The Squeeze Method: A Method for Program Construction in the Equivalent Transformation Computation Model

Kiyoshi Akama
Information Initiative Center
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan
Email: akama@iic.hokudai.ac.jp

Ekawit Nantajeewarawat
Sirindhorn International Institute of Technology
Thammasat University, Rangsit Campus
Pathumthani 12121, Thailand
Email: ekawit@siit.tu.ac.th

Hidekatsu Koike
Faculty of Social Information
Sapporo Gakuin University
Ebetsu, Hokkaido, 069-8555, Japan
Email: koike@sgu.ac.jp

Katsunori Miura
Faculty of Computer Science
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan
Email: kmiura@uva.cims.hokudai.ac.jp

Abstract—In the equivalent transformation (ET) computation model, a specification provides background knowledge in a problem domain, a program is a set of prioritized rewriting rules, and computation consists in successive reduction of problems by rule application. As long as meaning-preserving rewriting rules, called ET rules, with respect to given background knowledge are used, correct computation results are guaranteed. In this paper, a framework for program synthesis in the ET model is discussed. The framework comprises two main phases: (1) equivalent transformation of specifications, and (2) generation of a set of prioritized rewriting rules from a specification. A method for rule generation in the second phase, called the squeeze method, is presented. It constructs a program by accumulation of rules one by one on demand, with the goal of producing a correct, efficient, and non-redundant program.

Keywords—program synthesis; incremental program construction; equivalent transformation; rule generation

I. INTRODUCTION

Equivalent transformation (ET) is one of the most fundamental principles of computation, and it provides a simple and general basis for verification of correctness. Computation by ET was initially implemented in experimental natural language understanding systems during 1990–1992 [1], and the idea was further developed into a new computation model, called the ET model [2], [5]. A program in this model is a set of prioritized rewriting rules for meaning-preserving transformation of problems, and a problem solving process consists in successive rule application.

In contrast with declarative computation paradigms such as logic programming (LP) [7] and functional programming [6], programs are clearly separated from specifications in the ET model. A specification provides background knowledge for associating declarative meanings with problems and specifies a set of problems of interest. From a specification, meaning-preserving rewriting rules, called ET rules, are generated.

In this paper, a framework for program synthesis in the ET model is presented. The framework consists of two main phases: (1) equivalent transformation of specifications, and (2) generation of a set of prioritized ET rules from a specification. A demand-driven method for program synthesis in the second phase, called the squeeze method, is described. Based on some heuristics, which are given as its parameters, the method suggests patterns of atoms to be transformed in a problem solving process, and uses meta-computation techniques [4] for automatic generation of ET rules for the suggested atom patterns.

Although the ET model can deal with data structures of various kinds, the paper is deliberately confined to the domain of first-order terms for reasons of simplicity. The paper progresses from here as follows. Section II recalls the concepts of specification, program and computation in the ET model, and provides two basic examples, which are then used as running examples in subsequent sections. Section III describes the two-phase program synthesis framework and the squeeze method. Section IV compares program synthesis in the ET model with that in the LP model by viewing the latter as a special case of the former. This comparison clarifies the relationship between the ET model and program transformation in LP [10], and explains major advantages of the ET model over the LP model.

II. SPECIFICATIONS, PROGRAMS, AND COMPUTATION IN THE ET MODEL

Some basic concepts of the ET model along with examples, which will be used as running examples, are given first.
A. Basic Definitions

Specifications and Problems: A specification in the ET model is a pair \( \langle D, Q \rangle \), where \( D \) is a set of definite clauses, representing background knowledge, and \( Q \) is a set of problems, each of which is also a set of definite clauses. It is assumed that \( D \) is self contained and each problem in \( Q \) depends only on \( D \).

Given a set \( K \) of definite clauses, the meaning of \( K \), denoted by \( M(K) \), is the set \( \bigcup_{n=1}^{\infty} T_K^n(\emptyset) \), where \( T_K \) is the usual one-step consequence operator determined by \( K \) (see [7]). The answer set of a problem \( prb \) with respect to a specification \( \langle D, Q \rangle \) is defined as \( M(D \cup prb) - M(D) \), i.e., the set of all ground atoms in \( M(D \cup prb) \) whose predicates are defined by \( prb \).

