ソフトウェア基礎論 11－1
（１９８４．１２．１４）

Prolog Program Transformation of Tree Manipulation Algorithm

Hiroshi Nakagawa
(Department of Information Engineering Yokohama National University
156 Tokiwadai Hodogayaku Yokohama 240 Japan 045-335-1451ext2902)

ABSTRACT

Since Prolog programs are regarded not only declarative predicates but also procedural programs, it is a reasonable way that at first we write a declarative clear Prolog program and transform it into a not clear but efficient procedural program. In this paper we present a Prolog program transformation method especially for a binary tree manipulation program. With the notion of virtual list, we can get a procedural tree manipulation program from a declarative one. In addition we find some heuristic knowledge to write an tree manipulation algorithm from processes of Prolog program transformation. Possibly they may be useful for automatic programming.

1. Introduction

In Prolog programming , declarative style programs are easy to write and understand , but are possibly inefficient. On the other hand, a procedural style Prolog programs are efficient but complicated and hard to understand. The notion of Prolog program transformation is a programming paradigm that gives one solution for this situation. Under this paradigm, one writes a clear , declarative though possibly inefficient program, and then transforms it into a program which is more efficient although probably less clear. Some of the class of program transformations are unfolding , folding for program written in a functional language [1], and the continuation-based program transformation [5]. Program transformation for programs written in a declarative language Prolog are proposed [4]. They are all correctness-preserving transformation. Transformations of program which manipulates data structures are also proposed [1],[2],[3].[3] introduces a d-list structure for efficiency. [2] synthesizes a program including abstract data types [1]. transforms tree manipulation programs. In their system , fairly explicit intension for tree manipulation are given a priori.

In this paper , we present a different strategy for tree manipulation programs transformation :The use of virtual list notion. Although tree structures can be mapped to lists , the cognitive science tells us that human can concentrate his attention only to less than 7 plus minus 2 elements in a list at a time. So in practice we regard some elements as one abstract element (it is usually called chunk) to grasp whole list. We will call this list including abstract elements as a virtual list hereafter.

In declarative Prolog programming for tree manipulation, a tree is

transformed into a list and some tasks for example element insertion or deletion are done on the list. Then the list is transformed into a new tree. This type of tree manipulation program is easy to write. In this process, if we regard the transformed list as virtual in above described sense, the tree manipulation program using a list can be transformed into a direct tree manipulation program by program transformation. In this paper we present this class of program transformation. The given program using a list is as declarative one, and the transformed program that directly manipulates a tree structure is as procedural one. From this standpoint, our approach may reveal some kind of human mind's process of algorithm discovering.

2. Tree insertion programs via lists

A declarative tree insertion program in Prolog consists of two parts. One is a predicate that expands a tree into a list. Another is a predicate that transforms a list into a tree.In this paper we concentrate our attention to a so called search tree: every subtree of a search tree has such a property that any key values in a left subtree of it are less than the value of its root, and any key values in a right subtree of it are greater than the value of its root. When we insert an element *a into a suitable place in a given search tree t(*l, *x, *r), we usually use a well known algorithm 'binary search'. In Prolog, a predicate 'tins' that expands a tree t(*l, *x, *r) into a list and inserts *a into a suitable place of the list using the binary search algorithm is defined as

```
tins([], [], []) .                                              (1.1)
tins(*a, [], [*a]) .                                            (1.2)
tins([], t(*l, *x, *r), *y) <- tins([], *l ,*ly),tins([], *r ,*ry),
                               append(*ly, [*x|*ry], *y).       (1.3)

tins(*a, t(*l, *x, *r), *y) <- *a<*x,
                               tins(*a, *l, *ly),tins([], *r, *ry),
                               append(*ly, [*x|*ry], *y).       (1.4)
tins(*a, t(*l, *x, *r), *y) <- *a>=*x,
                               tins([], *l, *ly),tins(*a, *r, *ry),
                               append(*ly, [*x|*ry], *y).       (1.5)
```

where "*x","*l",etc. denote variables. (1.1) and (1.3) expand a tree t(*l,*x,*r) into a list *y. (1.2) is for insertion of *a. The binary search algorithm is expressed in (1.4) and (1.5). We use predicates 'tins' as basic primitives in tree manipulation hereafter. Next, we are going to define a predicate which makes up a tree from a given list. A list-to-tree predicate 'bltree' is defined as

```
bltree([], []) .                                                (2.1)
bltree(*y, t(*l, *x, *r)) <- append(*ly, [*x|*ry], *y),
                             bltree(*ly, *l),bltree(*ry, *r).   (2.2)
```

The append in (2.2) generates lists *ly and *ry, and atom *x from the given list *y. We use the predicate 'bltree' as a basic prototype to make up a tree from a list.

