数理解析研究所講究録
381巻 1980年 180-194

An Extended Iteration Statement and Its Computability

Takeo Yaku, JAPAN, Department of Mathematical Sciences,
Tokai University, Hiratsuka, Kanagawa 259 - 12.
Kokichi Futatsugi, JAPAN, Computer Science Division,
Electrotechnical Laboratory, Sakura-mura, Niihari-gun,
Ibaraki 300-31.
Akeo Adachi, JAPAN, Department of Academic and Scientific
Programs, IBM Japan, Roppongi, Minato-ku, Tokyo 106.

2

ABSTRACT

A "loop n P" statement generates a chain of the iteration module P with length n.

The "loop" statement is extended and a new control structure "substitution", implemented by "call" statement, is introduced. A "call n" statement generates a k-ary tree ($k \geq 1$ is a constant) with depth n of the substitution module. The statement generates $\sum_{i=0}^{n-1} k^i$ occurances of the substitution module without any dynamic change of the control variables during the execution.

Computability of the static programs, in which the control variables are not changed during the execution, is extended to exponential time computation from polynomial time computation.

1. INTRODUCTION

The class of programs are considered in this paper with the control variables fixed by initial input values, and not changed dynamically during the execution. The programs are said to be static. In a static program, image of computation structure can be statically found with respect to given input values before running. The static programs thus have the following properties

(1). Static program is easier to be comprehehded than non-static program [2, 6, 7].

(2). The running time of static program can be exactly evaluated before execution (cf. [1, 4]).

It is noted that "loop n P" statement generates a chain P^n of length n of loop module P. A static program with the iteration (loop statements) is executed in polynomial time, and the running time of it can be exactly evaluated before running. On the other hand the programs with exponential time complexity are not static with the iteration (loop statements). In general, these programs with exponential time complexity dynamically change control variables during computation. The changes violates the above properties (1) and (2).

We extend in Section 2 the iteration statement "loop" and introduce a new control structure "substitution." The substitution is implemented by "call" statements. A "call n statement generates a k-ary tree (k is a constant) of the substitution module with depth n. Thus the statement in static programs allows the exponential running time computation.

Consequently, the computation power of the static programs is extended to exponential time computation from polynomial time computation, conserving above properties (1) and (2). The extension is important from following reasons.

It is necessary to consider programs with exponential computation time, since we occasionally encounter this type of problems such as NP complete problems represented by known algorithms. We note that even programs with computation time bounded by a linearly exponential function $f(n) = k^n$ (k is a constant) is indeed almost intractable in practical computing. It is thus impotant to evaluate exactly the computation time of a program before running. Then we can know the tractable range of input values for the program, which is possibly almost empty.

The substitution is implemented as follows. Statements "call n" and "recall" are employed for implementation of the substitution. For example, a statement "call n do recall ; recall ; P end" is defined syntactically as call n do recall ; recall ; P end

$\rightarrow$ call n - 1 do recall ; recall ; P end ;

call n - 1 do recall ; recall ; P end ;

P (n > 1)

$\rightarrow$ P (n = 1)

Two "recall" statements are both sustituted by "call n - 1 do recall ; recall ; P end" statements. Thus the computation structure generated by the substitution statement above is a binary tree of the substitution module P with depth n. Accordingly, a statement "call n" possibly generates $\sum_{i=1}^{n-1} k^i$ occurances of the sustitution module P, where k is the number of recall statements in "substitute" scope between do and end statements. We note that an iteration "loop n do P end" is represented by the substitution as "call n do recall ; P end". A program with call - recall statements is called a "recall" program.

Followings are examples of a static recall program that computes a exponential function $f(n) = 2^n$, and its computation tree with respect to n = 2 [2].

EXAMPLE 1.

```
begin
     y← 1
     call n do
          recall ;
          recall ;
          y← y + 1 end end ;
```

page 6
Figure 1.

