IMPLEMENTATIONS OF FUNCTIONAL PROGRAMMING LANGUAGES

Katsuro Inoue

Department of Information and Computer Sciences

Faculty of Engineering Science

Osaka University

February 1984

ABSTRACT

Purely functional languages have good properties such as simply defined semantics and mathematical elegance.

A purely functional language called FP System proposed by Backus has particular properties such that each function has a single argument and an object (data) which can be processed is an atom or a tree. In order to present the problems of languages such as FP System and their implementations, we have constructed two interpreters for a purely functional language FPL based on FP System. The interpreters do not rewrite expressions directly but repeatedly compute the values of the objects. One interpreter has been developed on an ACOS-900 and written in PASCAL. The objects processed by this interpreter are represented by binary lists. The other has been developed on a small microprogrammable machine HOP and written in an assembly language for its microcodes. The objects processed are represented by linear sequences in a one-dimensional array. The efficiency of these interpreters was almost satisfactory for performing small-scale computation. However, several problems of the interpreters were discovered, which are caused by the particular properties of FPL.

Next, we present a purely functional language ASL/F based on an algebraic specification language, in which there are no restrictions on the number of arguments of functions or on types of data to be processed. Methods of compiling and optimizing an ASL/F program are studied. We propose a "lazy evaluation" method which can be implemented very simply by using a single LIFO stack, and optimization techniques which are different from ordinary ones applied to procedural languages.

An optimizing compiler for ASL/F has been implemented on a MELCOM COSMO 900-II, which generates an object program in assembly language. Experimental results show that (1) all optimization techniques proposed here are useful in reducing the execution time and/or memory requirement, and (2) the execution time of an ASL/F program is about 75 to 135% of that of a PASCAL program using the same algorithm.

ACKNOWLEDGMENTS

I would like to thank Professor Tadao Kasami of the Department of Information and Computer Sciences, Osaka University. He provided me with continuous support and invaluable suggestions.

I am deeply grateful to Professor Toshio Fujisawa, Professor Koji Torii, Professor Nobuki Tokura, Professor Kensuke Takashima, Professor Junichi Toyoda, and the late Dr. Kokichi Tanaka for their invaluable and pertinent suggestions.

I wish to express my deep gratitude to Associate Professor Takuji Okamoto of Okayama University and Dr. Saburo Yamamura of Kobe University of Mercantile Marine for their important suggestions.

I express my gratitude to Associate Professor Kenichi Taniguchi, Associate Professor Hideo Miyahara, and Associate Professor Shinichi Tamura of our department for their important and pertinent guidance. I would like to express my thanks Dr. Yuji Sugiyama and Mr. Toru Tanizawa for their useful help and discussions. I am grateful to Mr. Hiroyuki Seki for his hearty help and fruitful discussions. I thank Mr. Susumu Masuda of Nippon Elec. Co. for his useful assistance in developing important programs.

TABLE OF CONTENTS

LIST OF PUBLICATIONS

[1] Inoue, K., Ito, M., Sugiyama, Y., Taniguchi, K., Tanizawa, T., and Okamato, T., " An Implementation of a Functional Programming Language on a Microprogrammable Computer", Papers of Technical group on Electronic Computers, IECE Japan, EC80-36, Sep. 1980 (in Japanese).

[2] Inoue, K., Tanizawa, T., Taniguchi, K., and Okamoto, T., "Implementations of a Functional Programming Language FPL", Trans. of IECE Japan, Vol.J65-D, No.5, May 1982 (in Japanese).

[3] Inoue, K., Seki, H., Sugiyama, Y. and Kasami, T., "Code Optimization at Compilation of Functional Programming Language ASL-F Programs", Papers of Technical Group on Electronic Computers, IECE Japan, EC82-18, June 1982 (in Japanese).

[4] Inoue, K., Seki, H., Taniguchi, K., and Kasami, T., "Functional Programming Language ASL/F and Its Optimizing Compiler", Trans. of IECE Japan (to appear).

CHAPTER 1

INTRODUCTION

Many functional programming languages have been proposed. Some of them are called purely functional languages in the sense that there are no assignment statements to variables and the order of the program statements does not affect the computation results unless explicitly specified. For example, pure LISP and FP [Backus 78] are purely functional languages.

Since the meaning of the program written in those languages is defined very clearly by using, for example, rewriting of expressions, the correctness of programs can be verified relatively easily.

A purely functional programming language called FP System has been proposed by Backus [Backus 78]. In addition to the properties of purely functional languages mentioned above, FP System has particular properties such that, each function has a single argument, and the structure of objects (data) which are the values of the arguments is an atom or a sequence representing a tree.

In order to present the problems of languages such as FP System and their implementations, we have constructed two interpreters for a purely functional language FPL based on FP System. Though several non-von Neumann computers which directly execute those programs have been

proposed [Magó 79] [Berkling 76], construction of them remains difficult.

Semantics of FPL is defined by using the concept of rewriting of expressions. Hence, we can easily interpret an FPL program in such a way that an expression to be evaluated is represented by, for example, a linked list and the expression is rewritten repeatedly according to the definition of the semantics. But this method does not seem to be efficient. Therefore, we adopt a method of interpretation by which only the objects are "computed" repeatedly.

In order to be interpreted effectively, an FPL program P is transformed to a directed graph P_M which shows correspondence between functions and their arguments. Then, the interpreter refers to P_M and computes objects at the time of execution.

One system for FPL, including the interpreter and translators for the program text and input data, has been developed using PASCAL on an ACOS-900. Objects processed by the interpreter are represented by binary lists to avoid redundant copying of objects.

We have also developed a similar system for FPL on a small microprogrammable computer HOP [Hosomi et al. 76]. The interpreter in this system has been written in an assembly language of its microcodes, and the objects to be processed are represented by linear sequences in a one-dimensional array. Since this representation does not allow sharing of the objects, copying of the objects occurs frequently.

We show the execution time and the number of consumed cells (used for representing objects), obtained by executing several quicksort programs on both systems. The efficiency of the systems was satisfactory for small-scale computations.

The following problems in the interpretation were found, which are caused by the particular properties of FPL.

(1) Since every function in FPL has only one argument, we need to use time- and space-consuming operations to separate certain elements from sequences or put elements together.

(2) Data structures of objects which can be directly computed in FPL are sequences (representing trees) only. Therefore, if other data structures and functions operating on them are essential to solve the problem, we must simulate the needed data structures and the functions by using sequences and the functions on the sequences.

In Chapter 3, we present a purely functional language ASL/F (a Functional language as a subset of Algebraic Specification Language [Nakanishi et al. 1979] [Sugiyama et al. 82b]), and study methods of compiling and optimizing ASL/F programs.