Rewriting Rules: The class of rewriting rules used in this paper is now described. A rewriting rule \( r \) is an expression of the form

\[
H_s \Rightarrow \{E_{s_1}\}, B_{s_1}; \quad \Rightarrow \ldots \quad \Rightarrow \{E_{s_n}\}, B_{s_n},
\]

where \( n \geq 1 \), \( H_s \) is a nonempty sequence of atoms, the \( E_{s_i} \) are (possibly empty) sequences of executable atoms, and the \( B_{s_i} \) are (possibly empty) sequences of atoms. For each \( i \) \( (1 \leq i \leq n) \), the pair \( \{E_{s_i}\}, B_{s_i} \) is called a body of \( r \), and \( \{E_{s_i}\} \) and \( B_{s_i} \) are called an execution part and a replacement part, respectively. An execution part is optional. When \( H_s \) contains more than one atom, \( r \) is called a multi-head rule. It is called a single-head rule otherwise.

Variables used in rewriting rules and those used in definite clauses are of different types (they have different instantiation characteristics). For the sake of syntactically clear distinction, the former variables always have an asterisk prefixed to their names. Given a definite clause \( C \) containing atoms \( b_1, \ldots, b_m \), where \( m \geq 1 \), in its antecedent, the rule \( r \) is applicable to \( C \) at \( b_1, \ldots, b_m \) iff \( H_s \) matches these atoms by a substitution \( \theta \) (i.e., \( Hs\theta \) is the sequence \( b_1, \ldots, b_m \)). When applied, the pattern-matching substitution \( \theta \) is additionally required to instantiate all variables that occur in \( r \) but not in \( H_s \) into distinct usual variables that do not occur in \( C \), and the rule transforms \( C \) into at most \( n \) clauses in the following way: for each \( i \) \( (1 \leq i \leq n) \), if \( E_{s_i}\theta \) is executed successfully (by a given evaluator) and the execution yields an answer substitution \( \sigma \), then the clause obtained from \( C\sigma \) by replacing \( b_1\sigma, \ldots, b_m\sigma \) with \( B_{s_i}\theta\sigma \) is produced. The reader is referred to [3] for more precise operational semantics of rewriting rules.

Programs and Computation: A program is a set of prioritized rewriting rules. A program \( prg \) is said to be applicable to a problem \( prb \) iff some rule in the program is applicable to some definite clause in \( prb \). It is said to transform \( prb \) into \( prb' \) in one step iff

- \( prb' \) is obtained from \( prb \) by application of some rule \( r \) in \( prg \), and
- for any other rule \( r' \) in \( prg \), if \( r' \) is applicable to \( prb \), then \( r' \) does not take priority over \( r \).

Computation of a program takes a form of state transition, where problems are regarded as states. A final state is a problem that consists of only unit clauses. A computation of a program \( prg \) on a problem \( prb \) is a nonempty finite or infinite sequence \( com = \{st_0, st_1, st_2, \ldots \} \) of states such that \( st_0 = prb \) and the following conditions are satisfied:

1) For any two successive states \( st_i, st_{i+1} \) in \( com \), \( st_i \) is not a final state and \( prg \) transforms \( st_i \) into \( st_{i+1} \) in one step.

2) If \( com \) is finite, then last(\( com \)) is a final state or \( prg \) is not applicable to last(\( com \)), where last(\( com \)) denotes the last element of \( com \).

If \( com \) is finite and last(\( com \)) is a final state, then the answer set obtained from \( com \) is the set

\[ \{g \mid \langle(a \rightarrow) \in last(\com) \rangle \& \langle g \text{ is a ground instance of } a \rangle\}, \]

and it is undefined otherwise.

B. Examples

Assume as background knowledge a set \( D_{pal} \) consisting of the five definite clauses in Figure 1, where \( pal \), \( rv \) and \( ap \) stand for “palindrome”, “reverse” and “append”, respectively.

Running Example 1: Let \( QA \) be the set of all problems of the form

\[ \{yes \leftarrow pal(gl)\}, \]

where \( gl \) is a ground list of integers. Then the pair \( SA = \langle D_{pal}, QA \rangle \) is a specification. Consider a problem

\[ prb_1 = \{yes \leftarrow pal([1, 2, 1])\} \]

in \( QA \). With respect to \( SA \), the clause in \( prb_1 \) is intended to mean “the answer is yes if \([1, 2, 1] \) satisfies the definition of \( pal \) given in \( D_{pal} \)”; accordingly, \( prb_1 \) represents the query “Is the list \([1, 2, 1] \) a palindrome?” The answer set of \( prb_1 \) with respect to \( SA \) is the set

\[ M(D_{pal} \cup prb_1) - M(D_{pal}) = \{yes\}, \]

meaning that “yes” is the only answer to this problem.

