An Inductive Inference Algorithm to Synthesize Prolog Programs from Specification by Example

AKIHIKO NAKASE*,**, YOSHIAKI FUKAZAWA* and TOSHIO KADOKURA*

By utilizing an inductive inference method, we developed an automatic programming system LIPS (LIst-processing Program Synthesizer). This system synthesizes Prolog programs from its specification by example.

LIPS employs three unique data structures in the program synthesis process. They are Data Metamorphosis History (DMH), Constant Argument Clause Set (CACS), and Variable Argument Clause Set (VACS). DMH shows the runtime behavior of arguments of the specification by example, when it was evaluated by a target program. CACS is generated from DMH and shows the runtime behavior of the target program when the data of the specification by example is entered. VACS, which shows the general runtime behavior of the target program, is a general form of CACS.

In this paper, we show the details of these three data structures and some algorithms in order to generate them from the specification by example. The comparison of LIPS with other systems and future prospects of LIPS are also described.

1. Introduction

We developed an inductive inference system which automatically synthesizes list-processing programs of Prolog, using its specification by example. We named our system LIPS (LIst-processing Program Synthesizer). A specification by example, we mean here, is both input examples to the target program, and expected outputs from it. For example, if we wish to synthesize a program which appends two lists, we may write a specification by example as follows:

"append([a, b, c], [d, e, f], [a, b, c, d, e, f])".

There are many automatic programming systems for Lisp functions using specification by example [1]-[5]. But few studies were done for Prolog programs. In Lisp, there is a precise distinction between input lists and output list, therefore we can predict the behavior of a function how it transforms inputs to output. In Prolog programs, however, there are no input-output distinctions. Therefore we cannot use the strategies applied to Lisp functions.

In order to cope with it, we developed a new algorithm which can predict the behavior of programs regardless of the input-output distinction.

A good inductive inference system which synthesizes Prolog programs from its specification by example, is reported by Shapiro[6]. Comparison of his system with ours is argued in Chapter 4.

*Department of Electrical Engineering, School of Science and Engineering, Waseda University, Okubo 3-4-1, Shinjuku-ku, Tokyo 160, Japan.
**Now with Information Systems Laboratory, Toshiba Research and Development Center.

2. System Configuration and Its Behavior

Figure 1 is the block diagram of LIPS.

In LIPS, specification by example is transformed in four stages. At first, specification by example is transformed into Data Metamorphosis History (DMH) using Data Metamorphosis Rules (DMR). Secondly, DMH is transformed into Constant Argument Clause Set (CACS). Thirdly, CACS is transformed into Variable Argument Clause Set (VACS). Lastly, VACS is transformed into a target Prolog program. In this sec-

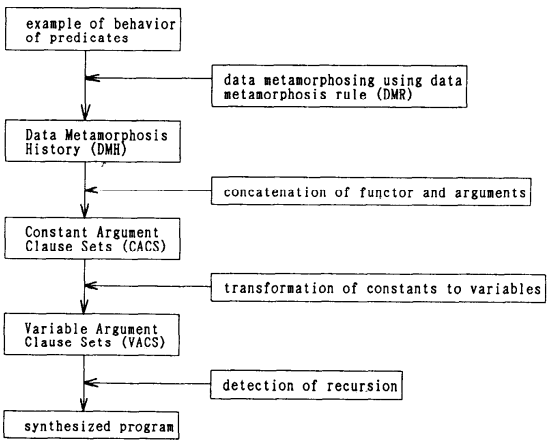

Fig. 1 Block Diagram of LIPS.

tion, each transformation process is explained by assuming that specification by example is

"append([a, b, c], [d, e, f], [a, b, c, d, e, f])". (a)

2.1 Generation of DMH

LIPS transforms the argument data of specification by example, and makes Data Metamorphosis History (DMH). Argument data means constant arguments in specification by example. In the above example, argument data is "[a, b, c], [d, e, f], [a, b, c, d, e, f]".

In order to transform argument data to DMH, LIPS utilizes the following four fundamental rules.