With predicates 'tins' and 'bltree', a tree insertion predicate 'ins' that inserts an element *a to a tree *t is defined as

```
ins(*a, *t, *ta) <- tins(*a, *t, *y),bltree(*y, *ta).           (3)
```

The variable *ta is the result tree. The predicate 'ins' is a declarative version of tree insertion program that uses lists on the intermediate stage. A procedural tree insertion program generates a result tree whose structer ia as same to the original tree old tree as possible.

But this program generates a result tree of above described after some backtrackings.

3. Transformation

In this section, a Prolog program transformation with the notion of virtual list is described along a transformation of the predicate 'ins'. In the whloe course of transformation, we mainly use unfolding,folding and sometimes introduce a new predicate. Besides these method, we introduce heuristics H1.-3. based on the notion of virtual list.

At the first step of transformation, the literal 'tins' in the body of the predicate 'ins' is unfold and the result is

```
ins([], [], *a) ← bltree([], *a).                                    (4.1)
ins(*a, [], *b) ← bltree([*a], *b).                                  (4.2)
ins([], t(*a, *b, *c) *e) ← tins([], *a, *f),tins([], *c, *g),
                    append(*f, [*b|*g], *h),bltree(*h, *e).          (4.3)
ins(*a, t(*b, *c, *d), *f) ← *a<*c ,
                    tins(*a, *b, *g),tins([], *d, *h),
                    append(*g, [*c|*h], *i),bltree(*i, *f).          (4.4)
ins(*a, t(*b, *c, *d), *f) ← *a>=*c ,
                    tins([], *b, *g),tins(*a, *d, *h),
                    append(*g, [*c|*h], *i),bltree(*i, *f).          (4.5)
```

From (4.1)and(4.2), by unfolding bltree we get termination conditions as follows

```
ins([], [], []) .                                                    (5.1)
ins(*a, [], t([], *a, [])) .                                         (5.2)
```

Here we introduce a heuristics based on the notion of virtual list as next stated.

H1. When [] is inserted into or deleted from a tree, the structure of the tree does not change.

It is intuitively valid because an abstract element which is a component of virtual list remains unchanged if we apply no operation. Hereafter we use H1. as true hypothesis. If we apply the H1. to (4.3), we get the next clause directly.

```
ins([], t(*a, *b, *c), t(*a, *b, *c)) .                              (5.3)
```

The rest is transformations of (4.4) and (4.5) into a direct tree manipulation programs.
If unfolding is applied to 'bltree' of (4.4), we get the next clause.

```
ins(*a, t(*b, *c, *d), t(*f, *g, *h)) ← *a<*c,
                tins(*a, *b, *i),tins([], *d, *j),
                append(*i, [*c|*j], *k),
                append(*l, [*g|*m], *k),
                bltree(*l, *f),bltree(*m, *h).                       (5.4)
```

The first 'append' appends *i and [*c|*j] and give the result *k. On the other hand, the second 'append' generates *l, *g and *m from *k. Usually, the first solution of *l is [] and [*g|*m] is *k, and in the next solution *l is the first elemnt of *k, etc. But there is a special solution in which *i,*g and *m correspond to *i, *l and *j respectively. Here we adopt a heuristics that the first solution of the second 'append' is this special one. So, the second heuristics is as follows.

H2. Instances bound to varibles in the second 'append' for generator correspond to the instances bound to variables in the first 'append' for append.

If we use H2. we lose the equivalence of programs. But on the other hand we get algorithm and efficiency. In (5.4), renaming of *1 ← *i, *g ← *c, and *m ← *j is applied to the whole assertion including the variables in the head. After this renaming, two 'append's are the same and have no effect in this predicate, therefore we eliminate these two 'append's.

