A Note on Recursion

Kim Byung-chul

Recursion에 관한 小考

金 炳 喆

1. Introduction

Recursion refers to several related concepts in computer science and mathematics.

Such as stepwise decomposition and structured programming, the idea of recursion can enable us to see to the heart of many problems and to design algorithms that are straightforward, easy to understand, and correct.

The first language to use recursive subroutines on a regular basis were the IPL languages of Newell, Shaw, and Simon. Lists were used for the stack and the saving and restoring was done explicitly by the programmer. The first language to provide an automatic mechanism for recursion was LISP. Algol 60 and its successors, Pascal, Ada, also allow recursion, as do such other popular languages as APL, PL/I, C, and SNOBOL.

In this paper, correct perception, deep understanding, and paractical usages of recursion are attempted through the global perspective and full review. However, theoretical study, implementation method, and detail applications are neither planed nor stated.

師範大學 專任講師

2. Uses of recursion

1) recursive defintition

A form of definition will be termed ***recursive*** if free occurrences of the defined identifier on its right-hand side are to denote the value it defines (1) (Tennent, 1981).

One reason BNF is a powerful notation is that it permits *recursive defintion* of syntactical forms, allowing us to describe syntactical forms whose structures are inherently recursive.

For example, the followings are ***left*** and ***right*** recursive rules respectively.

```
<unsigned integer> = <digit>
    | <unsigned integer> <digit>
<exp-list> = <exp> | <exp>, <exp-list>
```

These are not *circular* definitions because the single ⟨digit⟩ or ⟨exp⟩ provide a *base* or starting point for the recursion. That is, one of the alternatives in each definition is not recursive. This is called a *well-formed* recursive definition.

2) recursive data structure

Some data structures such as list, tree, graph etc. are recursive structures. These types are recursively defined, usually by using pointers. For example, a binary tree is defined as follows in C language:

```
typedef struct node
       |int no;
        char *name;
        struct node *left;
        struct node *right;
       | TREE;
```

And the natural kind of algorithm that manipulates above recursive data structures turns out to *recursive algorithm.*

In LISP, where list structures are the primary data structures available, recursion is the primary control mechanism.

3) recursive program (procedure, function or subroutine)

A procedure that contains a procedure call to itself, or a procedure call to a second procedure which eventually causes the first procedure to be called is a *recursive procedure.*

When a procedure includes a call to itself we refer to this as *direct recursion.* When a procedure calls another procedure which then causes the original procedure to be invoked, we call this *indirect recursion.* (Horowitz and Sahni, 1978).

For example, the following is a recursive procedure for the problem about the Tower of Hanoi in Pascal:

```
  procedure hanoi (n: integer, sndl, indl, dndl:
char);
  begin
    if n<>0
    then begin
      |move n-1 disks from starting needle to
intermediate needle|
hanoi(n-1, sndl, dndl, indl):

      |move disk n from start to destination|
writeln (move disk, n: 1, 'from', sndl, 'to', dndl);

      |move n-1 disks from intermediate needle
to destination needle|
      hanoi(n-1, indl, sndl, dndl)
    end;
  end; |hanoi|
```

Recursion, in to the form of recursive subprogram calls, is one of the most important sequence-control structure in programming.

3. Structure and description of recursion

1) structure

Often in mathematics, an *inductive definition* is composed of three parts; basic clause, inductive clause, and external clause.

Sometimes the external clause is not stated in mathematics, however, it is always excluded in programming text.

Recursion is the name given to the technique of defining a set or a process in terms of itself. So the structure of recursion must contain two steps: stopping (or condition) step and inductive step. For example, an Algol factorial procedure:

```
integer procedure fact(n);
value n; integer n;
```

fact := *if* n=0 *then* 1 *else* n×fact(n−1);

The right side of above assignment is called a *conditional expression* (as opposed to the usual *conditional statement*). (Maclenman, 1983).

In this expression, the first part "*if* n=0 *then* 1" is a stopping step, and the last part "*else* n× fact(n−1)" is an inductive step.

Therefore we have to avoid creating an infinite regression by making sure that we eventually break out of the recursive cycle by containing the stopping step.

2) description

There are three ways to describe recursive objects.

(1) equation: For example, the Fibonacci sequence is given by the equations.

$f_0=1,\ f_1=1,\ f_{n+1}=f_n+f_{n-1}$

(2) relation: For example, a differential equation is to be solved numerically, such as

$f(x_0+nh)=F(f(x_0+(n-1)h),$
$f(x_0+(n-2)\ h),\cdots\cdots f(x_0+(n-k)h))$

(3) verbal statement: For example, a list is defined to be any finite sequence of zero or more atoms or lists.