(1) If the heads of some lists in argument data are identical, remove them.

(2) If the tails of some lists in argument data are same, remove them.

(3) If neither (1) nor (2) is applicable, separate one list into head part and tail part. (This rule has some restriction. See Appendix.)

(4) If none of (1)–(3) is applicable, stop to apply rules.

Complete Data Metamorphosis Rules (DMR) are listed in Appendix. When one rule is applied, that rule and argument data should be stored in DMH.

For the example (a), first of all, argument data of specification by example is extracted. Next, argument data is transformed by means of one of above four rules. Rules (1) and (2) are applicable to argument data, because heads of first and third argument are the same and tails of second and third argument are the same. In this case, we choose rule (1). (Rule (2) can be also applied and the results of the program synthesis are the same. For clear explanation of the following steps, we choose rule (1).) After application of it, argument data is transformed into "[b, c], [d, e, f], [b, c, d, e, f]". In the next step, by the same transformation, argument data comes to be "[c], [d, e, f], [c, d, e, f]". By similar operations, argument data lastly comes to be "[ ], [ ], [ ]". Details of transformation are listed below.

To record the process of data transformation (which rules are applied and which lists' elements are removed) and the results of transformation, we enumerate argument data before/after each rule is applied. Generally, by our data metamorphosis rules, some heads/tails of lists are removed or divided. So for each step of transformation, the argument data before transformation is first enumerated. In this argument data, elements which will be removed or divided are specified using delimiter "|". Next, we enumerate the argument data after transformation. By repeating these two kinds of enumeration for each transformation step, we can obtain the following DMH.

[a, b, c], [d, e, f], [a, b, c, d, e, f]	specification by example
1st transformation.	
[a|[b, c]], [d, e, f], [a|[b, c, d, e, f]]	argument data before rule(1) is applied
[b, c], [d, e, f], [b, c, d, e, f]	argument data after rule(1) is applied
2nd transformation.	
[b|[c]], [d, e, f], [b|[c, d, e, f]]	argument data before rule(1) is applied
[c], [d, e, f], [c, d, e, f]	argument data after rule(1) is applied
3rd transformation.	
[c|[ ]], [d, c, f], [c|[d, e, f]]	argument data before rule(1) is applied
[ ], [d, e, f], [d, e, f]	argument data after rule(1) is applied
4th transformation.	
[ ], [d|[e, f]], [d|[e, f]]	argument data before rule(1) is applied
[ ], [e, f], [e, f]	argument data after rule(1) is applied
5th transformation.	
[ ], [e|[f]], [e|[f]]	argument data before rule(1) is applied
[ ], [f], [f]	argument data after rule(1) is applied
6th transformation.	
[ ], [f|[ ]], [f|[ ]]	argument data before rule(1) is applied
[ ], [ ], [ ]	argument data after rule(1) is applied
end of transformation.	
[ ], [ ], [ ]	no rules can be applied

Generally, DMH shows the following configuration.

specification by example.
argument data before a rule is applied.
argument data after the rule is applied.
...
argument data to which no rules can be applied.

2.2 Generation of CACS

At the next step, LIPS transforms DMH into Constant Argument Clause Set (CACS). In this step, each argument data of DMH is inserted in some functor. For that purpose, unique functors must be generated. In our system, unique functors are generated by concatenating a functor of specification by example and an integer. In the example of "append([a, b, c], [d, e, f], [a, b, c, d, e, f]).", the base functor is "append", so the generated functors used in CACS will be "append1", "append2" and so on. From now on, we call the functor of CACS "predicate-symbol-subscript", whose "predicate-symbol" means the functors of specification by example, and "subscript" means the integer to be concatenated to the functor of specification by example.

First, we make a predicate whose functor is "predicate-symbol-1" and the arguments are the last data of DMH, and enumerate it to CACS. In the above example, it is

"append1([ ], [ ],[ ])".