The problem \( prb_1 \) can be solved by successive problem transformation using the rewriting rules in Figure 2. Let \( PA \) be a program consisting of these rules, where all of them have the same priority. Table I illustrates a computation of \( PA \) on \( prb_1 \), where atoms to which the rules are applied are
where \( m \) is arbitrary. Specialized rules are useful for this purpose.

For preventing infinite computation or improving computation efficiency, it is often possible to use rules with such wide applicability, it is often difficult to achieve good computation control, for example, for preventing infinite computation or improving computation efficiency. Specialized rules are useful for this purpose.

### Running Example II

Next, let \( Q_B \) be the set of all problems of the form

\[
\text{ans}(X) \leftarrow \text{pal}([a_1^1, a_2^1, \ldots, a_n^1 | X]), \\
\text{pal}([a_1^2, a_2^2, \ldots, a_n^2 | X]), \\
\ldots \\
\text{pal}([a_1^m, a_2^m, \ldots, a_n^m | X]),
\]

where \( m \geq 2, n_i \geq 1 \) for each \( i \) \( (1 \leq i \leq m) \), and the \( a_i^j \) are integers. (The predicate \( \text{ans} \) stands for “answer”.) Then the pair \( S_B = \langle D_{pal}, Q_B \rangle \) is a specification. Consider a problem

\[
\text{prb}_2 = \{ \text{ans}(X) \leftarrow \text{pal}([1 | X]), \text{pal}([2 | X]) \}
\]

in \( Q_B \). With respect to \( S_B \), the clause in \( \text{prb}_2 \) is intended to mean “\( X \) is an answer if both \([1 | X]\) and \([2 | X]\) satisfy the definition of \( \text{pal} \) given in \( D_{pal} \).” As such, \( \text{prb}_2 \) represents the problem “Find all ground terms \( t \) such that \([1 | t]\) and \([2 | t]\) are palindromes”. The answer set of \( \text{prb}_2 \) with respect to \( S_B \) is the set

\[
\mathcal{M}(D_{pal} \cup \text{prb}_2) - \mathcal{M}(D_{pal}) = \{ \text{ans}([]) \},
\]

meaning that the empty list is the only ground term satisfying the requirement.

The rules in Figure 3 are devised for finding this answer set. Let \( P_B \) be a program consisting of these rules, where all
rules except \( r_{ap} \) have the same priority and take priority over \( r_{ap} \). Table II shows a sequence of problem transformation steps by using \( P_B \). The transformation sequence changes the initial problem \( prb_2 \) into the singleton set \( \{ans([[]) \} \) \), which means "the empty list is an answer (unconditionally) to the problem and there exists no other answer". Again the correctness of this computation can be verified by proving that each rule in Figure 3 is a meaning-preserving rule with respect to \( D_{pal} \).

Applicability of rules is determined by pattern matching. All single-head rules in Figure 3, except \( r_{pal} \), are devised for atoms having specific patterns, and are called specialized rules. Employment of specialized rules enables content-based control of computation, i.e., appropriate transformation steps can be decided based on run-time content of computation states.

The program \( P_B \) can also compute the answer sets of several other problems in \( Q_B \) in finite time. For example, when applied to the problem

\[
prb_3 = \{ ans(X) \leftarrow pal([1, 2, 1|X]), pal([2|X]) \},
\]

\( P_B \) terminates with the final state \( \{ans([[]) \} \) \), and when applied to the problem

\[
prb_4 = \{ ans(X) \leftarrow pal([1, 2, 1|X]), pal([1, 3|X]) \},
\]

it terminates with the empty set, which denotes nonexistence of any answer, as the final state. However, some problem in \( Q_B \) has an infinite answer set; e.g. the answer set of the program

\[
prb_5 = \{ ans(X) \leftarrow pal([1|X]), pal([2|X]) \}
\]

contains infinitely many ground atoms, including

\[
[1], [2, 1], [2, 2, 1], [2, 2, 2, 1], [2, 2, \ldots, 2, 1].
\]

Since these atoms cannot be represented by a finite number of patterns, the answer set of \( prb_5 \) cannot be computed in finite time.