There are three types of functions in ASL/F; primitive functions (corresponding to built-in operations in commonly used programming languages such as arithmetic operations), IF functions (corresponding to conditional statements) and defined functions. A defined function g has its own definition statement in a source program:

$g(x_1, \ldots, x_n) == right_g$

where $x_1, \ldots, x_n$ are distinct variables. $right_g$ is a term and no variables except for $x_1, \ldots, x_n$ appear in $right_g$.

The semantics of ASL/F is defined clearly by using the concept of rewriting of terms. In an ASL/F program text, it is not necessary for the programmer to present rewriting rules for built-in sorts (e.g., integer or boolean) and their primitive functions (e.g., ADD or AND). In

addition to these built-in sorts, any data type and its primitive functions can be included in ASL/F and used in the program text very naturally if additional rewriting rules for them are presented by the programmer.

Here, we adopted a compiling method such that a generated object program "computes" the values of terms instead of rewriting terms directly. At the time of execution of this object program, we use only a simple LIFO stack for storage allocation.

The primary aim of the optimizations studied here is to reduce the execution time and memory requirement of the object program. Reducing such time and space is considered to be more important than reducing the size of the object program and the time for compilation and optimization. The following (a) to (d) are the optimizations discussed in this paper and implemented in the compiler we have constructed.

(a) Pre-computation of arguments of defined functions: In order to obtain the computation value of an ASL/F program without fail, we usually adopt a method called "lazy evaluation" [Henderson 80] for computing values of an argument of a defined function. However, this method is inefficient.

To increase the efficiency, we propose the following method: At compile-time, the compiler detects "needed" arguments of defined functions. The i-th argument of g is said to be needed if, in order to get the value of $g(t_1,\ldots,t_n)$, we must always compute the value of t_i regardless of the computation order and the terms $t_1,\ldots,t_n$. Hence, the values of needed arguments can be pre-computed and passed as actual parameters to a procedure without causing useless or infinite computation.

This optimization may be applied to procedural languages, but it has not been discussed. The reason for this is that, in most procedural languages, it is easy for programmers to specify the time to start the computation of a value which is passed from one procedure to another. In a functional language proposed by Keller [Keller 82], a related problem of "evaluation of demand flow" which corresponds to a simplified case of this optimization problem is presented [Tanaka and Keller 82].

Here, we formulate this optimization problem , give a general method for the optimizations, and show how effective this optimization is in reducing the execution time and memory requirement (maximum length of the run-time stack).

(b) Avoidance of duplicate computation for common subterms: In order to avoid the duplicate computation for common subterms (subexpressions) within a term in a program, we use a flag indicating whether or not the value of the common subterm has been already computed.

The object program computes the value of each term in such a way that the value of each term is defined in terms of rewriting of DAG's (Directed Acyclic Graphs) [Sugiyama et al. 82a] instead of rewriting of terms. In addition to the elimination of duplicate computation, flag tests are also eliminated, if possible.

(c) Globalization of sorts: For a sort (data type) such as an array which requires large memory space to store its value, we should allocate no memory space for the values of the sort dynamically. Instead we should only allocate a fixed memory space statically (which is sufficient to store any single value of the sort), if the source program satisfies certain conditions. If this optimization was not performed, the arrays would be copied repeatedly on the run-time stack in general,

and memory space and execution time would be exhausted. Thus, this optimization is essential.

This optimization is related to the problems of "live variable analysis" and "register allocation" applied to procedural languages [Aho and Ullman 77] , and the problem of "attribute allocation" applied to an attribute grammar [Sasaki and Katayama 83].

(d) Elimination of tail recursions: Tail recursions are transformed into iterative forms as seen in functional and procedural languages [Aho et al. 83]. This optimization reduces the memory requirement at the time of execution.

(e) Elimination of Auxiliary Functions: If in the source text of an ASL/F program, there are subterms which can be rewritten without knowing the values assigned to variables, we can then rewrite (expand) such subterms repeatedly before compilation and other optimizations.

This optimization may correspond to the expansion of open subroutines in procedural languages.

These optimization techniques can be applied not only to ASL/F but also to other functional languages. For these optimizations, only the source text of the program is analyzed. Although many ordinary optimization techniques which are used for procedural languages such as FORTRAN (e.g., register allocation, loop-invariant move, etc.) can be applied further to the procedural program generated by our compiling and optimizing method, those optimizations are not considered in this thesis.

An optimizing compiler for ASL/F has been implemented on a MELCOM COSMO 900-II, which generates an object program in assembly language. We

also present the time and space efficiency of several sample programs. Experimental results show that (1) all optimization techniques adopted here are useful in reducing the execution time and/or memory requirement, and (2) the execution time of an ASL/F program is about 75 to 135% of that of a PASCAL program using the same algorithm.

In Chapter 4, we present the conclusion and directions for further study.

CHAPTER 2

IMPLEMENTATIONS OF FUNCTIONAL PROGRAMMING LANGUAGE FPL

In this chapter, we present a purely functional programming language FPL based on FP System proposed by Backus [Backus 78], and systems for FPL on conventional machines.

In Section 2.1, the definition of FPL is given. A method of interpreting FPL programs is shown in Section 2.2 and Section 2.3. We have constructed an interpreter using this method on a relatively large machine and executed several quicksort programs. The time and space efficiency of these sample programs are presented in Section 2.4. In Section 2.5, we also present another system for FPL on a small machine. This system uses the same method of interpretation as the former system but has distinct representation of data objects.

2.1 Functional Programming Language FPL

FPL is based on FP System proposed by Backus. The details of FP are described in [Backus 78].

2.1.1 Functions, Objects, and Applications

Objects

An integer, a boolean value, or ø (a sequence of length 0) is an **atom**. An atom is an object. A **sequence** $\langle x_1, \ldots, x_n \rangle$ consisting of **elements** x_i which are themselves objects is also an object (where n is

called the **length** of the sequence, and $n \geq 0$). For example, <5, <9, 25>,73> is a sequence of length 3.

Functions

There are three types of functions in FPL.

1. Built-in **primitive functions** (see Table 2.1).
2. **UD-functions** (User-Defined function) defined by the programmer in the program text.
3. Functions defined by built-in **functional forms** (see Table 2.2).

Each function has an object as its argument and the (resulting) value of the function is also an object.

A primitive function ID is the identity function. For a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ as an argument, the values of primitive functions, (a)HEAD, (b)SELi, (c)TAIL, and (d)LEN, are objects (a)x_1, (b)x_i, (c)$\langle x_2, \ldots, x_n \rangle$, and (d)n, respectively. LEN1 is a predicate whether or not the argument is a sequence of length 1, and LT compares two integers. PHI generates ϕ (a sequence of length 0) and CONSTi generates integer i for any argument. NULL is a predicate whether or not the argument is ϕ.