Next, we repeatedly make clauses whose style is as follows, starting from the bottom of DMH to the top of DMH, and enumerate them into CACS.

predicate-symbol-($n+1$)
(argument data before a rule is applied):-
predicate-symbol-n
(argument data after the rule is applied).

Subscript "n" should be incremented by one in each clause of CACS. In the above example, next clause to be generated is

"append2([ ], [f|[ ]], [f|[ ]]):-append1([ ], [ ],[ ]).".

And lastly, we append a rule whose head is specification by example, and whose body is "predicate-symbol-nmax(argument data of specification by example)". This nmax means maximum number of subscript used in CACS. In this time, a rule of DMR applied in each transformation step is added to each clauses in CACS. Therefore, from the above example, following CACS will be obtained.

```
append1([ ], [ ], [ ]).
                (DMR(4) is applied)
append2([ ], [f|[ ]], [f|[ ]]):-append1([ ], [ ], [ ]).
                (DMR(1) is applied)
append3([ ], [e|[f]], [e|[f]]):-append2([ ], [f], [f]).
                (DMR(1) is applied)
append4([ ], [d|[e, f]], [d|[e, f]]):-
                              append3([ ], [e, f], [e, f]).
                (DMR(1) is applied)
append5([c|[ ]], [d, e, f], [c|[d, e, f]]):-
                        append4([ ], [d, e, f], [d, e, f]).
                (DMR(1) is applied)
append6([b|[c], [d, e, f], [b|[c, d, e, f]]):-
                    append5([c], [d, e, f], [c, d, e, f].
                (DMR(1) is applied)
append7([a|[b, c], [d, e, f], [a|[b, c, d, e, f]]):-
                append6([b, c], [d, e, f], [b, c, d, e, f]).
                (DMR(1) is applied)
append([a, b, c], [d, e, f], [a, b, c, d, e, f]):-
            append7([a, b, c], [d, e, f], [a, b, c, d, e, f]).
```

Generally, CACS has the following form.

predicate-symbol-1
(argument data to which no DMR can be applied).
(applied rule of DMR)
predicate-symbol-2
(argument data before some DMR are applied):-
predicate-symbol-1
(argument data after some DMR is applied).
(applied rule of DMR)
predicate-symbol-n(argument data before some DMR is applied):-predicate-symbol-($n-1$)(argument data after some DMR is applied).
(applied rule of DMR)
specification-by-example:-predicate-symbol-n(argument data of specification by example).

2.3 Generation of VACS

From CACS, Variable Argument Clause Set (VACS) is generated. We make no change to a clause which has no body in CACS and whose arguments are null lists ([ ]) only. So in the above example, "append1([ ], [ ], [ ])." is not changed at all. If they are some lists with some literals, we change them to some appropriate variables.

For a clause which has a body, we translate its arguments by the following ways. As mentioned above, each clause of CACS has a description showing which rule of DMR is applied. For the clause with DMR(1) (or DMR(2)), heads (or tails) of some argument which is explicitly divided by "|" are removed. So we translate them to a variable of same variable symbol. Symbol of the variable is generated by concatenating some base variable symbol (for example "A") and an integer.

For the clause with DMR(3), head of lists in head clause and head of lists in body clause which are explicitly divided by "|" should be translated into variables in same variable symbol.

After that, constant lists should be translated into variables. In this time, n-th argument of head of the clause and n-th argument of body of the clause should be translated into same variable symbols. Variable symbols are generated by concatenating a base variable symbol (for example "X") and subscript "n" (which indicates n-th argument).

In the above example,

"append2([ ], [f|[ ]], [f|[ ]]):-append1([ ], [ ], [ ])."

is first translated into

"append2([ ], [A1|[ ]], [A1|[ ]]):-
append1([ ], [ ], [ ]).",

is then translated into

"append2(X1, [A1|X2], [A1|X3]):-
append1(X1, X2, X3).".