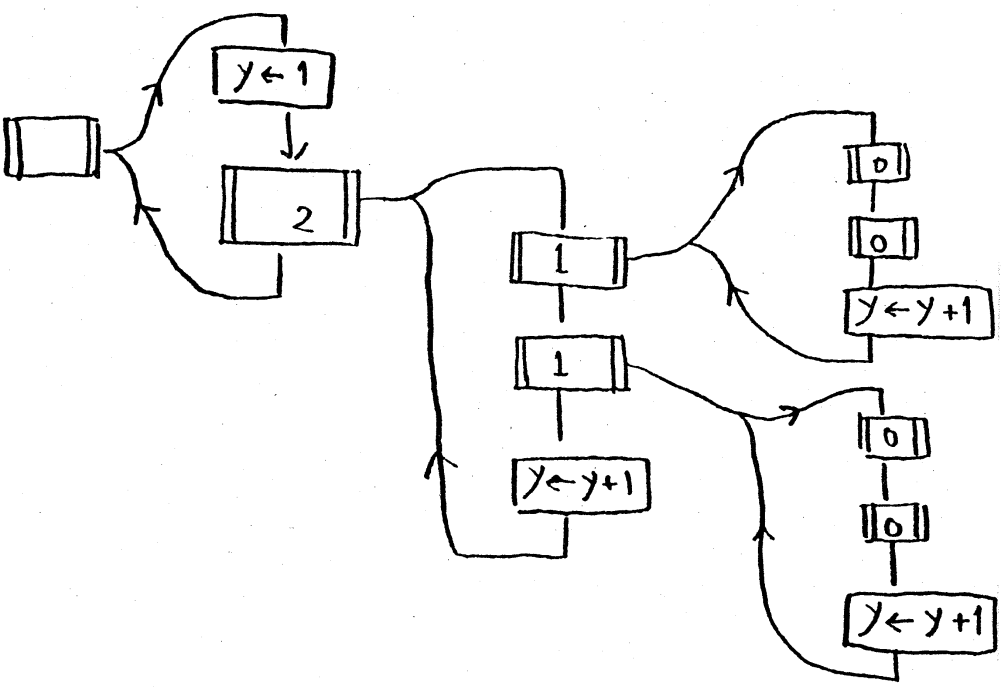

Figure 1

2. STATIC RECALL PROGRAMS

Programs with extended iteration are introduced in this section and are called recall programs.

A loop statement generate a chain of iteration modules and control the length of the chain. The computation structure is thus a chain generated by a loop statement, where a vertex corresponds to the iteration modules. We extend here the iteration to generate a tree of iteration modules from a chain. The extended iteration statement is called a call statement.

The statement generates a k-ary tree of repetition modules (k is a fixed positive integer), and control the depth of the tree. A program with call statement is called a recall program.

Only static programs are dealt in this section, in which the control variables are fixed by input values and not changed during the computation. Now we define the syntax and semantics of a static recall program syntactically as follows.

DEFINITION. Let X and S be fixed mutually disjoint countable sets of symbols. An element of X and S is called a control variable a simple variable, respectively. Let Var denote the all variables

$$\underline{\mathrm{Var}} = X \cup S$$

A static recall program is a sequence of statements over Var defined recursively as follows.

<atomic statement> :: as expected (asignment statement)

< call statement> ::= call x do <statement list>* end

*This stament list includes at least one 'recall' statement.

<statement> ::= <atomic statement> | <call statement> | recall

<statement list> ::= <statement> | <statement> ; <statement list>

<static recall program> ::= begin <statement list>** end

where u, v in S, x in X, and c is an integer. (1)

** this <statement list> does not include 'recall'.

DEFINITION. A function $c : Var \to N$ is called a memory configuration, The set of memory configurations is denoted by C. The expansion expan(P, c) of a static recall program P in a memory configuration c is a sequence of atomic statement defined syntactically as follows.

1. if P is an atomic statement s, then

   expan(s, c) = s.

2. if P is a call statement P call x Q, where

   Q = do s1 ; recall ; s2 ; recall ;
   ... ; recall ; sN end

   and $c(x) > 0$, then

   expan(P, c) = s1 ; call x - 1 Q ;
   s2 ; call x - 1 Q ;
   ... ;
   s N-1 ; call x - 1 Q ; sN

3. if P is a call statement

   P = call x Q , and $c(x) = 0$,

   then expan(P, c) = ε (null string),

   where s1, s2, ..., sN are statement lists without recall statement.