The syntax of a definition statement of a UD-function is as follows.

"DEF" the name of UD-function "=" the definition body

The name of a UD-function on the left-hand side is a string of alphanumeric characters (maximum of 8 characters) with a leading alphabetic character and is not a the key words (e.g., DEF, END, ADD, ...). The definition body on the right-hand side is a description of a function which is usually a function defined by functional forms.

Table 2.1 Primitive Functions of FPL

```
1.ID      2.HEAD    3.TAIL    4.LEN     5.LEN1
6.AND     7.OR      8.NOT     9.ADD     10.SUB
11.MULT   12.DIV    13.MOD    14.LT     15.EQ
16.SELi   17.CONSTi           18.NULL
19.PHI (PHI:x = ∅)
20.UNION (UNION:x =  x is <y,∅>  ->  <y> ;
                     x is <∅,z>  ->  <z> ;
     x is <y,<z1,...,zn>> -> <y,z1,...,zn> ;
                     else undefined        )
21.IF (IF:<x,y,z> =  z is True  ->  y ;
                     z is False ->  x ;
     z is a sequence or an atom other than
                 True,False -> undefined ;
                     IF:<x,y,z>            )
```

Inside the brackets of 20 and 21 McCarthy's expression is used.

Table 2.2 Functional Forms of FPL

```
1. CONSTR(Construction)  [f1,f2, ... ,fn]
    [f1,f2, ... ,fn] :x = <f1:x,f2:x, ... ,fn:x>
2. COND(Condition)  (f1 ; f2 <- f3)
    (f1 ; f2 <- f3):x = IF:<f1:x,f2:x,f3:x>
3. COMP(Composition)  f1.f2. ... .fn
    f1.f2. ... .fn:x = f1: f2: ... : fn:x
4. INS(Insert)  /f\
    /f\:x = x is <x1> -> x1  ;
            x is <x1,x2, ... ,xn> ->
                        /f\:<f:<x1,x2>,x3, ... ,xn>  ;
            else undefined
5. APPL(Apply to all)  'f"
    'f":x = x is ∅ -> ∅  ;
            x is <x1,x2, ... ,xn> ->
                        <f:x1,f:x2, ... ,f:xn>  ;
            else undefined
```

A functional form has some functions as its functional arguments and it defines a function. CONSTR gives a sequence consisting of the values of functions which are the functional arguments of CONSTR. COND is defined by a primitive function IF. The value of IF is the value of the second element of its argument when the third element is a boolean value True, and that is the value of the first element when the third element is False.* In FPL, programmers cannot define any functional form.

Applications

An **application** denoted by ":" is the only operation in FPL, and it generates an expression by giving an object to the function as an argument. We call an object an **expression.** f:E is also called an expression where f is any function (including a function defined by functional forms) and E is an expression. f:E denotes that a function f is applied to an object E. If this expression can be rewritten into an

---

* Assume that we rewrite an expression IF:A. If A is not a sequence of length 3 or if the third element of A is either a sequence or an atom other than True or False, the value of IF is not defined. If the third element of A is an expression E including application symbols ":", the resulting expression is still IF:A (the third element must be evaluated first.) These definitions of COND and IF seem to be complicated. But by the definition of the conditional expression in Backus' FP System, the rewritings for the value of a predicate must be executed on an equation other than the equation for the value of the program. By using those definitions for COND and IF, however, all rewritings can be executed on a single equation.

expression E′ by using the definitions of primitive functions, UD-funtions, and functional forms, we write f:E == E′.

2.1.2 FPL Program

Structure of Program

An FPL program P as a whole defines a UD-function f_P to be evaluated. At the top of the program text, the name of f_P is written, which for input data D generates the value of P.

Figure 2.1 shows an FPL program which computes the greatest common divisor of two integers by the Euclid's algorithm. In this program, f_P is GCD. EQ0 is a predicate whether the argument is 0 or not. The outline of the definition of GCD is as follows.

```
If      (x mod y) = 0
     then      GCD(x, y) = y
     else      GCD(x, y) = GCD(y, (x mod y))
```

Semantics

The value of an expression is an object obtained by rewriting the expressions repeatedly according to the definition of the functions or functional forms until no application symbols ":" appear in the expression.

Data D for a program P is an object and is passed to P at the time of execution. The value of expression f_P:D is called the value of P for D.

Assume that an expression E has more than two subexpressions which can be rewritten. The value of E does not depend on the order of the applications among those subexpressions. Suppose we evaluate an

expression E where E is IF:<f_1:x, f_2:x, f_3:x>. Rewriting for the value of f_1:x or f_2:x before rewriting for the value of f_3:x may cause useless or infinite rewritings even if E has a value. Then, we rewrite subexpression f_3:x first. Figure 2.2 shows an example of rewriting steps where <6,4> is passed to the program of Figure 2.1 as input data.

```
        GCD
  DEF EQ0 =EQ.[CONST0 , ID]
  DEF GCD =(GCD.[SEL2 , MOD] ; SEL2 <- EQ0.MOD)
        END
```

Figure 2.1 An FPL Program which Computes the Greatest Common Divisor

```
   GCD:<6,4>
== (GCD.[SEL2,MOD];SEL2 <- EQ0.MOD):<6,4>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,EQ0.MOD:<6,4>>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,EQ0:2>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,EQ.[CONST0,ID]:2>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,EQ:<CONST0:2,ID:2>>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,EQ:<0,2>>
== IF:<GCD.[SEL2,MOD]:<6,4>,SEL2:<6,4>,False>
== GCD.[SEL2,MOD]:<6,4>
== GCD:<4,2>
== (GCD.[SEL2,MOD];SEL2 <- EQ0.MOD):<4,2>
== IF:<GCD.[SEL2,MOD]:<4,2>,SEL2:<4,2>,EQ0.MOD:<4,2>>
 .....
== IF:<GCD.[SEL2,MOD]:<4,2>,SEL2:<4,2>,True>
== SEL2:<4,2>
== 2
```

Figure 2.2 Rewriting Steps for GCD: <6, 4>

2.2. Methods of Interpretation

We compared the following methods, (1) and (2), of interpreting an FPL program P for input data D.

(1) We represent the expression to be evaluated by a linear sequence in, for example, a one-dimensional array. Then, the represented expression is scanned repeatedly for subexpressions which can be rewritten immediately and the found subexpressions are rewritten in the array, until no application symbols ":" appear in the array. It is easy to implement this method, but repeated scanning of the array may cause inefficiency.

(2) We first store P and D in distinct areas. Then the objects including D are modified repeatedly according to the definition of functions or functional forms in P until the value of P is obtained. In this method, we have to save many temporary objects and states of intermediate steps for later computation. But this can be achieved easily by using subroutine calls of high-level languages (e.g., PASCAL, PL/I, ...). Therefore, we adopted this method.