Finally, in the above example, following VACS can be obtained from CACS.

```
append1([ ], [ ]. [ ]).                                  (2.3.1)
append2(X1, [A1|X2], [A1|X3]):-
                        append1(X1, X2, X3).             (2.3.2)
append3(X1, [A1|X2], [A1|X3]):-
                        append2(X1, X2, X3).             (2.3.3)
append4(X1, [A1|X2], [A1|X3]):-
                        append3(X1, X2, X3).             (2.3.4)
```

```
append5([A1|X1], X2, [A1|X3]):-
                append4(X1, X2, X3).   (2.3.5)
append6([A1|X1], X2, [A1|X3]):-
                append5(X1, X2, X3).   (2.3.6)
append7([A1|X1], X2, [A1|X3]):-
                append6(X1, X2, X3).   (2.3.7)
append(X1, X2, X3):-append7(X1, X2, X3).   (2.3.8)
```

Generally, VACS has similar form to CACS, provided arguments of its element are translated to variables.

2.4 Generation of Target Program

A target program is synthesized from VACS by executing the following three steps.

(1) Finding recursive clause set

In VACS, a set of successive clauses which have the same pattern is regarded as a recursive clause set. The same pattern of clauses, we defined, are clauses where all of which have same arguments in the head and the body. In the example of "append", clauses (2.3.2), (2.3.3) and (2.3.4) are in successive position in VACS, and arguments of the heads of three clauses are the same. Also bodies are. Finally, from the above example, we can find two recursive clause sets, namely {(2.3.2), (2.3.3), (2.3.4)} and {(2.3.5), (2.3.6), (2.3.7)}.

(2) Creation of self-recursive clauses

The recursive clause set discovered in (1) should have the following format.

```
predicate-symbol-(n+1)(argument set 1):-
                predicate-symbol-n(argument set 2).
predicate-symbol-(n+2)(argument set 1):-
                predicate-symbol-(n+1)(argument set 2).
predicate-symbol-(n+3)(argument set 1):-
                predicate-symbol-(n+2)(argument set 2).
                ...
```

In order to make a self-recursive clause, "argument set 1", "argument set 2", "predicate-symbol", and "n" are utilized from the above recursive clause set. The self-recursive clause we generated from above recursive clause set has the following format:

```
predicate-symbol-r-n(argument set 1):-
                predicate-symbol-r-n(argument set 2).
```

From the examples of recursive clause sets of "append", following self-recursive clauses are generated.

```
appendr1(X1, [A1|X2], [A1|X3]):-
                appendr1(X1, X2, X3).   (2.4.1)
appendr4([A1|X1], X2, [A1|X3]):-
                appendr4(X1, X2, X3).   (2.4.2)
```

(3) Replacement to self-recursive clauses

Generated self-recursive clauses are inserted into the recursive predicate sets in VACS. Now, the recursive clause set has a following format:

```
predicate-symbol-(n+1)(argument set 1):-
                predicate-symbol-n(argument set 2).
predicate-symbol-(n+2)(argument set 1):-
                predicate-symbol-(n+1)(argument set 2).
                ...
predicate-symbol-k(argument set 1):-
                predicate-symbol-(k-1)(argument set 2).
```

and self-recursive clause:

```
predicate-symbol-r-n(argument set 1):-
                predicate-symbol-r-n(argument set 2).
```

then we make the following three clauses.

```
predicate-symbol-r-n(argument set):-
                predicate-symbol-n(argument set).
predicate-symbol-r-n(argument set 1):-
                predicate-symbol-r-n(argument set 2).
predicate-symbol-k(argument set):-
                predicate-symbol-r-n(argument set).
```

And corresponding clauses of the recursive clause set should be deleted. In the first and third clauses, "argument set" should be decided by concatenation of the variable symbol and an integer which specifies the position of each argument. And "argument set" of the body of the clause should be the same as that of the head of the clause. Therefore from recursive clause set {(2.3.2), (2.3.3), (2.3.4)} and self-recursive predicate (2.4.1), the following clause set can be generated.

```
appendr1(X1, X2, X3):—append1(X1, X2, X3).
appendr1(X1, [A1|X2], [A1|X3]):-
                appendr1(X1, X2, X3).
append4(X1, X2, X3):-appendr1(X1, X2, X3).
```