III. PROGRAM CONSTRUCTION USING THE SQUEEZE METHOD

To solve a problem correctly it is required that the answer set of the problem is computed in finite time. The requirement means not merely that every atom belonging to the answer set is found, but also that nonexistence of any other such atom is known in finite time. A program synthesis problem is described by: given a specification \( (D, Q) \), construct a program that can solve correctly as many problems as possible in the set \( Q \).

A. A Two-Phase Program Synthesis Framework

As outlined in Figure 4, program synthesis in the ET model consists of two main phases:

1) equivalent transformation of specifications,
2) generation of rewriting rules.

In the first phase, an initially given specification is transformed into an equivalent specification that has a more suitable form for generation of efficient rewriting rules. Transformation methods and strategies from research works on program transformation (e.g. [10]) and partial evaluation (e.g. [8], [9]) can be employed in this phase. The second phase is concerned with generating a set of prioritized rewriting rules from an obtained specification. The squeeze method is applied in this phase.

B. The Squeeze Method

The squeeze method is outlined in Figure 5. It constructs a program by accumulation of rules one by one on demand, with the goal of producing a correct, efficient, and non-redundant program. Heuristics are used for suggesting a suitable rule to be added in each iteration. They are given through the following three parameters:

[RUN] Control of execution at Step 1.
[PAT] Guidelines on determination of a general atom pattern at Step 2.2.

To ensure correct computation results, only meaning-preserving rewriting rules, called ET rules, with respect to given background knowledge are generated at Step 2.3. A rewriting rule is an ET rule with respect to a set \( D \) of definite clauses iff for any problems \( prb \) and \( prb' \), if the rule transforms \( prb \) into \( prb' \), then \( M(D \cup prb) = M(D \cup prb') \). An algorithm for generating ET rules, based on meta-computation, has been developed in [4]—given a set \( D \) of definite clauses and an atom pattern as inputs, the algorithm automatically generates
TABLE III
CONSTRUCTING $P_A$ (WITH REFERENCE TO THE STATES IN TABLE I)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Last state</th>
<th>Atom pattern</th>
<th>Rule obtained</th>
<th>Priority assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1</td>
<td>$pal(x)$</td>
<td>$r_{pal}$</td>
<td>PR-1</td>
</tr>
<tr>
<td>2nd</td>
<td>2</td>
<td>$rv(x, y)$</td>
<td>$r_{rv}$</td>
<td>PR-2</td>
</tr>
<tr>
<td>3rd</td>
<td>6</td>
<td>$ap(x, y, z)$</td>
<td>$r_{ap0}$</td>
<td>PR-2</td>
</tr>
<tr>
<td>4th</td>
<td>12</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

TABLE IV
CONSTRUCTING $P_B$ (WITH REFERENCE TO THE STATES IN TABLE II)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Last state</th>
<th>Atom pattern</th>
<th>Rule obtained</th>
<th>Priority assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1</td>
<td>$pal(x)$</td>
<td>$r_{pal}$</td>
<td>PR-1</td>
</tr>
<tr>
<td>2nd</td>
<td>3</td>
<td>$rv([a</td>
<td>x], y)$</td>
<td>$r_{rv}$</td>
</tr>
<tr>
<td>3rd</td>
<td>5</td>
<td>$rv(x, y), rv(x, z)$</td>
<td>$r_{rv2}$</td>
<td>PR-1</td>
</tr>
<tr>
<td>4th</td>
<td>6</td>
<td>$ap(x, y, [a</td>
<td>z])$</td>
<td>$r_{ap1}$</td>
</tr>
<tr>
<td>5th</td>
<td>7</td>
<td>$rv(x, [a</td>
<td>y])$</td>
<td>$r_{rv3}$</td>
</tr>
<tr>
<td>6th</td>
<td>8</td>
<td>$ap([a</td>
<td>x], [b</td>
<td>c</td>
</tr>
<tr>
<td>7th</td>
<td>9</td>
<td>$ap([a</td>
<td>x], [b</td>
<td>c])$</td>
</tr>
<tr>
<td>8th</td>
<td>10</td>
<td>$rv([</td>
<td>, x])$</td>
<td>$r_{rv4}$</td>
</tr>
<tr>
<td>9th</td>
<td>11</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

an ET rule with respect to $D$ for transforming atoms that conform to the input pattern.