4. if P is a program begin s1 ; s2 ; ... ; sN end

   (si is a statement $1 \leq i \leq N$), then

   expan(P, c)

   = expan(s1, c) ; expan(s2 ; ... ; sN, c).

(2)

The reslut P(c) of a static recall program P for a configuration c is the configuration d such that d is obtained by application of expan(P, c) to c. A function y = f(n) is computed by a program P if and only if (i) P is over a set $\{x\}$ of control variable and a set S of simple variables, and (ii) there exists a configuration c and y in S such that c(x) = n, (s) = 0 (for any s in S) and

$$P(c)(y) = f(n) \quad \text{for any } n \text{ in } N.$$

A computation time time(P, c) of P for c is the number of all atomic statements occured in expan(P, c). The time complexity of a program P is a function $time_P : C \rightarrow N$ of the initial memory configurations C to N, where $time_P(c) = time(P, c)$. Example 1 is a static recall program that computes the function $f(n) = 2^n$.

A tree type function is defined inductively as follows :

i. Following basic functions are tree type

$$f(n) = c, \quad c \text{ is a constant}$$

$$f(n) = \sum_{i=0}^{n-1} k^i \quad \left(1 + k^1 + k^2 + \dots + k^{n-1} = \frac{k^n - 1}{k - 1} \right).$$

ii. If f(n) and g(n) are tree type then f + g and f g are tree type.

It is noted that any polynomial function and any exponential function of the form C^n are both tree type.

THEOREM 1. Let P be a static recall program over the control variable X = {x} and the simple variables S. If a memory configuration d is such that d(x) = n and d(s) = 0 (s in S), then time(P, d) is tree type over n.

Remark. For any tree type function f(n), there exists a static recall program P such that time(P, d) = f(n), where d is such that d(x) = n (x is the control variable) and d(s) = 0 (s is arbitrary simple variable).

It is noted that the running time time (P, d) of P for d can be evaluated syntactically and exactly before running from Syntaxes (1) and (2).

Following theorem shows the computation power of the static recall programs on successor function.

THEOREM 2. A function f is tree type if and only if there is a static recall program P on successor function such that P computes f.

12

4. CONCLUDING REMARKS

We have restricted the study in this paper to statics of control structure in programs during the execution. The static programs have two properties such that

(1). the control structure is simple enough to be comprehended.

(2). the execution time can be exactly evaluated before running.

On the other hand, the static programs with "loop" iteration run in polynomial time with respect to input values. It is, however, necessary to consider programs with exponential computation time.

From these considerations we extended iteration and introduced the substitution, which is implemented by "call" and "recall" statements. Programs with the substitution are called recall programs. Static recall programs are noted to conserve above properties (1) and (2). As a result computability of the static programs can be extended to linearly exponential time computation from polynomial time computation. In the programs, computation time is exactly evaluated before running.

From theoretical interest, we introduced in Section 3 a sequence "semi static recall program" of static recall programs. Then we showed that the class of all functions bounded by k - fold exponential functions is equal to the class of all functions computed by the semi static recall programs with length k. Hence the elementary functions are classified by the length of semi static programs.

In practical view several issues lie in this theory.

1. Repetition time in call statement is restrict ed to control variable itself in this paper. But we can consider that the time can expressed by expresseions of input variables. This case is not considered in this paper.

2. Explicite expression of algorithm are not provided in this paper, but we can construct it, that evaluate the exact running time.

3. Implementation techniques of call - recall statements in practical computing systems is not explicitely provided, but it is directly constructible from syntaxes (1) and (2) of the statements.

In the future of this theory, following should be considered :

4. Syntax (1) should be extended to practical use, For example "if then else" statement should be added to it.

5. Call - recall statements have another aspect, in addition to an extension of the iteration. It is very strong restriction of recursive subroutine call. If we take off the statics from programs, then recall programs can indeed compute the primitive recursive functions directly, in some sence. On the other hand if we attache

data structures to programs, then call recall statements are 14
very simple, possibly run in shorter time, implementation of recursive subroutine call in some restriction. We propose "call while < logical expression > " statement in above sense.