Finally, we can get the following "append" program from VACS, including two self-recursive clauses.

```
append1([ ], [ ], [ ]).
appendr1(X1, X2, X3):-append1(X1, X2, X3).
appendr1(X1, [A1|X2], [A1|X3]):-
                appendr1(X1, X2, X3).
append4(X1, X2, X3):-appendr1(X1, X2, X3).
appendr4(X1, X2, X3):-append4(X1, X2, X3).
appendr4([A1|X1], X2, [A1|X3]):-
                appendr4(X1, X2, X3).
append7(X1, X2, X3):-appendr4(X1, X2, X3).
append(X1, X2, X3):-append7(X1, X2, X3).
```

3. Expansion of LIPS

In above discussions, LIPS is dealing with only "structure of lists". But in real applications of the program synthesis, it is very inconvenient to deal with only structure of lists. Therefore we expand the domain of target program by modifying the transformation algorithm of LIPS. In this expansion, we make LIPS deal with not only structure of lists, but integers as counters, and literals which have special attributes.

3.1 Dealing with Integers as Counters

When we consider the program which checks the length of lists, we can think about its example as

"length([a, b, c, d,e, f], 6)"

or

"len([a, b, c, d], 4)".

LIPS can provide some methods to synthesize such programs. For that purpose, we make some expansion on making DMH, CACS and VACS. Suppose that specification by example is

"len([a, b, c, d], 4)".

In making DMH, three rules are added to Data Metamorphosis Rules (DMR). These are,

(5) If an integer is zero, we make no changes to this integer.

(6) If an integer is not zero, then this integer is decremented by one.

(7) Rule (6) should be applied accompanied with rules (1)-(3). DMH will be as follows.

[a, b, c, d],	4	specification by example
1st transformation		
[a|[b, c, d]],	4	argument data before rules (1) and (6) are applied.
[b, c, d],	3	argument data after rules (1) and (6) are applied.
2nd transformation		
[b|[c, d]],	3	argument data before rules (1) and (6) are applied.
[c, d],	2	argument data after rules (1) and (6) are applied.
3rd transformation		
[c|[d]],	2	argument data before rules (1) and (6) are applied.
[d],	1	argument data after rules (1) and (6) are applied.
4th transformation		
[d|[ ]],	1	argument data before rules (1) and (6) are applied.
[ ],	0	argument data after rules (1) and (6) are applied.
end of transformation		
[ ],	0	no rules can be applied

Next, we make CACS. By our algorithms above, CACS will be as follows. Rules of DMR attached to each clause are omitted for clear explanation.

```
len1([ ], 0).
len2([d|[ ]], 1):-len1([ ], 0).
len3([c|[d]], 2):-len2([d], 1).
len4([b|[c, d]], 3):-len3([c, d], 2).
len5([a|[b, c, d]], 4):-len4([b, c, d], 3).
len([a, b, c, d], 4):-len5([a, b, c, d], 4).
```

Then arithmetic predicate "is" is inserted after ":-" in each clause where a decrement of integer occurs. This format "is" is "(integer of body) is (integer of head)-1".

Therefore CACS is changed as follows.

```
len1([ ], 0).
len2([d|[ ]], 1):-0 is 1−1, len1([ ], 0).
len3([c|[d]], 2):-1 is 2−1, len2([d], 1).
len4([b|[c, d]], 3):-2 is 3−1, len3([c, d], 2).
len5([a|[b, c, d]], 4):-3 is 4−1, len4[b, c, d], 3).
len([a, b, c, d], 4):-len5([a, b, c, d], 4).
```

Then we make VACS. In order to change integers to variables, we firstly choose different variable symbols of the head and the body. For example, I1 for the head and I2 for the body. When we change integers and lists in clauses, decrement operator "−1" should not be changed into a variable.