The order of functions or functional forms to be applied may be determined partially before execution (interpretation), but for ease of implementation we adopted a method where the order of applications is determined by the interpreter (the interpreter is denoted by I) at the time of execution. For ease of referring to P by I, we transfer P into a directed graph P_M before interpretation.

Figure 2.3 shows a directed graph P_M representing P of Figure 2.1. Each vertex in P_M corresponds to an occurrence of a function or a

functional form in P and has a label of the name of the function or functional form. Each edge connects f and g as f -> g, where f is a vertex whose label is a functional form F and g is a vertex whose label is a function G (G may be a function defined by functional forms) which is one of the functional arguments of F. There are no vertices corresponding to the occurrences of UD-funtions in P, since a vertex whose label is the functional form having a functional argument of a UD-function H is connected directly with the vertex which corresponds to the occurrence of the outermost functional form (or maybe a function) on the right-hand side of the definition of H.

For example, the vertex labeled COMP at the top left of Figure 2.3 is connected with the vertex labeled COND which corresponds to the outermost functional form on the right-hand side of UD-function GCD. The number of sons of a vertex corresponding to a functional form F is the same as the number of functional arguments of F.

Here we call son u of a vertex v the "i-th son of v", if u corresponds to the i-th functional argument of the functional form which corresponds to v. For example, for the vertex labeled COND at the top of Figure 2.3, the vertex labeled COMP at the top left is the first son, the vertex labeled SEL is the second son, and the vertex labeled COMP at the top right is the third son.

P_M can be easily represented by a binary list as shown in Figure 2.4.* To convert P to this binary list representing P_M, we have designed

---

* In Figure 2.4, each cell corresponding to a functional form contains the name of the functional form in the left portion and a pointer in the right portion of the cell.

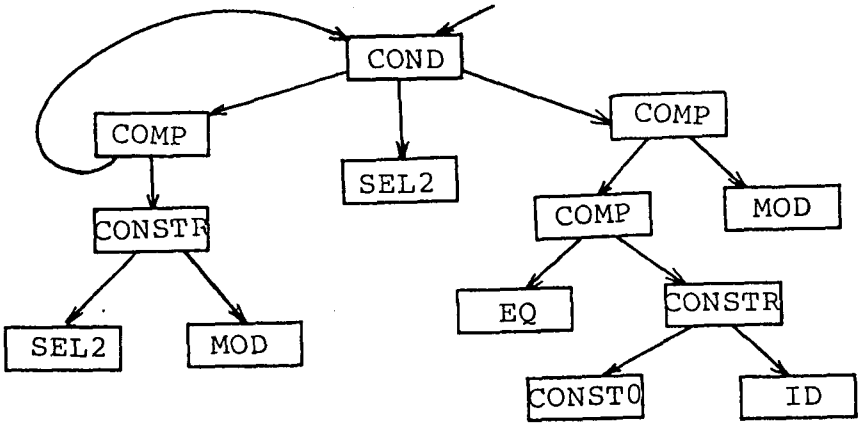


Figure 2.3 An Example of Directed Graph P_M

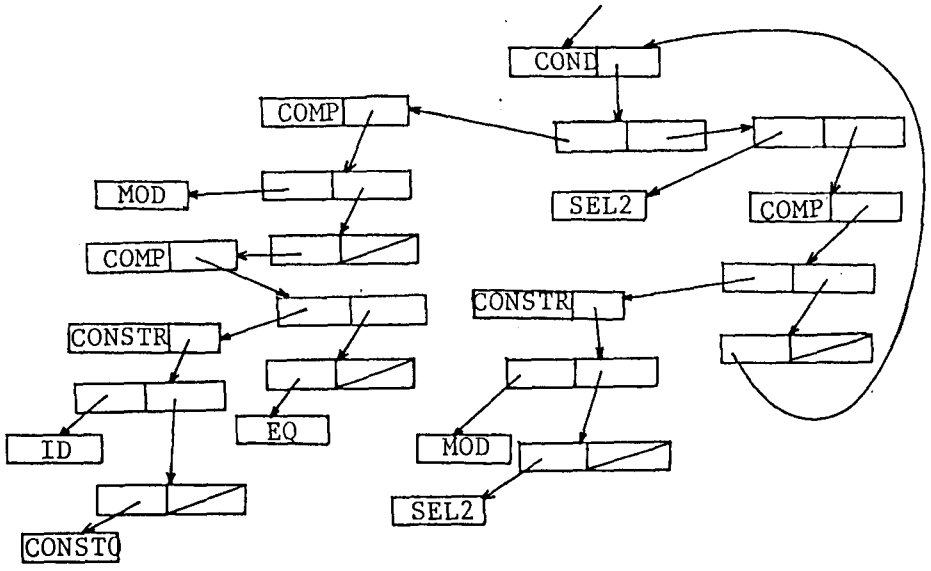


Figure 2.4 Representation of P_M by List

a translator T_1.

Objects (including input data D) can be easily represented by trees. Hence, they are also represented by binary lists as in Figure 2.5. Each cell corresponding to an atom in a list which represents an object has an integer value (-2^{35} - $+2^{35}-1$) or a boolean value. Input data D is transformed into this binary list by a translator T_2 before interpretation.

Then, the entire system for FPL consists of the translators T_1, T_2, and interpreter I. T_1, T_2, and I are written in PASCAL, a language with which we can write recursive procedures and list operations relatively easily. This system runs under TSS on an ACOS-900 (where a user can use a memory space having a maximum of 112K words [one word is 36 bits]). A cell containing an atom occupies 2 words and a cell containing two pointers occupies 3 words. The memory space having a maximum of 60K words (called the heap area) is reserved for these cells.

Together, T_1, T_2, and I are about 2000 lines long and it took 3 man-months to design and implement them.

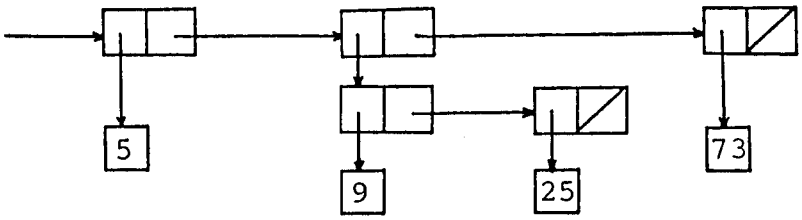


Figure 2.5 Representation of Object <5,<9,25>,73>

2.3 Interpreter

Figure 2.6 shows the outline of interpreter I. Here, we assume that a variable of type "obj" contains an object (even if it is a sequence), a variable of type "node" contains a vertex in P_M. The following, (1) to (5), describe interpreter I shown in Figure 2.6.

