating that a larger class of rules is needed for effective computation.

Key–Words: Componentwise program construction, Equivalent transformation, Rule-based computation, Program synthesis, Rewriting rules, Computation paradigm

1 Introduction

Equivalent transformation (ET) is one of the most fundamental principles of computation, and it provides a simple and general basis for verification of computation correctness. Computation by ET was initially implemented in experimental natural language understanding systems at Hokkaido University during 1990–1992 [1], and the idea was further developed into a new computation model, called the *ET model* [2, 3]. A program in this model is a set of prioritized rewriting rules for answer-preserving transformation of problems, and a problem solving process consists in successive rule application. Besides extensive use in the domain of first-order terms, the model has been applied in several data domains, including RDF and XML (e.g. in [5]).

As opposed to declarative computation paradigms such as logic programming (LP) and functional programming (FP), programs are clearly separated from specifications in the ET model. A specification defines a set of problems of interest and provides background knowledge for declaratively determining the answer sets of the problems. From a specification a program consisting of rewriting rules is constructed. When applied in the domain of first-order terms and first-order atoms, questions often arise as to whether the ET model offers advantages over the LP model.

Advantages of the ET model are best seen from the viewpoint of program synthesis. In this paper, comparisons between the ET model and the LP model as regards componentwise program construction is discussed. Such a program construction approach demands a high level of program component independency—it requires that the correctness and the efficiency of a program component can be verified and evaluated, respectively, independently of other components—and, moreover, capabilities of a partial program to provide a meaningful clue to creation of new components towards completing the program. Limitations of the LP model in regard to these requirements are identified, and how the ET model complies with the requirements is described. Expressive power of program components in the two models are compared by viewing computation in the LP model as problem transformation using only one specific class of rewriting rules in the ET model, and then showing that a larger class of rules is necessary for enhancement of problem solving ability.

To start with, basic requirements for componentwise program construction are described in Section 2. Sections 3 and 4 analyze the two computation models with respect to these requirements. Section 5 illustrates the need for a wide variety of program components and reveals limitations of program components in the LP model as to the expressiveness aspect.

2 Basic Requirements

A program prg is *partially correct* with respect to a specification S iff for each problem prb defined by S, prg yields the correct answer set of prb provided that it terminates when executing prb. It is *correct* with respect to S iff it is partially correct with respect to S and it terminates when executing each problem defined by S. Componentwise program construction is a process of generating a correct and sufficiently efficient program by creating and accumulating individually correct and efficient program components one by one on demand. This program construction approach imposes the following requirements:

(Rq1) The correctness (respectively, the efficiency) of a program component can be defined (respectively, measured).

(Rq2) The correctness (respectively, the efficiency) of a component can be verified (respectively, evaluated) independently of other components.

(Rq3) Accumulation of individually correct (respectively, individually efficient) components yields a partially correct (respectively, an efficient) program.

(Rq4) There is a wide choice of components that may be added towards completing a program.

(Rq5) Appropriate components to be added towards completing a program can be selected and created at low cost.

The first three requirements enable decomposition of program partial correctness and program efficiency into component correctness and component efficiency, respectively. The fourth requirement is concerned with the possibility of obtaining high-quality programs, especially in regard to program efficiency and program termination. It is complemented by the fifth requirement, which is concerned with the cost of a component accumulation process. These requirements altogether form a basis for analysis of the componentwise program construction in the LP model and in the ET model in the following two sections.

let $D = \emptyset$ and $G' = G$
let $Pred$ be the set of all predicates occurring in G
repeat
(a) select and remove a predicate p from $Pred$
(b) $G'' := G'|Pred$
(c) find a predicate definition D_p of p such that
(i) $G' = \mathcal{M}(D_p \cup unit(G''))$
(ii) D_p is efficient
(d) $D := D \cup D_p$
(e) $G' := G''$
until $Pred = \emptyset$