In order to obtain an efficient program, rules with fewer bodies are preferable as they typically lead to shorter transformation sequences. Rules are prioritized accordingly at Step 2.4. By the “demand-driven” characteristic of the method, redundancy in a resulting program tends to be minimized.

C. Examples

Construction of the programs $P_A$ and $P_B$ used in Subsection II-B will now be illustrated.

Constructing the Program $P_A$: The following parameters are used:

[RUN] Usual rule selection based on rule priority is employed.

[TAR] Exactly one atom is selected as a target atom each time.

[ PAT] The most general atom pattern with the predicate symbol of the selected atom is determined.

Table III shows how $P_A$ is constructed using the squeeze method, with reference to the transformation sequence in Table I. Details are given as follows.

The 1st iteration: The initial program contains no rule; running the program makes no change to the initial problem $pr_{b1}$. The body atom $pal([1, 2, 1])$ is selected, and the atom pattern $pal(x)$ is determined. An ET rule for this pattern is generated, and $r_{pal}$ is obtained. Since it has a single body, assign a high priority level, say PR-1, to this rule.

The 2nd iteration: The current program contains only $r_{pal}$. Following the first transformation step in Table I, running this program with the initial problem, yields the second state in Table I as the last state. The body atom $rv([1, 2, 1], [1, 2, 1])$ is selected, and the general atom pattern $rv(x, y)$ is determined. An ET rule for this pattern is devised, and $r_{rv}$ is obtained. Since $r_{rv}$ has two bodies, a lower priority level, say PR-2, is assigned to it.

The 3rd iteration: Following the first five transformation steps in Table I, running the current program now results in the sixth state in Table I as the last state. The first $ap$-atom in this state is selected, and the general pattern $ap(x, y, z)$ is determined. An ET rule for this pattern is generated, and $r_{ap0}$ is obtained. Since $r_{ap0}$ has two bodies, the priority level PR-2 is assigned to it.

The 4th iteration: Running the current program yields the last state in Table I. Since it contains only unit clauses, the construction ends.

Constructing the Program $P_B$: The following parameters are used:

[RUN] Usual rule selection based on rule priority is employed under one constraint: employment of low-priority rules should be minimized.

[TAR] One or more atoms are selected each time, using the following guidelines:

- Select an atom that has a specific structure (e.g. $ap([1|X], Y, Z)$ is preferable to $ap(X, Y, Z)$ since $[1|X]$ is more specific than $X$).
- Select atoms that have common variables (e.g. $ap(X, Y, Z)$ and $ap(X, V, W)$ with $X$ as a common variable).
- A smaller number of selected atoms is preferable.

[ PAT] A more general pattern is preferable as long as it does not lead to a rule with a larger number of bodies.

As shown in Table IV, with these parameters the squeeze method produces the rules in Figure 3 within nine iterations.

The 1st iteration: The initial program contains no rule; running the program makes no change to the initial problem $pr_{b2}$. Either $pal([1|X])$ or $pal([2|X])$ may be selected, and the atom pattern $pal(x)$ is determined. An ET rule for this pattern is generated, and $r_{pal}$ is obtained. Since $r_{pal}$ has a single body, assign a high priority level, say PR-1, to it.

The 2nd iteration: The current program contains only $r_{pal}$. Following the first two transformation steps in Table II, running this program yields the third state in the table as the last state. The two body atoms in this state have the same pattern, and one of them is selected as the target atom. Set $rv([a|x], y)$ as the target atom pattern. Generate an ET rule for this pattern, and $r_{rv}$ is obtained. Since $r_{rv}$ has a single body, assign the priority level PR-1 to it.

The 3rd iteration: Following the first four transformation
steps in Table II, running the current program results in the fifth state in the table. Select the two rv-atoms in this state as the target atoms, and set the pair of rv(∗x, ∗y) and rv(∗x, ∗z) as the target pattern.\(^3\) Devise the multi-head ET rule \(r_{rv_3}\) for it. Again assign the priority level PR-1 to this rule.