(1) Procedure "interpret" has blocks corresponding to the primitive functions and functional forms in FPL. One block is selected by the label of the vertex in variable "pc" (which is the first argument of procedure "interpret") and executed. The resulting object of "interpret" is computed for the second argument "data" in the selected block and stored in the third argument "return". The main program activates "interpret" with the vertex corresponding to f_p (the function to be evaluated) and input data D as an actual argument, and prints out the resulting object of the activated "interpret".

(2) In each block corresponding to a functional form F, we must obtain the values of several functions which are the functional arguments of F. Therefore, in each block corresponding to a functional form, "interpret" is activated recursively with the label of a vertex which is a son of a vertex in "pc". By using procedure activations of a high-level language, temporary objects (e.g., the content of "return" in "CONSTR" block) and intermediate states are saved and restored automatically.

(3) An object is represented by a binary list as mentioned above. Thus, a variable of type "obj" is implemented by a pointer pointing to a cell in a list. We make a cell on a list representing an object X pointed by the other pointer r so that r represents other object Y which is a subsequence of X. This reduces the number of cell copying operations. But the list pointed by the pointer in "data" of one instance of

```
PROGRAM main ;
  VAR work: obj ;


PROCEDURE interpret(pc: node,  data: obj,  VAR return:obj) ;
 VAR  i: INTEGER ;
      work1,work2: obj ;

 CASE sort(pc) OF                    — ①
   'ID'    :return:=data                    ;
   'HEAD'  :return:=select(1,data)          ;   —②
                                                          ③
   'PHI'   :return:=null                    ;   —④
   'CONSTR':BEGIN
              return:=null ;
              FOR i:=numson(pc) DOWNTO 1     — ⑤
              BEGIN
                interpret(son(pc,i), data, work1) ;     — ⑥
                return:=union(work1,return) ;           — ⑦
              END ;
            END ;
   'COND'  :BEGIN
              interpret(son(pc,3), data, work1) ;
              IF work1=TRUE THEN interpret(son(pc,2), data, return)
                            ELSE interpret(son(pc,1), data, return) ;
            END  ;
   'COMP'  :BEGIN
              work1:=data ;
              FOR i:=numson(pc) DOWNTO 1
              BEGIN
                interpret(son(pc,i), work1, work2) ;
                work1:=work2 ;
              END ;
              return:=work2 ;
            END ;
```

```
'INS'   :BEGIN
           work1:=select(1,data) ;
           FOR i:=2 TO length(data)                    —⑧
           BEGIN
             interpret(son(pc,1), union(work1,select(i,data)), work2) ;
             work1:=work2 ;
           END ;
           return:=work2 ;
         END ;
'APPL'  :BEGIN
           return:=null ;
           FOR i:=length(data) DOWNTO 1
           BEGIN
             interpret(son(pc,1), select(i,data), work1) ;
             return:=union(work1,return) ;
           END ;
         END ;
END ;


BEGIN   /* OF MAIN */
   interpret(fp, D, work) ;
   WRITE(work)  ;
END.
```

① Selection of the following blocks depending on the label contained in "pc" .

② "select(i,data)" is the i-th element of the sequence contained in "data".

③ apply each operation according to the definition of the primitive function.

④ "null" is ø (a sequence of length 0).

⑤ "numson(pc)" is the number of sons of the vertex whose label is contained in "pc".

⑥ "son(pc)" is the i-th son of the vertex whose label is contained in "pc".

⑦ "union(work1,return) is sequence $\langle f_i(x), f_{i+1}(x), \ldots, f_n(x) \rangle$ where "return" contains $\langle f_{i+1}(x), \ldots, f_n(x) \rangle$ and "work1" contains $f_i(x)$.

⑧ "length(data)" is the length of the sequence contained in "data".

Figure 2.6 Outline of Interpreter

"interpret" cannot be released and reclaimed after the completion of the instance, since parts of the list may be shared. For example, as shown in Figure 2.7, after the completion of one instance of "UNION" block which has data <5, <9, 25>> in variable "data", cell 1 and 2 cannot be released because the other pointers pointing them may exist.*

(4) When no more new cells can be obtained in the heap area, a garbage collection routine GC in I begins to reclaim unused cells. GC first traverses only used cells and marks them, and then traverses all cells in the heap area and reclaims unmarked cells. GC is also written in PASCAL, so that it takes a relatively long time to execute (for example, in the quicksort program which will be described later, the time for GC is about the same as the time for sorting). If it were possible to write the garbage collection routine in a lower level language, its execution time would be reduced greatly.

(5) When one of the following errors (a), (b), and (c) occurs, interpreter I prints out the content of "data" and the name of the function or the functional form corresponding to the block in which the error has occurred, and halts.

(a) The resulting value of the block is not defined for the object in "data".

(b) An overflow occurs.

(c) No cells can be reclaimed by GC (i.e., the heap area is not sufficient).

---

* It may be possible to check whether or not the cells can be released and reclaimed at the time of completion of each instance. But we are not aware of any simple and efficient method.

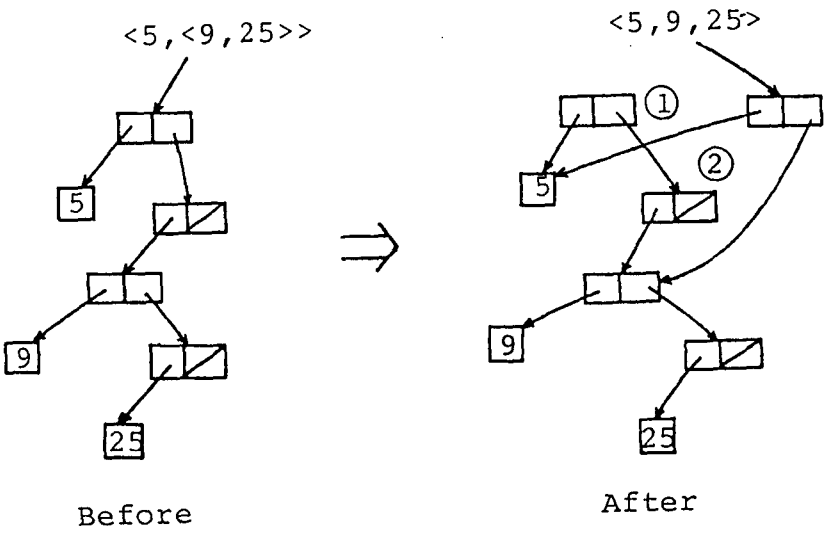


Figure 2.7 Execution of UNION:<5, <9, 25>>

2.4 Sample Programs

Quicksort 1

Figure 2.8 shows a quicksort program in FPL, which sorts a given integer sequence in nonincreasing order. This program was written in such a way that it can be considered to be a natural implementation of the quicksort algorithm in FPL; therefore no special attention on the execution time and memory usage was paid.