Figure 1: Componentwise program construction in the LP model

Set	Definition
G	$G_{pal} \cup G_{rv} \cup G_{ap} \cup G_{palpal}$
G_{pal}	$\{pal(t) \mid t$ is a ground palindrome list$\}$
G_{rv}	$\{rv(t_1, t_2) \mid t_1, t_2$ are ground lists, and t_1 is the reverse of $t_2\}$
G_{ap}	$\{ap(t_1, t_2, t_3) \mid t_1, t_2, t_3$ are ground lists, and appending t_2 to t_1 yields $t_3\}$
G_{palpal}	$\{palpal(t) \mid t$ is a ground list, and $pal([1\|t])$ and $pal([2\|t])$ are palindromes$\}$

Table 1: Background knowledge

3 A Solution in the LP Model

3.1 Componentwise Program Construction

In the LP paradigm, a set of definite clauses is regarded as a program, and a specification is a pair $\langle G, Q\rangle$, where G is a set of all true ground atoms in the problem domain, which is assumed in this paper to be known beforehand and is regarded as background knowledge, and Q is a set of atoms, each of which represents a query of interest. A program is executed using some proof engine, e.g. a Prolog-like SLD-resolution-based inference engine.

For any atom a, let $rep(a)$ denote the set of all ground instances of a. Given a set Q of atoms, let $rep(Q) = \bigcup\{rep(q) \mid q \in Q\}$. Given a set D of definite clauses, let $\mathcal{M}(D)$ denote the minimal model of D. A program construction problem in the LP model can be formalized by:

> Given a specification $\langle G, Q\rangle$, find a set D of definite clauses such that $\mathcal{M}(D) \subseteq G$, $rep(Q) \cap G \subseteq \mathcal{M}(D)$, and D is efficient with respect to Q.

Predicate definitions, each of which is a set of definite clauses with the same head predicate, are often regarded as program components. Following the componentwise program construction approach, a logic program is constructed from a given specification $\langle G, Q\rangle$ by using the algorithm in Fig. 1. To illus-

C_1: $palpal(X) \leftarrow pal([1|X]), pal([2|X])$
C_2: $pal(X) \leftarrow rv(X, X)$
C_3: $rv([\,], [\,]) \leftarrow$
C_4: $rv([A|X], Y) \leftarrow rv(X, R), ap(R, [A], Y)$
C_5: $ap([\,], X, X) \leftarrow$
C_6: $ap([A|X], Y, [A|Z]) \leftarrow ap(X, Y, Z)$

Figure 2: A set of definite clauses constructed from the background knowledge in Table 1

trate, suppose that the background knowledge represented as the set G of true ground atoms defined in Table 1 is given, where pal, rv and ap stand for "palindrome," "reverse" and "append," respectively. The predicate $palpal$ in the table has a special intended meaning, i.e., for any term t, $palpal(t)$ is true iff both $[1|t]$ and $[2|t]$ are ground palindrome lists. The definite clauses in Fig. 2 are constructed from this background knowledge using the algorithm in Fig. 1.

3.2 Analysis

Taking predicate definitions as program components, we now analyze componentwise program construction in the LP model. Referring to Fig. 1, Condition (i) at Step (c) provides the notion of correctness of a predicate definition D_p with respect to G' and G''. This notion is independent of any other predicate definition existing in D. Moreover, a set of individually correct predicate definitions is always partially correct when it is regarded as a logic program. Requirements (Rq1), (Rq2), and (Rq3) are fully fulfilled in regard to the correctness aspect.

As to the efficiency aspect, when a node in an SLD-tree is expanded using one predicate definition, each clause in the definition possibly yields one child node, and, therefore, the number of resulting child nodes is bounded above by the number of clauses in the definition. As such, the number of clauses in the definition serves as a measure of its efficiency—the smaller the number is, the more efficient the definition tends to be (when it is regarded as a program component). The number of clauses, however, provides only a rough measure—the exact number of resulting child nodes may be far less the the number of clauses in the definition since the unification of a goal atom and the heads of some clauses in the definition may fail.