Then, we can get the following VACS.

```
len1([ ], 0).
len2([A1|X1], I1):-I2 is I1−1, len1(X1, I2).
len3([A1|X1], I1):-I2 is I1−1, len2(X1, I2).
len4([A1|X1], I1):-I2 is I1−1, len3(X1, I2).
len5([A1|X1], I1):-I2 is I1−1, len4(X1, I2).
len(X1, X2):-len5(X1, X2).
```

Finally, the target program is generated by finding recursive clauses. Generated program is as follows.

```
len1([ ], 0).
lenr1(X1, X2):-len1(X1, X2).
lenr1([A1|X1], I1):-I2 is I1−1, lenr1(X1, I2).
len5(X1, X2):-lenr1(X1, X2).
len(X1, X2):-len5(X1, X2).
```

3.2 Dealing with Literals Which Have Special Attributes

Up to this point, we are only dealing with predicates which manipulate the structure of lists. Namely, literals of each list can be substituted for any literals. However if we consider a predicate which deletes all literals after the literal "*" in the first argument, and makes it the second argument, specification by example may be

"del([a, b, c, *, e, f], [a, b, c])".

In this specification by example, "a", "b" and so on can be substituted to any literals, but "*" should not. So in this sense, we call "*" a literal which has a special attribute. In dealing with such attributes, what we must consider is how we can specify such literals with special attributes, and how we can transform them from CACS to VACS. In the specification by example, literals which have special attributes should be written between "{" and "}". For example, above "delete" predicate should be specified by

"del([a, b, c, {*}, e, f], [a, b, c])".

By generating DMH and CACS, we can get the following CACS. For clear explanation, rules of DMR are omitted.

```
del1([f, e, {*}], [ ], [ ]).
del2([e, {*}], [f|[ ]], [ ]):-
                del1([f|[e, {*}]], [ ], [ ]).
del3([{*}], [e|[f]], [ ]):-
                del2([e|[{*}]], [f], [ ]).
del4([{*}|[e, f]], [ ]):-
                del3([{*}], [e, f], [ ]).
del5([c|[{*}, c, f]], [c|[ ]]):-
                del4([{*}, e, f], [ ]).
del6([b|[c, {*}, e, f]], [b|[c]]):-
                del5([c, {*}, e, f], [c]).
del7([a|[b, c, {*}, e, f]], [a|[b, c]]):-
                del6([b, c, {*}, e, f], [b, c]).
del([a, b, c, {*}, e, f], [a, b, c]):-
                del7([a, b, c, {*}, e, f], [a, b, c]).
```

When we change the lists in CACS to variables in VACS, we do not change the lists or literals with "{" and "}" which is to be removed or appended to some lists. "{" and "}" should be removed in VACS. Resulting VACS will be as follows.

```
del1(X1, [ ], [ ]).
del2(X1, [A1|X2], X3):-del1([A1|X1], X2, X3).
del3(X1, [A1|X2], X3):-del2([A1|X1], X2, X3).
del4([*|X1], X2):-del3(X3, X1, X2).
del5([A1|X1], [A1|X2]):-del4(X1, X2).
del6([A1|X1], [A1|X2]):-del5(X1, X2).
del7([A1|X1], [A1|X2]):-del6(X1, X2).
del(X1, X2):-del7(X1, X2).
```

From this VACS, a target program can be generated similarly.

4. Evaluation and Discussion

In [6], a system which automatically generates Prolog programs from specification by example, is discussed. It provides a set of many clauses L and repeatedly modifies the target programs using L and the diagnosis algorithms.

We believe that for list processing programs, by providing rules (in Prolog these rules correspond to clauses) which were shown above, many suits of programs can be generated. So far our LIPS uses less resources to synthesize programs, synthesizing time is rather short. In fact, the test system of LIPS is implemented in Prolog on a 16 bits personal computer and it takes only a few seconds to synthesize one predicate. By our experiments, using some test data of specification by example, it takes 4.07 sec to generate "append" program, and 4.10 sec for "reverse" program.