H3. After renaming for variable's name unification in two 'append's, these 'append's are eliminated.

The result is

```
ins(*a, t(*b, *c, *d), t(*f, *c, *h)) <- *a<*c ,
              tins(*a, *b, *i),tins([], *d, *j),
              bltree(*i, *f),bltree(*j, *h).                    (5.6)
```

Since there is no side effect between variables in the first 'tins' and 'bltree' (*a,*b,*i,*f) and variables in the second 'tins' and 'bltree' (*d,*j,*h), we can interchange the second 'tins' and the first 'bltree'. Then by folding two pairs of 'tins' and 'bltree' with 'ins' we get

```
    ins(*a, t(*b, *c, *d), t(*f, *c, *h)) <- *a<*c ,
              ins(*a, *b, *f),ins([], *d, *h).                  (5.7)
```

By virtue of H1. , the second 'ins' 's *h is replaced by *d. After this replaceing, the second 'ins' is also eliminated and the final result is

```
    ins(*a, t(*b, *c, *d), t(*f, *c, *d)) <- *a<*c,ins(*a, *b, *f).  (5.8)
```

If we trace the same course of program transformation about (4.5), the result is as

```
    ins(*a, t(*b, *c, *d), t(*b, *c, *f)) <- *a>=*c,ins(*a, *d, *f).(5.9)
```

The final results program of above described transformation (5.1), (5.2), (5.3), (5.8) and (5.9) is a procedural type program of direct tree insertion. We can also get a program that effciently delete an element from a tree by the almost same transformation described above.

4. Program transformation for a tree of special property

There are many kinds of tree that satisfies a special property, for example a condition for heights or number of nodes of subtrees. We have named them as a perfect balanced tree or as AVL-balanced tree and so on. In our declarative programs we can write tree manipulation programs for these trees only by adding a test predicate 'test' to 'bltree' as follows:

```
    bltree([], [], *n) <- .
    bltree(*z, t(*l,*x,*r), *n) <- append(*ly, [*x|*ry], *z),
         bltree(*ly, *l, *m),bltree(*ry, *r, *k),test(*m , *k, *n). (6.1)
```

The third argument of 'bltree' presents a property of this tree. If we use the virtual linear list notion, our main problem is that when 'test' failes and backtracks to 'append', what kind of other solution we must pick up. It is very difficult problem because it may be done by human heuristic knowledge. The first thing we have to do for this problem is to find and list up candidates of other solutions in a combination of 'append's. The candidates we know so far are 1) one element shift and 2) associative law of 'append'. 1) is that one element of the left part *ly is shifted to the root *x and the old *x is shifted to the right part *ry and vice versa. The

one elemnt shift can be used for a perfect balanced tree. 2) is that if there are more than two 'append's, we apply an associative law to the 'append's to transform 'append's. For example, append(*a,*b,*e), append(*e,*c,*f) is transformed into append(*b,*c,*e), append(*a,*e,*f). Since it is obvious that a tree structure corresponds to a form of 'append's combination, when we pick up other 'append's form we have to transform a tree structure according to the transformed 'append's, and also have to transform the predicate 'test' to be consistent to the new 'append's. If we consider these matters, the original 'bltree' is transformed into a next program in an abstract fashion as

```
bltree(*z,f(*x)) ← append(*),append(**),bltree,bltree,test1(*x).
bltree(*z,g(*x)) ← append(*1),append(**1),bltree,bltree,test2(*x).
      :                                   :
      :                                   :                    (6.2)
```

This transformation seems to be an implicit case split technique.