In general, as a predicate definition contains fewer definite clauses, the number of clauses in the definition provides a more accurate efficiency measure (i.e., a lower upper bound for the exact number of child nodes resulting from node expansion). Since a logic program typically involves several predicate definitions with multiple definite clauses, approximation of program efficiency based on the number of clauses in

Model	Correctness			Efficiency			Rq4	Rq5
	Rq1	Rq2	Rq3	Rq1	Rq2	Rq3		
LP	◎	◎	◎	△	△	△	×	△
ET	◎	◎	◎	○	○	○	◎	○

◎–"very good"; ○–"good"; △–"poor"; ×–"very poor"

Table 2: Comparisons between the LP model and the ET model

predicate definitions tends to be inaccurate. Requirements (Rq1), (Rq2), and (Rq3) are thus only poorly satisfied in regard to the efficiency aspect.

As will be discussed in Section 5, from the viewpoint of computation, predicate definitions are regarded as components of only one specific kind in the ET model, i.e., single-head general unfolding-based rules, and Requirement (Rq4) is not satisfied accordingly. (It will also be shown in Section 5 that employment of such a restricted class of components alone makes it impossible to construct correct logic programs for dealing with some class of queries.) As to Requirement (Rq5), although any arbitrary predicate occurring in a given background knowledge can be selected at Step (a) of Fig. 1, construction of a predicate definition often involves creation of a number of definite clauses and the overall construction cost can be considerable. The first row of Table 2 concludes this analysis.

4 A Solution in the ET Model

4.1 Computation by ET: An Introductory Example

Assume again as background knowledge the set of true ground atoms defined by Table 1. Consider the problem "find all ground terms t such that $palpal(t)$ is true." This problem is represented in the ET model as a set consisting of a single definite clause

$$ans(X) \leftarrow palpal(X),$$

where ans stands for "answer," and this definite clause is intended to mean "X is an answer if $palpal(X)$ is true." The rewriting rules in Fig. 3 are devised for solving this problem. Table 3 illustrates a sequence of problem transformation steps by successive application of these rules, where atoms to which the rules are applied are underlined and the rule applied in each step is given in the last column. The transformation sequence changes the initial problem into the singleton set $\{ans([\,]) \leftarrow\}$, which means "the empty list is an answer (unconditionally) to the problem and there exists no other answer." The correctness of this computation can be verified by proving

r_{palpal}: $palpal(*x) \Rightarrow pal([1|*x]), pal([2|*x])$.

r_{pal}: $pal(*x) \Rightarrow rv(*x, *x)$.

r_{rv_1}: $rv([*a|*x], *y) \Rightarrow rv(*x, *v), ap(*v, [*a], *y)$.

r_{rv_2}: $rv(*x, *y), rv(*x, *z) \Rightarrow \{=(*y, *z)\}, rv(*x, *y)$.

r_{ap_1}: $ap(*x, *y, [*a|*z])$
$\Rightarrow \{=(*x, [\,]), =(*y, [*a|*z])\}$;
$\Rightarrow \{=(*x, [*a|*v])\}, ap(*v, *y, *z)$.

r_{rv_3}: $rv(*x, [*a|*y])$
$\Rightarrow \{=(*x, [*u|*v])\}$,
$rv(*v, *w), ap(*w, [*u], [*a|*y])$.

r_{ap_2}: $ap(*x, [*a], [*b, *c|*y])$
$\Rightarrow \{=(*x, [*b|*v])\}, ap(*v, [*a], [*c|*y])$.

r_{ap_3}: $ap(*x, [*a], [*b]) \Rightarrow \{=(*x, [\,]), =(*a, *b)\}$.

r_{rv_4}: $rv([\,], *x) \Rightarrow \{=(*x, [\,])\}$.

Figure 3: Examples of rewriting rules

that each rule in Fig. 3 is an answer-preserving rule with respect to the background knowledge in Table 1.