4. Types of recursive functions

There are essentially two types of recursion, which are included in general recursive functions. An important role is played by primitive recursive functions.

1) primitive recursive function

The first type concerns *recursively defined functions* (or *primitive recursive functions*): an example of this kind is the factorial function, which is defined as

$$\mathrm{FACT}(N)=\begin{cases}1, & \text{if } N=0\\ N * \mathrm{FACT}(N-1), & \text{otherwise}\end{cases}$$

2) nonprimitive recursive function

The second type of recursin is the *recursive use of a procedure* (or *nonprimitive recursive function*). A typical example of this kind of recursion is Ackermann's function, which is defined as

$$A(M,\ N)=\begin{cases}N+1, & \text{if } M=0\\ A(M-1,\ 1), & \text{if } N=0\\ A(M-1,\ A(M,\ N-1)), & \text{otherwise}\end{cases}$$

5. Kinds of recursion

1) linear recursion

In the computation of n!, the length of the chain of deferred operations, and hence the amount of information needed to keep track of it, grows linearly with n.Such a process is called a *linear recursive recursive process*. (Abelson and Sussman, 1985).

2) tree recursion

Another common pattern of recursive computation is called tree recursion (or ***doubly recursion***). As an example, consider computing the sequence of Fibonacci numbers.

In general, the time required by a tree-recursive process will be propotional to the number of nodes in the tree, while the space required will be propotional to the maximum, **depth** of the tree. Other examples of tree recursion are copying the number of tips of a list, and so on.

3) tail recursion

A procedure is said to be *tail recursion* if the value returned is either something computed directly or the value returned by a recursive call. For example, FACT(N) is a tail recursion. (Winston and Horn, 1984).

6. Implementation and execution of recursive procedure

1) implementation

At the time of each recursive call, a new activation record is created, which is subsequent destroyed upon return. When an activation record is created, a block of storage must be allocated for it. This storage must remain allocated throughout the life time of the activation. When the activation ends, through execution of a RETURN instruction, the storage is freed for reuse. The allocation and freeing of activation records is implemented by using a *central stack*. This stack forms a *dynamic chain* composed of pairs (CIP, CEP) corresponding to (CALL, RETURN). Many computers have special instructions for handling stacks (e.g., PUSH and POP). Other machines have instructions that use a hardware stack directly for efficiency of recursive programming.

2) execution

The general algorithm model for any recursive procedure contains following steps:

(1) Save the parameters, local variables, and return address.

(2) If the base criterion has reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call). (Tramblay and Sorenson, 1984).

(3) Restore the most recently saved parameters, local variables, and return address. Go to this return address. Associated with each call to (or entry into) a recursive procedure is a *level number*. Another characteristic of recursive procedure is the *depth* of recursion, which is the number of times the procedure is called recursively in the process of evaluating a given argument or arguments. Usually, this quantity is not obvious, except in the case of extremely simple recursive functions, such as FACT(N), for which the depth N.

7. Styles of recursive functions

There are two styles for writing recursive functions.

1) down-going recursion

The down-going style keeps breadking the problem down recursively into a simpler version, until a terminal case is reached. Only then can it start to build up the answer, as intermediate results are passed back to the calling functions. For example, a down-ging version of FACT(N) is defined as follows in lambda notation:

fact= λ n. *if* m=0 *then* 1 *else* n *fact(n−1)

2) up-going recursion

In up-going recursion the intermediate results are computed at each stage of the recursion, thus building the answer up and passing it in a working space parameter, until the terminal case is reached. At this stage the answer is already complete and it has merely to be passed back to the top level calling function. We can write an up-going version of FACT(N) as:

fact= λ n. factn(n,1)

factn= λ n, w. *if* n=0 *then* w *else* factn(n−1, n * w)

8. Hints for writing recursive definitions

The stage for writing functions in the previous (6) styles are as follows. (Gray, 1984).

1) down-going recursion

(2) Write the definition for the terminal case

e.g., the number 0 or the empty list or a list of one element.

(2) For the non-terminal case assume you have a defintion that works for a case nearer the trivial case (n−1 or the remainder of a list). Use this to construct an expression for the next case up.

(3) Combine 1 and 2 with a conditional expression. Test it with examples.

2) up-going recursion

(1) Invent a function with extra workspace parameter(s), especially one to build up the result.

(2) For the terminal case set value of function =workspace parameter.

(3) For the nonterminal case recall the function with the new parameters expressed in terms of the old.

(4) Call the function with starting values for workspace parameters.

(5) Test it with examples and adjust starting values if necessary.