15

REFERENCES

(1). A. Adachi, T. Kasai and E. Moriya, A theoretical s study on the time analysis of programs, Lecture Notes in Computer Sciences 74 (1979), 201 - 207.

(2). K. Futatsugi and T. Yaku, Flow trees and their computation trees, 1979 National Convention Record of IECE Japan (1979), (6) - 123, (in Japanese).

(3). L. Kalmar, Egyszeru pelda eldonthetlen aritmetikai problemara, Mthematikai es fizika lapok 50 (1943), 1 - 23.

(4). T. Kasai and A. Adachi, A characterization of time complexity by simple loop programs, J. Comput. System Sci., to appear.

(5). A. R. Meyer and D. M. Ritchie, The complexity of loop programs, Proc. 22nd ACM National Meeting (1967), 465 - 469.

(6). T. Yaku and K. Futatsugi, Tree structured flowcharts, IECE Japan Report AL 78 - 47 (1978), 61 - 66

(7). T. Yaku and K. Futatsugi, Flowtrees and derivation trees of program texts, Proc. 20th National Convention Inform. Processing Soc. Japan (1979), 281 - 282, (in Japanese).

APPENDIX : An example of extended iteration (restricted recursion) program.

```
procedure HANOI
/* purpose */
     /* move n disks from tower A to tower C      */
/* data */
     /* A, B, C ; the names  of towers            */
     /* n       ; the number of disks  ; the      */
     /*           disks  are labeled by 1, 2,     */
     /*           ..., n from the smallest one    */
     /*           to the largest one.             */
     /* X1,X2,X3; array(0 .. n) of name
/* method */
     /*  initially, the  disks  are located at the */
     /*  tower A                                   */
     begin
        depth ← 0 ; index ← n + 1 ;
        X1(depth) ← A ; X2(depth) ← B ; X3(depth) ← C ;
        call n  do
           depth ← depth + 1 ; index ← index - 1 ;
           X1(depth) ← X1(depth - 1) ; X2(depth) ←
              X3(depth - 1) ; X3(depth) ← X2(depth - 1) ;
           recall ;
           writeln 'move disk', index, 'from',
              X1(depth), 'to', X3(depth) ;
           X1(depth) ← X2(depth - 1) ; X2(depth) ←
              X1(depth - 1) ; X3(depth) ← X3(depth - 1) ;
           recall ;
           index ← index + 1 ; depth ← depth - 1  end  end ;
```

Figure 2. The flow tree of procedure HANOI

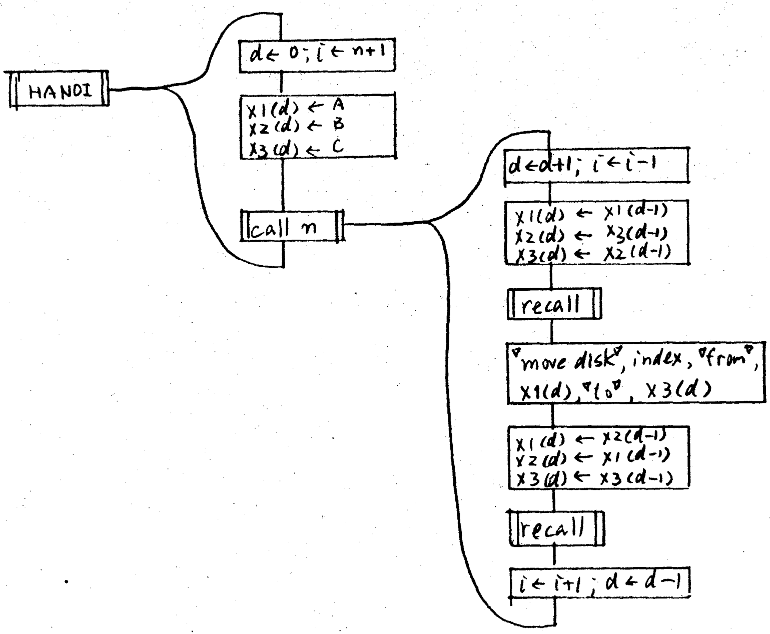

Figure 3. The computation tree of procedure HANOI for n = 3.