4.2 Componentwise Program Construction

In the ET model, a problem is represented as a set of definite clauses. To find its answer set, a given problem is successively transformed by using rewriting rules into a simpler but equivalent problem from which the answer set can be readily obtained. A program in this model is a set of prioritized rewriting rules, and a specification is a pair $\langle G, Q\rangle$, where G is a set of all true ground atoms in the problem domain, i.e., background knowledge, and Q is a set of problems of interest. It is required that the head predicate of each clause in a problem in Q does not occur in G. The answer set of a problem prb with respect to a specification $\langle G, Q\rangle$ is defined as the set

$\{a \mid (a \leftarrow b_1, \ldots, b_n)$ is a ground instance of some definite clause in prb and $\{b_1, \ldots, b_n\} \subseteq G\}$,

which is referred to as $T_{prb}(G)$.[1]

In order to assure correct computation results, only answer-preserving rewriting rules—correct rewriting rules—are used. A rewriting rule is said to be *correct* with respect to a set G of ground atoms iff for any problems prb and prb', if the rule transforms prb into prb', then $T_{prb}(G) = T_{prb'}(G)$. Obviously, the correctness of one rewriting rule does not depend on any other rewriting rule, and can be checked individually. Requirements (Rq1) and (Rq2) are thus fulfilled as to the correctness aspect. A correct rewriting rule with respect to G is called an *ET rule* with respect to G.

[1]Traditionally, T_{prb} is referred to as the one-step consequence operator determined by prb (see, e.g., [4]).

#	Problem	Rule
1	$\{ans(X) \leftarrow \underline{palpal(X)}\}$	r_{palpal}
2	$\{ans(X) \leftarrow \underline{pal([1\|X])}, pal([2\|X])\}$	r_{pal}
3	$\{ans(X) \leftarrow \overline{rv([1\|X], [1\|X])}, \underline{pal([2\|X])}\}$	r_{pal}
4	$\{ans(X) \leftarrow \underline{rv([1\|X], [1\|X])}, \overline{rv([2\|X], [2\|X])}\}$	r_{rv_1}
5	$\{ans(X) \leftarrow rv(X, A1), ap(A1, [1], [1\|X]), \underline{rv([2\|X], [2\|X])}\}$	r_{rv_1}
6	$\{ans(X) \leftarrow \underline{rv(X, A1)}, ap(A1, [1], [1\|X]), \overline{rv(X, A2)}, ap(A2, [2], [2\|X])\}$	r_{rv_2}
7	$\{ans(X) \leftarrow \overline{rv(X, A1)}, \underline{ap(A1, [1], [1\|X])}, ap(A1, [2], \overline{[2\|X]})\}$	r_{ap_1}
8	$\{ans([\,]) \leftarrow rv([\,], [\,]), ap([\,], [2], [2])$, $ans(X) \leftarrow \underline{rv(X, [1\|A3])}, ap(A3, [1], X), \overline{ap([1\|A3], [2], [2\|X])}\}$	r_{rv_3}
9	$\{ans([\,]) \leftarrow rv([\,], [\,]), ap([\,], [2], [2])$, $ans([A4\|A5]) \leftarrow rv(A5, A6), ap(A6, [A4], [1\|A3]), ap(A3, [1], [A4\|A5]), \underline{ap([1\|A3], [2], [2, A4\|A5])}\}$	r_{ap_2}
10	$\{ans([\,]) \leftarrow rv([\,], [\,]), \underline{ap([\,], [2], [2])}\}$	r_{ap_3}
11	$\{ans([\,]) \leftarrow \underline{rv([\,], [\,])}\}$	r_{rv_4}
12	$\{ans([\,]) \leftarrow\}$	–

Table 3: Transformation of problems

A program construction problem in this model is formalized as follows:

> Given a specification $\langle G, Q\rangle$, find a set P of prioritized ET rules such that P can compute the answer set of each problem in Q successfully and P is efficient with respect to Q.

An algorithm for componentwise program construction in the ET model is shown in Fig. 4.

4.3 Analysis

ET rules are program components in the ET model. How ET rules fulfill Requirements (Rq1) and (Rq2) in regard to the correctness aspect has been discussed in Subsection 4.2. Since a set of unit clauses obtained from a sequence of problem transformation steps using ET rules always yields a correct answer set, accumulation of ET rules always results in a partially correct program. Requirement (Rq3) as to the correctness side is also fully satisfied.