A UD-function QSORT1 takes an integer sequence x as its argument and sorts the sequence as follows:

(a) First, apply a UD-function SPLIT1 to an integer sequence x and obtain a sequence consisting of three elements denoted by <1st-elem, 2nd-elem, 3rd-elem>, where 1st-elem is BIGGER:x, 2nd-elem is HEAD:x, and 3rd-elem is SMALLER:x. 1st-elem is a sequence of the integers which appear in a sequence TAIL:x (a sequence obtained by removing the first element of x) and which are greater than an integer HEAD:x (i.e., the first element of x). 3rd-elem is a sequence of the integers which appear in TAIL:x and which are not greater than HEAD:x.

(b) Next, apply QSORT1 to both sequence 1st-elem and 3rd-elem obtained by (a).

(c) Finally, make the value of QSORT1:x by concatenating the sequences obtained by (b) and HEAD:x. When x is ø, the resulting object is also ø.

The value of BIGGER:x is obtained as follows.

First we construct a sequence of three elements denoted by <1st-elem, 2nd-elem, 3rd-elem>, where 1st-elem is ø, 2nd-elem is HEAD:x, and 3rd-

```
     QSORT1
DEF QSORT1=(FORM.[QSORT1.HEAD , SEL2 , QSORT1.SEL3].SPLIT1 ; PHI <- NULL)
DEF SPLIT1 =([ BIGGER , HEAD , SMALLER ] ; [ PHI , HEAD , PHI ] <- LEN1)
DEF BIGGER=BLOOP.[ PHI , HEAD , TAIL ]
DEF BLOOP=((BLOOP.[HEAD,SEL2,TAIL.SEL3]
           ; BLOOP.[UNION.[HEAD.SEL3,HEAD],SEL2,TAIL.SEL3]
          <- LT.[SEL2 , HEAD.SEL3]) ;HEAD <- NULL.SEL3  )
DEF SMALLER=SLOOP.[PHI , HEAD , TAIL]
DEF SLOOP=((SLOOP.[UNION.[HEAD.SEL3,HEAD],SEL2,TAIL.SEL3]
           ; SLOOP.[HEAD,SEL2,TAIL.SEL3]
             <- LT.[SEL2,HEAD.SEL3]) ; HEAD <- NULL.SEL3 )
DEF FORM =APPEND.[HEAD , UNION.[SEL2 , SEL3]]
DEF APPEND=(UNION.[HEAD.HEAD , APPEND.[TAIL.HEAD , SEL2]] ; SEL2<-NULL.HEAD)
   END
```

Figure 2.8 Quicksort 1

Table 2.3 Execution of Quicksort 1

The number of integers to be sorted	50	100	200	300	400	500
Execution time (sec.)	1.4	3.0	7.7	13.3	19.1	24.9
The number of consumed cells (x 1000)	4.9	10.6	26.7	47.2	67.4	88.5

elem is TAIL:x initially. Then, each integer in 3rd-elem is picked up one by one, and compared with 2nd-elem repeatedly. If it is greater than 2nd-elem, it is added to 1st-elem. The value of BIGGER:x is 1st-elem at the time when 3rd-elem becomes ϕ.

On the other hand, SMALLER:x is obtained by the similar operations but the integers to be added to 1st-elem are not greater than 2nd-elem.

APPEND connects two sequences.

In Table 2.3, we show the execution time (except for the time for T_1, T_2, and GC) and the number of consumed cells (cells reclaimed by GC are counted every time they are reused) of Quicksort 1. It took 0.7 second to transform the source text of Quicksort 1 by T_1, and 2 seconds to transform a sequence of 500 integers by T_2.

Quicksort 2

In Quicksort 1, in order to obtain the sequences of greater integers and not greater integers than HEAD:x, we used two UD-functions BIGGER and SMALLER. But if both sequences, BIGGER:x and SMALLER:x, can be obtained by a UD-function at the same time, the efficiency will be increased. Thus, we defined SPLIT2 and LOOP in Quicksort 2 as shown in Figure 2.9, instead of SPLIT1, BIGGER, BLOOP, SMALLER, and SLOOP in quicksort 1. The argument of LOOP is a sequence of four elements denoted by <1st-elem, 2nd-elem, 3rd-elem, 4th-elem>. 1st-elem is a sequence whose elements are greater than 2nd-elem, 3rd-elem is a sequence whose elements are not greater than 2nd-elem, 4th-elem is a sequence whose elements have not yet been compared with 2nd-elem. The elements in 4th-elem are picked up one by one, compared with 2nd-elem, and added to 1st-elem or 3rd-elem according to the results of the

```
        QSORT2
DEF QSORT2=(FORM.[QSORT2.HEAD , SEL2 , QSORT2.SEL3].SPLIT2 ; PHI <- NULL)
DEF SPLIT2 =(LOOP.[PHI , HEAD , PHI , TAIL] ; [PHI , HEAD , PHI] <- LEN1)
DEF LOOP=((LOOP.[HEAD, SEL2 , UNION.[HEAD.SEL4 , SEL3] , TAIL.SEL4] ;
           LOOP.[UNION.[HEAD.SEL4 , HEAD] , SEL 2 , SEL3 , TAIL.SEL4]
           <- LT.[SEL2 , HEAD.SEL4]) ; [HEAD , SEL2 , SEL3] <- NULL.SEL4)
DEF FORM =APPEND.[HEAD , UNION.[SEL2 , SEL3]]
DEF APPEND=(UNION.[HEAD.HEAD , APPEND.[TAIL.HEAD , SEL2]] ; SEL2<-NULL.HEAD)
        END
```

Figure 2.9 Quicksort 2

Table 2.4 Execution of Quicksort 2

The number of integers to be sorted	50	100	200	300	400	500
Execution time (sec.)	1.0	2.1	5.3	8.9	13.0	16.8
The number of consumed cells (x 1000)	3.7	8.1	20.1	35.1	50.6	66.4

comparisons. The resulting object of LOOP is a sequence consisting of 1st-elem, 2nd-elem, and 3rd-elem at the time when 4th-elem becomes ø.

The execution time and the number of cells consumed by Quicksort 2 were reduced by 30% and 25%, respectively (see Table 2.4).

Quicksort 3 and 4

A UD-function APPEND in Quicksort 1 and 2 is defined recursively. For the number (denoted by n) of integers to sort, APPEND is executed about (1/2)n log n times, and uses (5/2)n log n cells as a result. (In most cases for each execution of APPEND, two cells for the outside CONSTR on the definition of APPEND, another two cells for the inside CONSTR, and one cell for the UNION, a total of five cells are usually consumed.)