H4: According to a backtrack caused by 'tset' predicate, we add assertions for other solutions that include other 'appennd's forms (for example one element shift, associative law of 'append's etc.) and tree structures as shown in (6.2)

After applying H4, we eliminate 'append's by H3 and apply unfolding and/or folding sometimes to eliminate list manipulation predicates 'tins' ('tdel') and 'bltree'. Suppose that our original program is as

```
p(*a,*b,*c) ← tins(*a,*b,*1),bltree(*1,*c).  ('tins' is replaced
```

by 'tdel' for tree deletion program)
By these transformations, from (6.2) we get a result as follows (in abstract fashion)

```
p(*a,*b,f(*z)) ← q,r(*a,*b,*z),test1(*z).
p(*a,*b,g(*z)) ← q,r(*a,*b,*z),test2(*z).
      :                   :                                      (6.3)
```

Usually r(*a,*b,*z) is a predicate for transforming the tree *b into *z, therfore very time cosuming. For efficiency we want to reduce the times of calling time consuming r(*a,*b,*z). By intrducing a new predicate s(*z,*x), the assertion (6.3) is transformed into an assertion which calls r(*a,*b,*z) only once as

```
p(*a,*b,*x) ← q,r(*a,*b,*z),s(*z,*x).
s(*z,f(*z)) ← test1(*z).
s(*z,g(*z)) ← test2(*z).
      :             :                                            (6.4)
```

H5: Assertion of the (6.3) type is transformed into assertion of the (6.4) type for efficiency.

5. AVL-balanced tree insertion

In this section we shall do a fair-sized example of a program transformation for a tree of special property : AVL-balanced tree insertion. In this transformation we introduce a new heuristics which maps an associative law of 'append' into a pointer rotation of sub-trees.

5.1 Declarative AVL-balanced tree insertion program

An AVL-balanced tree is one kind of search tree and has such a property for every sub-tree that a difference of a height of its left sub-

tree and a height of its right sub-tree is less than two. From this static property we write a program that inserts an element into a AVL-balanced tree as

brins(*a, *b, *c, *n) ← tins(*a, *b, *d),bltree(*d, *c, *n). (7.1)

where anelement *a is inserted into a tree *b and the result tree is *c, and the height of the tree *c is *n. A variable *d is a linear list made from *b. A new predicate 'bltree' is based on 'bltree' and added a condition for AVL-balanced tree as

bltree([], [], 0) ←. (7.2)

bltree(*z, t(*1, *x, *r, *n), *n) ← append(*11, [*x|*r1], *z),
bltree(*11, *1, *m),bltree(*r1, *r, *k),test(*m, *k, *n). (7.3)

where 'test' is a predicate to test wether difference between *m (the height of the left sub-tree) and *k (the height of the right sub-tree) is less than 2 , and to caluculate a height of the tree t(*1,*x,*r) as follows

test(*m, *k, *n) ← *m>*k, add(*k, *d, *m),2>*d,add(*d,*m ,*n).
test(*m, *k, *n) ← *m<*k , add(*m, *d, *k), 2>*d ,add(*d,*k, *n).
test(*m, *m, *n) ← add(*m, 1, *n). (7.4)

(7.1)-(7.4) are a declarative version of AVL-tree insertion program.

5.2 Transformation for tree rotation

By unfolding 'tins' of (7.1) and furter unfolding, we get termination conditions as

brins([], [], [], 0) ←. (7.5)

brins(*a, [], t([], *a, [], *b), *b) ← test(0, 0, *b). (7.6)

By the heuristics H1. we get an assertion for no operation as

brins([], t(*1, *x, *r, *n), t(*1, *x, *r, *n), *n) ←. (7.7)

The rest of unfolded (7.1) is as

brins(*a, t(*b, *c, *d, *e), *f, *g) ← *a<*c ,
tins(*a, *b, *h),tins([], *d, *i),append(*h, [*c|*i], *j),
bltree(*j, *f, *g). (7.8)

brins(*a, t(*b, *c, *d, *e), *f, *g) ← *a>=*c ,
tins([], *b, *h),tins(*a, *d, *i),append(*h, [*c|*i], *j),
bltree(*j, *f, *g). (7.9)

Here we will examine a transformation of (7.8) since (7.9) will be transformed by the same way of (7.8) because of a symmetry of (7.8) and (7.9). By unfolding the first 'tins' and 'bltree' of (7.8) we get a n assertion as

brins(*a, t(t(*b,*c,*d,*e),*f,*g, *h), t(*i,*j,*k,*1), *1) ←
*a<*f, *a<*c, tins(*a, *b, *m), tins([], *d, *n),
append(*m,[*c|*n], *o), tins([], *g, *p),append(*o, [*f|*p], *q),
append(*r, [*j|*s], *q), bltree(*r, *i, *t), bltree(*s, *k, *u),
test(*t, *u, *1). (7.10)