As to the efficiency side, a rule with fewer bodies is considered as a more efficient program component inasmuch as it narrows down a search space. Obviously, the number of rule bodies is an individual property of a rule. In the LP model, a program often

let $P = \emptyset$
repeat
1. run the program P under certain control of execution
2. **if** some obtained final clause is not a unit clause
   begin
   (a) select one or more atoms in the body of a non-unit final clause
   (b) determine a general pattern of the selected atoms
   (c) generate an ET rule for transforming atoms that conform to the obtained pattern
   (d) assign a priority level to the obtained rule
   (e) add the obtained rule to P
   end

until all obtained final clauses are unit clauses

Figure 4: Componentwise program construction in the ET model

contains many definite clauses and thus yields many branches of computation, some of which may be cut off by further computation; accordingly, an efficient measure based on the number of clauses is rather inaccurate. In the ET model, by contrast, when specialized ET rules are created, elimination of failure branches can be taken into account based on the expressive power of the rules. Resulting specialized rules therefore have fewer bodies compared with the general unfolding-based rules for their respective predicates. Since many failure branches can be determined and eliminated beforehand during a rule construction process, the efficiency measure based on the number of rule bodies is more accurate.

Based on the fundamental structure of the ET model, a very large class of rules can be employed—any rule whose application always results in answer-preserving transformation with respect to given background knowledge can serve as an ET rule. An operation of an ET rule can also be decomposed into a number of simpler operations that are realized by finer-grained ET rules. A very large choice of ET rules is available, satisfying Requirement (Rq4).

As to Requirement (Rq5), an incomplete program in this model can always be executed, and a set of definite clauses that preserves the answer set of an initially given problem is always obtained. The body atoms of non-unit clauses occurring in such an obtained set always provide a clue to creation of new ET rules—there are finitely many such body atoms and heuristics can be used for selection of appropriate atoms. In regard to the construction cost aspect, the possibility of decomposing a component into finer-grained components enables construction of each individual component at low cost. The second row of Table 2 sums up this analysis.

5 Necessity for a Large Variety of Rewriting Rules

Referring to Fig. 3, a classification of rules will be introduced. The need for a large variety of rule classes will be explained by pointing out that severe restriction on the choice of program components in the LP model is the root cause of the failure of logic programs in finding the answer set of the query illustrated in Subsection 4.1.

The rule r_{rv_2} in Fig. 3 makes replacement of two atoms simultaneously (see, e.g., its application to the sixth problem in Table 3), and is called a *multi-head rule*. Every other rule in the figure replaces a single atom at a time, and is called a *single-head rule*. Each single-head rule in the figure operates as an unfolding rule using some definite clauses, and is called an *unfolding-based rule*. Note that the multi-head rule r_{rv_2} is devised based on the functionality of the "reverse" relation, and its operation is completely different from unfolding. The rules r_{palpal} and r_{pal} are applicable to any $palpal$-atom and pal-atom, respectively, containing any arbitrary term, and are called *general rules*. All other rules in the figure are applicable to atoms having certain specific patterns, and are called *specialized rules*. Employment of specialized rules allows *content-based control* of computation [3]—an appropriate transformation step can be decided based on the run-time contents of clauses occurring in a computation state.

When computation by SLD resolution is viewed in the ET framework, expansion of a node (generation of its children) in a search tree for finding SLD-refutations corresponds to an unfolding transformation step. Accordingly, computation in logic programming can be seen as computation using only one specific class of rewriting rules, i.e., single-head general unfolding-based rules. By employment of such a restricted class of rules alone, it is often difficult to achieve effective computation control, in particular, for preventing infinite computation or for improving computation efficiency.

As an example, consider a logic program consisting of the definite clauses in Fig. 2, which is constructed from the set G of true ground atoms in Table 1, and the query "find all ground terms t such that $palpal(t)$ is true" illustrated in Subsection 4.1, which is represented in LP as the single goal atom $palpal(X)$. When executing this query, this logic program enters infinite computation after giving $X = [\,]$, i.e., it fails to infer that the empty list is the "only" possible ground instance of X that satisfies the query, and thus, does not yield the correct answer set. Indeed, as will shown in this section, any logic program for checking palindromes fails to terminate when ex-

ecuting this query.