Thus, we added a block for APPEND to interpreter I so that APPEND can be executed as a primitive function. As a result about (1/2)n log n cells are consumed for APPEND, and compared with Quicksort 2 the execution time was reduced by 20% and the number of cells consumed was reduced by 15%. (Quicksort 3 is defined by removing the definition of APPEND from Quicksort 2.)

For each execution of a function LT.[SEL2, HEAD.SEL4] on the definition of LOOP, two cells are consumed, and this function is executed n log n times. Thus, we also added a block for this function to the interpreter as a primitive function denoted by LT24. As a result both the execution time and the number of cells consumed were reduced by 20% compared with Quicksort 3. (The quicksort program using the primitive function LT24 is called Quicksort 4.) The execution time and the number of consumed cells of Quicksort 4 are shown in Table 2.5.

Table 2.5 Execution of Quicksort 4

The number of integers to be sorted	50	100	200	300	400	500
Execution time (sec.)	0.7	1.5	3.5	6.2	8.5	11.4
The number of consumed cells (x 1000)	2.8	5.9	14.7	25.8	36.1	47.3

2.5 An Implementation on a Small Machine

Using the methods mentioned in the preceding sections, we have also implemented a similar system for FPL on a small machine in a low level language.

2.5.1 Outline of the System

(1) We have developed the system for FPL on a microprogrammable machine HOP [Hosomi et al. 76]. HOP has a main memory of 32K bytes and a control memory of 3K words (one word is 30 bits). It executes one microinstruction in 350 ns. The tools for developing programs, e.g., a self-assembler, a disassembler, an editor, a debugger and an emulator for the 8080 microprocessor had been developed on it.

(2) The system consists of translators T_1',T_2' and an interpreter I' corresponding to T_1, T_2 and I mentioned in Section 2.2, respectively. T_1' and T_2' were written in the 8080 assembly language and together are about 1500 steps long. The interpreter I' was written in an assembly language for HOP's microcodes and is about 1000 steps long. It took about 3 man-months to complete them.

(3) In this system, objects are represented by linear sequences in a one-dimensional array (as shown in Figure 2.10) rather than binary lists, since operations for the sequences are considered to be relatively easy by using HOP's instructions. The array has a maximum length of 24K bytes. A cell containing an integer occupies 4 bytes and a cell containing a symbol occupies 1 byte.

(4) The structure of I' is similar to that of I shown in Figure 2.6. In I', variables of type "obj" are implemented with pointers pointing to ends of the linear sequences representing objects in the array. One

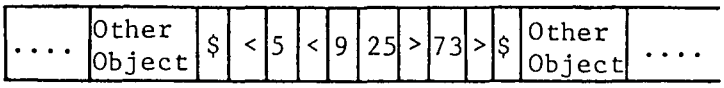


Figure 2.10 Representation of Object <5,<9,25>,73> in I'

Table 2.6 Execution of Quicksort 3 using I'

The number of integers to be sorted	50	100	200	300	400	500
Execution time (sec.)	1.0	6.2	23.6	59.0	90.9	152.8
The number of consumed cells (x 1000)	0.6	1.0	2.0	3.8	4.6	7.1

linear sequence in the array is pointed by exactly one pointer, and no subsequences are pointed by other pointers. In I′, the instructions for saving and restoring the local variables at the time of procedure activations and returns from procedures are written explicitly, since I′ was written in a low level language (microcodes).

2.5.2 Execution Results

Quicksort 2, shown in Figure 2.9, can not be executed by I′ when the number of integers to be sorted exceeds 100, since there is not enough space in the array. This shortage occurs because at the time of execution of the UD-function APPEND, the objects created by repeated execution of TAIL are stored in the array successively. (For the number (denoted by n) of integers to sort n, $O(n^2)$ cells are consumed each time APPEND is executed.)

Then, we added the block for computing APPEND as a primitive function to I′ so that the number of consumed cells is O(n) each time APPEND is executed. The execution time and the number of cells of Quicksort 3 by I′ are shown in Table 2.6.

The time required for of Quicksort 3 is O(n log n) by I. On the other hand, it is $O(n^2)$ by I′. The reason for the difference is that the time required for executing the primitive functions such as TAIL is O(n) by I′, while it is a constant by I.

Unused objects (e.g., an object pointed by a pointer in "data" at the time of completion of procedure "interpret") can be removed from the array instantly. The maximum size of the space for the cells in the array (i.e., the maximum length of all objects together) is approximately proportional to n when Quicksort 3 is executed by I′.

2.6 Discussion

(1) No difficulty occurred when both systems were developed, and the efficiency of the systems is satisfactory for small-scale computation*. The implementation methods discussed here are considered effective for executing programs of functional languages such as FPL.

(2) The following (<a>, <b>, and <c>) are the disadvantages of FPL found during the development of some FPL programs.

<a> The number of primitive functions and functional forms in the current version of FPL is insufficient, since we sometimes had difficulty in writing straightforward programs.

<b> Since every function in FPL takes only one argument, we often need to use primitive functions which separate certain elements from sequences (e.g., SELi, HEAD, TAIL, etc.) and primitive functions or functional forms which put elements together (e.g., UNION, CONSTR, etc.) as seen in the definition of LOOP of Quicksort 2, for example. Those primitive functions and functional forms would be unnecessary if functions could take more than one argument.

<c> Data structures whose values can be directly processed in FPL are restricted to sequence only. Therefore, if other data structures and functions on them are essential to solve the problem, we must simulate the needed data structures and the functions by using sequences and the functions on the sequences.

(3) By the interpreter I, one memory cell is consumed each time UNION is

---

* A LISP program corresponding to Quicksort 2 was executed on the ACOS-900. It took 3.5 seconds to sort 500 integers.

executed, and n cells are consumed each time CONSTR (having n functional arguments) is executed. The repeated execution of those primitive functions and functional forms may consume a large number of memory cells and cause inefficiency. Therefore, we attempted to avoid using them for the sake of efficiency, but eventually used many of them for the reason mentioned in (2)-<b>.

(4) The time required for executing primitive function LEN by interpreter I or I′, and the time required for executing TAIL by I′ are proportional to the length of sequences which are the arguments of the primitive functions. The usage of those primitive functions may increase the order of execution time by one. Therefore, we must be careful in using them.

CHAPTER 3

FUNCTIONAL PROGRAMMING LANGUAGE ASL/F AND ITS OPTIMIZING COMPILER

In this chapter, we present a purely functional programming language called ASL/F, and propose compiling and optimizing methods applicable to ASL/F.

The definition of ASL/F is given in Section 3.1. A method of compiling an ASL/F program is studied in Section 3.2 and Section 3.3. Several optimization problems are discussed in Section 3.4. The execution time and memory requirement of the object programs of several sample ASL/F programs such as quicksort are given in Section 3.5.