Using the virtual linear list notion namely H2 , *r,*j and *s are renamed *o,*f and *p respectively. In order to examine *o further more we unfold the first 'bltree' once more and rename variables in 'append' according to H1 , then we get as follows

brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), t(t(*i,*c,*k,*1),*f,*m,*n), *n)
← *a<*f, *a<*c, tins(*a,*b,*o), tins([],*d,*p),
append(*o,[*c|*p],*q), tins([],*g,*r),append(*q,[*f|*r],*s),
append(*q,[*f|*r],*s),append(*o,[*c|*p],*q),bltree(*o,*i,*v),
bltree(*p,*k,*w),test(*v,*w,*1),bltree(*r,*m,*y),test(*1,*y,*n).
(7.12)

Before 'append's elimination by H3. we apply H4 to (7.12) to make an other assertion for the next solution. Using an 'append's associative law we get a candidate of the next solution in a form of third and fourth 'append's as append(*o,[*c|*q],*s),append(*p,[*f|*r],*q) ---(7.13) from

append(*q,[*f|*r],*s),append(*o,[*c|*p],*q). The result of transformed tree according to (7.13) is as t(*i,*c,t(*k,*f,*m,*1),*n)---(7.14). The results are as

```
brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), t(t(*i,*c,*k,*1),*f,*m,*n),*n)
 <- *a<*f, *a<*c, tins(*a,*b,*o), tins([],*d,*p), tins([],*g,*r),
      bltree(*o,*i,*v), bltree(*p,*k,*w), bltree(*r,*m,*y),
      test(*v,*w,*1), test(*1,*y,*n).                          (7.15)
brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), t(*i,*c,t(*k,*f,*m,*1),*n), *n)
 <- *a<*f, *a<*c, tins(*a,*b,*o), tins([],*d,*p), tins([],*g,*r),
      bltree(*o,*i,*v), bltree(*p,*k,*w), bltree(*r,*m,*y),
      test(*w,*y,*1), test(*1,*v,*n).                          (7.16)
```

From a knowledge of what variables are used for input/output, 'tins's and 'bltree's are interchanged suitably, for example the first 'bltree' comes to the place between the first and second 'tins's, and we get a pattern as tins(*a,*b,*o),bltree(*o,*i,*v) etc. Now by folding these patterns with the original 'brins' (7.1) we get assertions that include neither 'tins' nor 'bltree' from (7.15) and (7.16). Using the knowledge that [] insertion dose not change a tree structer (in this program (7.7)), patterns brins([],*a,*b,*c) can be transformed into brins([],*a,*a,*c). After all these transformations we get assertions as follows

```
brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), t(t(*i,*c,*d,*1),*f,*g,*n),*n)
<- *a<*f, *a<*c, brins(*a,*b,*i,*v), brins([],*d,*d,*w),
     brins([],*g,*g,*y), test(*v,*w,*1), test(*1,*y,*n).       (7.17)
brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), t(*i,*c,t(*d,*f,*g,*1),*n),*n)
<- *a<*f, *a<*c, brins(*a,*b,*i,*v), brins([],*d,*d,*w),
     brins([],*g,*g,*y), test(*w,*y,*1), test(*v,*1,*n).       (7.18)
```

These assertions show the sigle LL-rotation algorithm. In these assertions, assertions having a pattern brins([],*x,*x,*y) are unified only once with (7.6), therefore not time consuming. In the case of *a>=*f,*a<*c, to get the right assertion we must unfold 'tins' and 'bltree' appeared inan assertion corresponding to (7.12) once more. Then we apply associative law to rotation rule H4. twice. So we get double rotation algorithm. The final results will be shown latter.