3) tree reucrsion (down-going)

(1) Use atom(t), and may be also null(t) to test for the terminal case.

(2) For the general case assume that your function works on both car(t) and cdr(t), and write an expression that combines them.

9. Comparison with iteration

1) Recursion, why not used?

Recursion is a powerful programming techenique which unfortunately is not employed to the extent it should. There are at least two reasons for this. One is the fact that old languages being now used widely such as FORTRAN, COBOL, BASIC etc. do not permit recrusion. Thousands of people who have learned the art of programming using them have thus been unable to experience its benifits. Two is the fact that there is often a heavy penalty in terms of execution time where one uses recursion on some compilers.

2) Recursion and iteration are theorectically equivalent

In theory, any recursive program can be converted into and iterative program by maintaining such a stack. (There is an algorithm that transforms mechanically any promotive recursive function into an equivalent iterative process, but not vice versa. And the tail recursion can be easily transformed into iteration.) This is effectively what an Algol or Pascal compiler does: it converts recursion into iteration. Although this reduction is a theoretical possibility, it is not practical to do by hand in most cases, since the resulting program is so much more compilcated. From the programer's viewpoint, recursion is more powerful than iteration. Since recursion can in principle be reduced to iteration, we might wonder if iteration is more powerful than recursion. This is also false.

3) advantages of recursion

The recursive approach to problem solving often brings us simple and elegant expression and method.

Recursion is widely used in many area such as: grammar defintion (language), recursive decent parsing(compiler), tree traversal(graph), list processing, recursive computation, and recursive simulation etc.

Recursion is very powerful technique for dealing with hierearchical structures. And recursion is becoming increasingly important in symbolic manipulation and nonnumeric applications.

4) disadvantages of recursion

Recursive programs tend to be harder to understand, as well as hard to follow in printouts, than nonrecursive ones. When something goes wrong, debugging may be very difficult. As we get buried deeper and deeper in the recursion, the tracing of the flow by means of printouts can become quite difficult. And another disadvantage of recursion inefficiency.

On each entry to a recursive program it is necessary to save information, such as partially computed results. This bookkeeping is done automatically in recursion, but on the other hand it is time-consuming.

10. Directions of research on recursion

The systematic study of recursion began in the 1920s when mathematical logic began to treat questions of definability, computability, and decidability. An important result for computer science is that the general recursive functions concide with functions defined by a Turing machine, which is a simple form of computer.

Both program and general recursion schemata, in general, give *partial functions* because the computation may terminate for some values of the arguments and for others.

The connection between current research in recursive function theory and computing practice, or even current research in computer science, is rather tenuous.

McCarthy et al. has some discussion of implimentation of recursion in LISP, and Randell and Russell discuss the implementation of recursion in Algol. Wirth discusses when to use recursion and when to use iteration. Peter has a thorough treatment of subclasses of general recursive functions. The standard reference on recursive function theory was written by Kleene(1952), who gave a more elementary treatment in a later book(1967).

Two aspects of recursion are current research topics in computer science. First, the notion of recursive program is being extended in various ways, and methods of implementing these extensions by compilers and interpreters are being studied. Second, the formal properties of recursive programs are being studied as part of the mathematical theory of computation, which has as its major object the ability to prove assertions about programs and check these assertions on a computer. (McCarthy, 1983)

Literature cited

Tennent, R. D: Principles of Programming Language; Prentice-Hall International, INC., 1981, p.131.

Horowitz Ellis, Sartaj Sahni: Fundamentals of Computer Algorithms; Computer Science Press, Inc., 1978, p.13.

Maclenman Bruce J: Principles of Programming Languages; CBS College Publishing, 1983, p.138.

Abelson Harold, Gerald Jay Sussman: Structure and Interpretation of Computer Programms; The MIT Press McGraw-Hill Book Company, 1985. pp.32-39.

Winston Patrick Henry, Berthold Klaus Paul Horn: LISP, 2nd, ed.; Addison-Wesley Publishing Company, 1984. pp.67-68.

Tremblay Jean-Paul, Paul G. Sorenson: An Introduction to Data Structure with Application, 2nd.

ed.; McGraw-Hill, Inc., 1984. pp.177-180.

Gray Peter: Logic, Algebra and Databases; Ellis Horwood Limited, 1984, pp.67-69.

McCarthy John: Encyclopedia of Computer Science and Engineering, 2nd. ed.; Van Nostrand Reinhold Company Inc., 1983, pp.1273-1275.

Goodman S. E, S. T. Hedetniemi: Introduction to Design and Analysis of Algorithms: McGraw-Hill, Inc., 1977, p.134.