In order to clarify the ability and standpoint of our system, LIPS is evaluated from four different points of view. First one is the computational complexity of the algorithm, second one is evaluated by a point of a practical application, third one is the limitation of LIPS, and the last one is the evaluation of LIPS as a knowledge acquisition system.

4.1 Computational Complexity of Algorithm

As we mentioned in chapter 2, the program synthesis process of LIPS is divided into four parts, namely, generation of DMH, transformation of DMH to CACS, transformation of CACS to VACS, and generation of target program from VACS. We derived the number of operations (processing steps) in order to evaluate the time requirements of each process. We assume that the length of specification by example is L, and the number of DMR is R.

(1) Number of operations for generation of DMH

In order to apply one rule, we must try R rules in the worst case. By applying DMR, at least two elements of specification by example are removed. So in order to finish making DMH, it takes

$$R*(L-1)+R*(L-3)+\ldots+R$$

steps. Number of operations for generation of DMH $T1(R, L)$ is

$$T1(R, L)=R*(L^2/4).$$

And the length of DMH is $L^2/4$.

(2) Number of operations for transformation of DMH to CACS

In order to make CACS, it takes the same number of steps as the length of DMH. So the number of operations for transformation of DMH to CACS $T2(R, L)$ is

$$T2(R, L)=L^2/4.$$

(3) Number of operations for transformation of CACS to VACS

In order to make VACS, it takes the same number of

steps as the length of DMH. So by the same process of (2),

$$T3(R, L)=L^2/4.$$

(4) Number of operations for generation of target program

From VACS, it takes $L^2/4$ (length of VACS) steps to find the recursive clause set, and at most $L^2/8$ (half number of VACS) of recursive clause sets are found. So number of operations for generation of target program is

$$T4(R, L)=L^2/4+L^2/8.$$

Therefore total number of operations of the algorithm is

$$\begin{aligned} T(R, L)&=T1(R, L)+T2(R, L)+T3(R, L)+T4(R, L)\\ &=(R/4+1/4+1/4+1/4+1/8)*L^2\\ &=(R/4+7/8)*L^2. \end{aligned}$$

4.2 Applicability of LIPS

LIPS can generate small size programs each behavior is indicated by one specification by example, however it cannot generate a large size program whose behavior cannot be indicated with one specification by example. Most of users want to generate are the latter type. Therefore, one good usage of LIPS is to utilize it in order to generate lower level subroutines of large size programs. As an applicability test of LIPS, we apply it to support making a line editor.

Functions of this line editor are as follows.

(1) Input text
(2) Insert one line
(3) Delete one line
(4) Insert one character
(5) Delete one character
(6) Display text

This test case editor interprets user's command and performs one of above functions. LIPS can generate lower level subroutines which actually perform functions (2), (3), (4) and (5). Main routine which calls appropriate lower level subroutines and subroutines which interpret user's commands can not be generated by LIPS, so these are hand made. Paying attention to the number of steps of test case line editor, 53% of the target program can be generated by LIPS.

4.3 Limitation of LIPS

As LIPS is a program synthesis system which utilizes the technique of inductive inference, and the target program can only manipulate lists, LIPS cannot generate the following two kinds of programs.

(1) A program whose behavior cannot be expressed by examples of list.

For example, an interactive program is expressed typically in prolog as follows.

```
program:-input(A), process(A, B), output(B).

input(A):-. . .          /*data input routine*/
process(A, B):-. . .     /*data process routine*/
process(A, B):-. . .
output(B):-. . .         /*data output routine*/
```

In the above program, "input(A):-. . ." and "output(B):-. . ." cannot be expressed by specification by examples, because it cannot express the side effect such as read from keyboard and write to display. And "program:-. . ." itself cannot be expressed by specification by examples because it doesn't manipulate list but subprocesses. 47% of predicates in 4.2 are these kinds of predicates. These arguments are, however, strongly dependent on the programming style of each programmer.

(2) A program which was expressed by specification by example which has many ways of interpretation.