5.3 Introduce a new predicate for efficiency

In (7.17) and (7.18) a very time consuming assertion brins(*a,*b,*i,*v) is called again when 'test' failed. For efficiency we apply H5. to transform (7.17) and (7.18) into more efficient assertions and the result is as

```
brins(*a, t(t(*b,*c,*d,*e),*f,*g,*h), *x, *n) <- *a<*f, *a<*c,
     brins(*a,*b,*i,*v), brins([],*d,*d,*w), brins([],*g,*g,*y),
     sll(*i,*c,*d,*f,*g,*v,*m,*y,*n,*x).                       (7.19)
sll(*i,*c,*d,*f,*g,*v,*m,*y,*n,t(t(*i,*c,*d,*k),*f,*g,*n)) <-
     test(*v,*m,*k),test(*k,*y,*n).                            (7.20)
sll(*i,*c,*d,*f,*g,*v,*m,*y,*n,t(*i,*c,t(*d,*f,*g,*k),*n)) <-
     test(*m,*y,*k),test(*k,*v,*n).                            (7.21)
```

As to LL-rotation algorithm (7.19)-(7.21) are the final version of transformed assertions. Other rotation algorithms are also transformed along above mentioned course. For example transformed assertions of LR-rotation are as

```
brins(*a, t(t(*b,*c,t(*d,*e,*f,*g),*h),*i,*j,*k), *x,*n) <-
     *a<*i, *a>=*c, *a<*e, brins([],*b,*b,*o), brins(*a,*d,*p,*q),
     brins([],*f,*f,*r), brins([],*j,*j,*s),
     slr(*b,*c,*p,*e,*f,*g,*h,*i,*j,*k,*q,*r,*o,*s,*n,*x).     (7.22)
```

```
brins(*a, t(t(*b,*c,t(*d,*e,*f,*g),*h),*i,*j,*k), *x, *o) ←
     *a<*i, *a>=*c, *a>=*e, brins([],*b,*b,*p), brins(*a,*f,*1,*q),
     brins([],*d,*d,*r), brins([],*j,*j,*s),
     slr(*b,*c,*d,*e,*1,*g,*h,*i,*j,*k,*r,*q,*p,*s,*o,*x).     (7.23)
slr(*p1,*p2,*p3,*p4,*p5,*p6,*p7,*p8,*p9,*p0,*a,*b,*e,*f,*j,
    t(t(*p1,*p2,t(*p3,*p4,*p5,*p6),*p7),*p8,*p9,*p0)) ←
     test(*a,*b,*d), test(*e,*d,*g), test(*g,*h,*j).           (7.24)
slr(*p1,*p2,*p3,*p4,*p5,*p6,*p7,*p8,*p9,*p0,*a,*b,*e,*f,*j,
    t(t(*p1,*p2,*p3,*d),*p4,t(*p5,*p8,*p9,*h),*j)) ←
     test(*a,*e,*d), test(*b,*f,*h), test(*h,*d,*j).           (7.25)
```

5.4 The final result

We get an other termination condition assertion from (7.8) by unfolding 'tins' and 'bltree' of (7.8) some times and folding with 'brins'. The result is as

```
brins(*a, t([],*b,*c,*d), t(t([],*a,[],*e),*b,*f,*g), *g) ←
     *a<*b, test(0,0,*e),brins([],*c,*f,*i),test(*e,*i,*g).    (7.26)
```

As to (7.9) which is an insertion *a into a right sub-tree, we get assertions corresponding to (7.19)-(7.26) in the same course of transformation described above (since it is too long, the result assertions are not presented here).

6. Conclusions

We described transformations of some kinds of tree manipulation programs. Our aim is discovering human knowledge used to create a new algorithm. If we find them, a true automatic programming comes into our perspective.

REFERENCES

[1] Burstall R.M. and Darlington J. , A Transformation System for Developing Recursive Programs, J.ACM Vol. 24, No. 1, Jan. 1977,

[2] Darlington J. , The Synthesis of Implementations for Abstract Data Types, Computer Program Synthesis Methodlogies Proc. of NATO Advanced Study Institute, Oct. 1981

[3] Hansson A. and Tarnlund S.-A. , Program Transformation by Data Structure Mapping, LOGIC PROGRAMMING , Academic Press, 1982,

[4] Sato T. and Tamaki H. , Unfold/fold transformation of logic programs, Proc. of 2nd International Logic Programming Conference July 1984

[5] Wand M. , Continuation-Based Program Transformation Strategies, J.ACM Vol.27, No.1, Jan. 1980