**The 4th iteration:** Following the first five transformation steps in Table II, the current program now yields the sixth state as the last state. Select the first \(ap\)-atom in this state as the target atom, and set \(ap(∗x, ∗y, [∗a|∗z])\) as the target atom pattern. Generate an ET rule for this pattern; \(r_{ap_1}\) is obtained. Since it has more than one \(ap\) to body, assign a lower priority level, say PR-2, to this state as the target atom, and set \(r_{rv_3}\) as the last state. Select the first \(ap\)-atom in this state as the target atoms, and set the pair of \(ap(∗x, ∗y, [∗a|∗z])\) and \(ap(∗x, ∗w, [∗b|∗z])\) as the target pattern. Devise this state as the target pattern. Generate an ET rule for this.

**The 5th iteration:** By the first six transformation steps of Table II, the current program transforms \(prb_2\) into the seventh state in the table. By the constraint imposed upon by the parameter [RUN], although this state can be transformed further using \(r_{ap_1}\), this transformation step is not made. Instead, a new rule is constructed. The first \(rv\)-atom in the second clause of this state is selected as the target atom, and the atom pattern \(rv(∗x, [∗a|∗y])\) is determined. The ET rule \(r_{rv_3}\) is then generated, and the priority level PR-1 is assigned to it.

**The 6th iteration onwards:** By following the squeeze method three more iterations, the ET rules \(r_{ap_2}, r_{ap_3}\), and \(r_{rv_3}\) are generated and added to the program in succession. The priority level PR-1 is given to each of them. When running the resulting program with the input problem \(prb_2\), a problem consisting only of unit clauses is obtained, and the construction ends.

**D. Improving the Efficiency of an Obtained Program**

In general, rules with fewer bodies lead to more efficient computation. For example, suppose that the multi-head single-body rule \(r_{ap_1}\), in Figure 6 is used instead of the multi-body rule \(r_{ap_1}\) in the sixth transformation step in Table II, then a shorter sequence of transformation steps shown in Table V will be obtained.

The rule \(r_{ap_4}\) can be obtained using the squeeze method by changing the parameter [RUN] into:

Only single-body rules (i.e., rules with the highest priority level) are used.

Then, since selection of a single \(ap\)-atom in the fourth iteration in Table IV results in a two-body rule (i.e., \(r_{ap_1}\), a search for alternative selection should be made in that iteration. The

\(^3\)According to [TAR], any of the \(ap\)-atoms in this state is an alternative choice. However, selection of such an \(ap\)-atom would result in a two-body rule (see the 4th iteration). By selecting the two \(rv\)-atoms, a single-body rule, which is preferable, is obtained.
ET model as regards program synthesis will be less difficult and major differences between them can be seen more clearly when discussion is made based on their common structure and trivial differences between them are omitted. Having this in mind, in this section, it will be assumed that

- a specification in the LP model as well as that in the ET model is a set of definite clauses, rather than a pair of a set of definite clauses and a set of problems of interest;\(^5\)
- a program in the LP model is a set of definite clauses—a selection function is not taken into consideration;
- a program in the ET model is a set of prioritized ET rules—a selection function is not taken into consideration.\(^6\)

Table VII summarizes the terminology.

### B. Viewing Logic Programs in the ET Framework

When computation in LP is viewed in the framework of ET computation, expansion of a node (generation of all of its children) in a search tree for finding SLD-refutations corresponds to an unfolding transformation step. When the role of a selection function is not taken into account, computation using a logic program can therefore be regarded as computation using general unfolding-based rules. Accordingly, a program \(P\) (a set of definite clauses) in the LP model corresponds to a program \(P'\) (a set of ET rules) in the ET model such that

- \(P'\) contains one general unfolding-based rule for each predicate defined by \(P\), and contains nothing else;
- all rules in \(P'\) have the same priority level.

The first row of Table VIII describes the concepts of specification and program in LP model when they are viewed in the ET framework.

<table>
<thead>
<tr>
<th>Model</th>
<th>Specification</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>a set of definite clauses</td>
<td>a set of definite clauses</td>
</tr>
<tr>
<td>ET</td>
<td>a set of definite clauses</td>
<td>a set of prioritized ET rules</td>
</tr>
</tbody>
</table>

**TABLE VII**

**COMPARISON OF CONCEPTS**

<table>
<thead>
<tr>
<th>Model</th>
<th>Specification</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>a set of definite clauses</td>
<td>a set of general unfolding-based rules</td>
</tr>
<tr>
<td>ET</td>
<td>a set of definite clauses</td>
<td>a set of prioritized ET rules</td>
</tr>
</tbody>
</table>

**TABLE VIII**

**VIEWING LP CONCEPTS IN THE ET MODEL**

C. Inclusion of LP Program Synthesis in the ET Program Synthesis Framework

In this view, LP program synthesis can be considered as a special case of ET program synthesis (Figure 4), where only general unfolding-based rules are produced in the second phase (rule generation). Since a set of definite clauses determines a unique (up to variable renaming) set of general unfolding-based rules, rule generation in this case is very simple. Being restricted to only this specific class of rewriting rules, improvement of programs in LP can be made only in the first phase (equivalent transformation of specifications).\(^7\) No search for more effective rewriting rules is made and thus the power of the rule generation phase is not used.