This difficulty is overcome in the ET model by content-based control of computation and the possibility of employing several types of rewriting rules, including specialized rules and multi-head rules. To illustrate, attention will now be drawn to the role of the multi-head rule r_{rv_2} in successful termination of the transformation sequence in Table 3. First, consider the two atoms $pal([1|X])$ and $pal([2|X])$ in the second problem in the table, and the case when X is instantiated into a nonempty ground list, say l_X. By the definition of G_{pal} in Table 1, the first pal-atom restricts the last element of l_X to 1, whereas the second one restricts it to 2. This contradiction proves nonexistence of any answer other than $X = []$. In terms of computation, finding this contradiction involves exchange of information about the restrictions on l_X between descendants of the first pal-atom and those of the second pal-atom in a computation process.

Now consider the transformation step using the multi-head rule r_{rv_2} in Table 3. Prior to this transformation step, X is the only information connection between the two groups of descendants. Via this connection, however, the constraint on the last element of l_X cannot be exchanged in a finite form. For example, passing the restriction "the last element of l_X must be 1" through X to the descendants of the $pal([2|X])$ entails passing infinitely many patterns such as $X = [1]$, $X = [v_1, 1]$, $X = [v_1, v_2, 1]$, $X = [v_1, v_2, v_3, 1]$, and so forth, where the v_i are newly introduced variables. With this information connection alone, a contradiction must be found for each such pattern, one at a time, leading to infinite computation.

The multi-head rule r_{rv_2} provides a simple remedy. It creates an additional information connection, i.e., $A1$, through which the constraint on the last element of l_X can be passed to $ap(A1, [2], [2|X])$ as a single finite pattern, i.e., $A1 = [1|v]$, where v is some new variable. Consequently, the all-embracing contradiction can be found using a finite number of transformation steps. Such an additional information connection (common variable) cannot be created by single-head rules. Since computation in logic programming corresponds to computation using only single-head general unfolding-based rules, any logic program for checking palindromes has no information connection other than X when executing the query $\leftarrow pal([1|X]), pal([2|X])$, and therefore fails to terminate.

6 Concluding Remarks

Theoretically, program synthesis can be viewed as a search for a sufficiently efficient program in a certain space of correct programs with respect to a given specification. The chance that such a program can be found increases as the program space is larger. Clear-cut separation between specifications and programs in the ET model opens up the possibility of extending a correct-program space. A specification in this model provides background knowledge for declaratively defining the answers to problems, whereas a program consists of procedural rewriting rules for computing the answers by problem transformation. A very large class of rules can be employed—any rule whose application always results in answer-preserving transformation with respect to given background knowledge can serve as an ET rule. As a result, various classes of rules, with varying expressive power, can be introduced. For example, in the Equivalent Transformation Interpreter (ETI) system[2] developed at Hokkaido University, rules with guard conditions (possibly involving extra-logical predicates), rules with execution parts, and multi-head rules are provided.

The algorithm for componentwise program construction discussed in Section 4 provides a basis for rule generation in the second phase. Based on this framework, a program synthesis system has been implemented and used for constructing many nontrivial programs.

References:

[1] K. Akama, Y. Nomura, and E. Miyamoto, Semantic Interpretation of Natural Language Descriptions by Program Transformation, *Computer Software* 12, 1995, pp. 45–62.

[2] K. Akama, Y. Shigeta, and E. Miyamoto, Solving Logical Problems by Equivalent Transformation—A Theoretical Foundation, *Journal of the Japanese Society for Artificial Intelligence* 13, 1998, pp. 928–935.

[3] K. Akama and E. Nantajeewarawat, Formalization of the Equivalent Transformation Computation Models, *Journal of Advanced Computational Intelligence and Intelligent Informatics* 10, 2006. (In press; to appear in May 2006.)

[4] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.

[5] V. Wuwongse et al., A Data Model for XML Databases, *Journal of Intelligent Information Systems* 20, 2003, pp. 63–80.

[2] ETI is an interpreter system that supports ET-based problem solving. It is available at http://assam.cims.hokudai.ac.jp/etpro.