In LIPS, the position of atom in a list is only distinguished by the constant symbol of atom. In the example of append program, "a" in "[a, b,c]" means first element of the list. It never means the atom "a" itself. Therefore "a" in "[a, a, b]" means first and second elements of the list. In this manner, LIPS cannot synthesize the append program from following specification by example.

```
append([a, a], [a, a], [a, a, a, a]).
```

From the above example, LIPS generates a program which is equivalent to the following program.

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

This program cannot append two lists.

4.4 Evaluation of LIPS as a Knowledge Acquisition System

Program synthesis process by means of specification by example is considered as a knowledge acquisition system, because a target program can be considered as a general form knowledge of specification by example. Generally, in order to acquire some knowledges, we must provide a lot of knowledge (we will call it "Basic Knowledge Set"), and we will find a target knowledge or a set of target knowledge from them. Considering the knowledge acquisition in order to generate a list processing program, elements of Basic Knowledge Set may be infinite. However fundamental proceses of Basic Knowledge Set may be a finite number. We consider it may be "car", "cdr", "cons" and so on in Lisp. LIPS utilizes this characteristic of the list processing, therefore LIPS provides only ten or more Basic Knowledge Set, which are fundamental combination of "car", "cdr", "cons" and so on. These are corresponding to Data Metamorphosis Rules in LIPS. Therefore it can be implemented on a small personal computer.

5. Conclusion

From the above evaluation, we can conclude that by selecting effective knowledges and some algorithms to manipulate these knowledges, and effective automatic programming system can be implemented with a rather low cost of resources.

Further development of LIPS is to make it an interactive system. As we can see in the generation of DMH, there are more than one rule that can be applied. Therefore, a searching tree can be or-parallel. This means that many kinds of programs can be generated from one specification by example. Moreover, the quality of the synthesized program is influenced by the quality of specification by example. Thus, by making LIPS an interactive system, where users can check the synthesized program and can indicate other search possibility interactively, LIPS can become a more convenient system.

References

1. ANGLUIN, D. and SMITH, C. H. Inductive Inference Theory and Methods, *Computing Surve.* **15**, 3. September (1983).
2. SUMMERS, P. D. A Methodology for LISP Program Construction from Examples, *J. ACM*, **24**, 1, January (1977).
3. HARDY, S. Synthesis of Lisp Functions from examples, *Proc. of IJCAI* (1975).
4. JOUANNAUD, J. P. and KONDRATOFF, Y. Program Synthesis from Example of Behavior, *Computer Program Synthesis Methodologies, NATO* (1983).
5. SHOW, D. E., SWARTOUT, W. R. and GREEN, C. C. Inferring Lisp Programs from Examples, *Proc. of IJCAI* (1975).
6. SHAPIRO, E. Y. Algorithmic Program Debugging, *MIT Press* (1982).

(Received January 28, 1988; revised August 10, 1988)

Appendix: Data Metamorphosis Rules (DMR)

(1) If heads of some lists in argument data are the same, remove them.

(2) If tails of some lists in argument data are the same, remove them.

(3) If (1) and (2) are both inapplicable, separate one list into the head part and the tail part.

[Notice] If rule(3) is applied repeatedly, the number of argument will increase unlimitedly. Therefore, head part of separated list are appended to the head of "work list" which is initially [ ]. By this restriction, the number of arguments can be controlled to be n or $n+1$, if the number of arguments of specification by example is n. For example, suppose specification by example is

"reverse([a, b, c], [c, b, a])",

DMH will be as follows.

([ ])	[a, b, c],	[c, b, a]
([ ])	[a|[b, c]],	[c, b, a]
[a]	[b, c],	[c, b, a]
[a]	[b|[c]],	[c, b, a]
[b, a]	[c],	[c, b, a]
[b, a]	[c|[ ]],	[c, b, a]
[c, b, a]	[ ],	[c, b, a]

...

(4) If there are some integers in the argument data, decrease it by one and apply (1)–(3), provided that the integer is not zero.

(5) Apply built-in predicate and stop to apply DMR.

(6) if (1)–(5) cannot be applied, then stop to apply DMR.