Referring to Running Examples I and II, two concrete examples illustrating inclusion of LP program synthesis in ET program synthesis and the power of the rule generation phase will now be given. The first example explains that all program transformation techniques in LP can be used in ET program synthesis, and at least the same efficiency improvement can be obtained. The second example uses the problem \(prb_2\) to show that, compared to program transformation in LP, the rule generation phase in the ET framework can yield very different program improvement results.

**Example 1:** Let \(P_1\) and \(P_2\) be the sets consisting of the definite clauses in Figures 7 and 8, respectively. It is demonstrated in [10] that \(P_1\) can be transformed into \(P_2\) by application of the folding, the unfolding, and the goal replacement transformation rules, along with the tupling and the generalization strategies, and for testing whether a given ground list is a palindrome, \(P_2\) is more efficient than \(P_1\) when they are regarded as logic programs.\(^5\) As depicted by Figure 9, this transformation corresponds to equivalent

\[
\begin{align*}
pal([],[]) & \leftarrow \\
pal([H|L]) & \leftarrow \\
pal([H|L]) & \leftarrow \text{newp}(H,L) \\
\text{newp}(L,L) & \leftarrow \\
newp(H,L) & \leftarrow \\
\text{newp}(H,L,U) & \leftarrow 
\end{align*}
\]

Fig. 7. A definition of \(pal\) (\(P_1\))

\[
\begin{align*}
pal([],[]) & \leftarrow \\
pal([H]) & \leftarrow \\
pal([H|T]) & \leftarrow ap(Y,[H,T],pal(Y)) \\
ap([\text{false}],X,X) & \leftarrow \\
ap([\text{true}],[A|X],Y,[A[Z]]) & \leftarrow ap(X,Y,Z)
\end{align*}
\]

Fig. 8. An equivalent definition of \(pal\) (\(P_2\))

\(^5\)A set of problems of interest is often not taken as part of a specification in the LP model.

\(^6\)The ET model normally uses content-based control of computation, rather than fixed control by employment of a selection function.

\(^7\)Note that since a set of definite clauses is also regarded as a program in LP, equivalent transformation in the first phase is normally called program transformation.

\(^8\)By using \(P_2\), double visits of lists can be avoided, and the time complexity is improved from \(O(n^2)\) to \(O(n)\), where \(n\) is the size of the input list.
transformation in the first phase of Figure 4. Generation of general unfolding-based rules from $P_2$ yields the rules in Figure 10, and computation by $P_2$, when it is regarded as a logic program, corresponds to computation by using these rules. Employment of a larger class of rules facilitates further program improvement. Figure 11 provides examples of specialized rules that can be generated from $P_2$. When a low priority level is given to the last rule, these specialized rules use $(n+1)/2$ steps for verifying a ground $n$-element palindrome list, while the general rules in Figure 10 use $n+1$ transformation steps. It is worth remarking that although, in this case, similar efficiency gain may be obtained in Prolog by using the cut operator, 3 incorporation of such a non-logical construct makes correctness verification in LP problematic and greatly increases the difficulty of program synthesis in that context.

Advantages of the rule generation phase can be seen more obviously in the next example.

**Example 2:** When applied the problem prb2 (“Find all ground terms $t$ such that $[1|t]$ and $[2|t]$ are palindromes”), which is represented in LP as the query clause

\[
\text{← pal}([1|X]), \text{pal}([2|X]),
\]

both $P_1$ and $P_2$, when regarded as logic programs, enter infinite computation after giving $X = [\]$. As such, none of them yields the correct answer set, i.e., they all fail to infer

\[
\begin{align*}
r'_{pal1}: & \quad \text{pal}([]) \Rightarrow. \\
r'_{pal2}: & \quad \text{pal}([a]) \Rightarrow. \\
r'_{pal3}: & \quad \text{pal}([a,b|x]) \Rightarrow \text{newp}([a], [b|x]). \\
r'_{newp1}: & \quad \text{newp}(x, x) \Rightarrow. \\
r'_{newp2}: & \quad \text{newp}(x, [a|x]) \Rightarrow. \\
r'_{newp3}: & \quad \text{newp}(x, [a|y]) \Rightarrow \text{newp}([a|x], y).
\end{align*}
\]

Fig. 11. Examples of specialized rules obtained from $P_2$

that the empty list is the “only” possible ground instance of $X$ that satisfies the query. Transformation of $P_1$ into $P_2$ does nothing towards a solution to this non-termination problem.

Indeed, any logic program obtained from $P_1$, $P_2$, or the definition of $\text{pal}$ in Figure 1 by means of program transformation backtracks infinitely many times when it answers the query $\text{← pal}([1|X]), \text{pal}([2|X])$. The reason is that the two subgoals are processed one by one sequentially, and the first subgoal, $\text{pal}([1|X])$, generates infinitely many patterns of instances of $X$. With the possibility of employing a larger rule class, including specialized rules and multi-head rules, and the possibility of searching for appropriate rules in the rule generation phase (i.e., by using the squeeze method), it has been shown in Subsection III-C that this difficulty can be overcome.

Attention will now be drawn to the role of the multi-head rule $r_{rv2}$ (in Figure 3) in successful termination of the transformation sequence in Table II. Consider the two $\text{pal}$-atoms, $\text{pal}([1|X])$ and $\text{pal}([2|X])$, in the body of the initial clause, and the case when $X$ is instantiated into a nonempty ground list, say $l_X$. By the definition of $\text{pal}$, the first $\text{pal}$-atom restricts the last element of $l_X$ to 1, whereas the second one does that to 2. This contradiction proves nonexistence of any answer when $X \neq [\]$. In terms of computation, finding this contradiction involves exchange of information about the restriction on the last element of $l_X$ between descendants of the first $\text{pal}$-atom and those of the second $\text{pal}$-atom. This form of restriction, however, cannot be represented using a finite information pattern. For example, making the information “the last element of $l_X$ must be 1” available to a certain part of a clause entails passing to it infinitely many patterns such as $X = [1], X = [Y1, 1], X = [Y1, Y2, 1], X = [Y1, Y2, Y3, 1]$, and so forth. Consequently, computation becomes infinite; a contradiction must be found for each such pattern, one at a time.

The multi-head rule $r_{rv2}$ provides a simple remedy—it creates an additional information connection through which the constraint on the last element of $l_X$ can be passed as a finite pattern. Referring to Table II, application of $r_{rv2}$ to the fifth state results in the clause

\[
\text{ans}(X) \leftarrow \text{rv}(X, A1), \text{ap}(A1, [1], [1|X]), \text{ap}(A1, [2], [2|X]),
\]

in which the constraint “the last element of $l_X$ must be 1” can be represented using a single pattern such as $A1 = [1|Y]$, based on the definition of the predicate $\text{rv}$ (“reverse”). With the information connection via $A1$, the all-embracing
contradiction can be found in finite time.

V. 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. The ET model is developed with an aim of extending such a correct-program space. Clear-cut separation between specifications and programs opens up the possibility of such extension. A specification provides background knowledge for declaratively defining the answers to problems, whereas a program consists of 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 meaning-preserving transformation with respect to given background knowledge can serve as an ET rule. As a consequence, various classes of rules, with varying expressive power, can be introduced. For example, in the Equivalent Transformation Interpreter (ETI) system\textsuperscript{10} developed at Hokkaido University, rules with guard conditions (possibly involving non-logical predicates), rules with execution parts (possibly including non-logical constructs), and multi-head rules are provided. Program synthesis in the ET model consists of two main phases—equivalent transformation of specifications, followed by generation of a set of prioritized ET rules from an obtained specification. Incorporation of the second phase makes program synthesis in the ET model significantly different from that in declarative computation paradigms, wherein specifications are regarded as programs and, accordingly, the possibility for enhancement of programs using this phase is excluded. Since many kinds of ET rules can be generated by meta-computation based on the general principle of ET, this framework provides a systematic and powerful method of program synthesis.

References


\textsuperscript{10}ETI is an interpreter system that supports ET-based problem solving. It is available at http://assam.cims.hokudai.ac.jp/etpro.