3.1 Summary of ASL/F

3.1.1 Syntax of ASL/F Programs

[Sorts, Functions and Terms]

A **Sort** is the name of a data type such as integer, boolean,... . ASL/F supports most sorts found in commonly used programming languages (e.g., integer, real, boolean, character string, array, tuple, etc.) and primitive functions (e.g., ADD, SUB, AND, OR, ...) on them. (Table 3.1 shows primitive functions which are implemented in the current version of the compiler.) By the notation $f : s_1 , s_2 ,..., s_n \rightarrow s$, we mean that function f has n-arguments of sorts s_1, s_2 ,..., s_n, and

Table 3.1: **Primitive Functions Implemented in the Current Version of the Compiler**

AND, OR, XOR, NOT

(Logical operations on booleans)

ADD, SUB, TIMES, DIV, MOD, NEG

(Arithmetic operations on integers)

EQ, NEQ, GT, GE, LT, LE

(Relational operations on integers)

CONTENT, ASSIGN

(Operations on arrays)

< >, PR1, PR2, PR3, PR4, PR5, PR6, PR7, PR8, PR9

(Operations on tuples)

Note: * AND, ... , ADD, ... , EQ, ... etc. are functions which are equivalent to the functions (operators) in commonly used languages (TIMES is multiplication).

* The value of CONTENTS(X, i) is the i-th element of array X, and ASSIGN(X, i, d) is the array obtained by replacing the i-th element of X with d.

* $<d_i, d_2, \ldots>$ is the tuple with d_1, d_2, ... as its components, and PR1(Y), PR2(Y), ... are the first component, the second component, ... of tuple Y respectively.

the value of f is of sort s.

A **term** of sort s is defined as follows. A constant of sort s or a variable of sort s is a term of sort s. For a function f such that f : s_1 , s_2 ,..., s_n -> s , if terms t_1 ,..., t_n are of sorts s_1, ..., s_n , respectively, then $f(t_1, ..., t_n)$ is also a term of sort s.

[Defined Functions and IF Functions]

There are three types of functions in ASL/F; **primitive functions, IF functions** and **defined functions.** A defined function g has its own definition statement in a source program:

$g(x_1 ,..., x_n) == right_g$

where x_1 ,..., x_n are distinct variables. $right_g$ is a term and no variables except for x_1 ,..., x_n appear in $right_g$. We call $g(x_1$,..., $x_n)$ and $right_g$ the **left-hand side** and **the right-hand side** of the definition statement for g, respectively.

The value of an IF function is either the value of the second argument if the value of the first argument is TRUE, or the value of the third argument if FALSE. For every sort s, there exists an IF function IF_s (IF_s: bool , s , s -> s) that satisfies the following two axioms :

$IF_s(TRUE , x_1, x_2) == x_1$

$IF_s(FALSE, x_1, x_2) == x_2$

For convenience, in what follows, "IF_s" is simply written as "IF" when sort s is understood.

[Structure of ASL/F Programs]

Figure 3.1 is an ASL/F program which sorts the integers (in an

```
(1) SPEC QUICKSORT ;
(2) INCLUDE ARRAY(INT, 5000, ARY) ;
(3) OP QSORT    : ARY, INT, INT                -> ARY ;
(4)    SPRQSORT : ARY, INT, INT, INT, INT, INT -> ARY ;
(5)    LEFT     : ARY, INT, INT                -> INT ;
(6)    RIGHT    : ARY, INT, INT                -> INT ;
(7)    EXCH     : ARY, INT, INT                -> ARY ;
(8)    MID      : INT, INT                     -> INT ;
(9)    INC      : INT                          -> INT ;
(10)   DEC      : INT                          -> INT ;
   AXIOM
(11)  QSORT(X, I, J) == IF( GE(I, J), X,
          SPRQSORT( X, I, J, I, J, CONTENT(X, MID(I, J))) )  ;
(12)  SPRQSORT(X, I, J, L, R, B) ==
        IF( LT(LEFT(X, L, B), RIGHT(X, R, B)),
           SPRQSORT( EXCH(X, LEFT(X, L, B), RIGHT(X, R, B)),
                     I,  J,
                     INC(LEFT(X, L, B)),  DEC(RIGHT(X, R, B)),
                     B  ),
           IF( EQ(LEFT(X, L, B), RIGHT(X, R, B)),
               QSORT( QSORT(X, I, DEC(RIGHT(X, R, B)) ),
                      INC(LEFT(X, L, B)),  J  ),
               QSORT( QSORT(X, I, RIGHT(X, R, B) ),
                      LEFT(X, L, B),  J  ) ) )                ;
(13)  LEFT(X, L, B) ==
        IF( GE(CONTENT(X, L), B), L, LEFT(X, INC(L), B) )     ;
(14)  RIGHT(X, R, B) ==
         IF( LE(CONTENT(X, R), B), R, RIGHT(X, DEC(R), B) )   ;
(15)  EXCH(X, P1, P2) == ASSIGN(ASSIGN(X, P1,CONTENT(X, P2)),
                                 P2, CONTENT(X, P1) )         ;
(16)  MID(I, J) == DIV(ADD(I, J), 2)                          ;
(17)  INC(I) == ADD(I, 1)                                     ;
(18)  DEC(I) == SUB(I, 1)                                     ;
   END
(19)  QSORT(X, 1, N)
```

SPRQSORT(X, I, J, L, R, B) sorts the elements X[I], X[I+1], ..., X[J-1] and X[J] of array X under the assumption that $X[k] \leq B$ for each k , $I \leq k \leq L$ and $X[l] \geq B$ for each l , $R \leq l \leq J$.

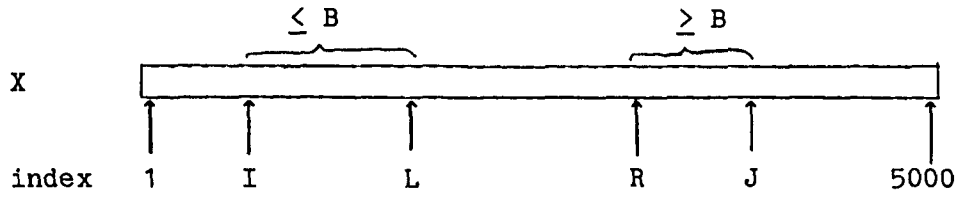


Figure 3.1: **Quicksort Program Written in ASL/F**

```
PROCEDURE QUICKSORT ;
VAR  N,I : INTEGER ;

PROCEDURE SORT(L, R :INTEGER) ;
VAR I, J, M, W :INTEGER ;
BEGIN
  I:=L ;  J:=R ;  M:=X[(L+R) DIV 2 ] ;
  REPEAT
     WHILE X[I] < M DO I:=I+1 ;
     WHILE M < X[J] DO J:=J-1 ;
     IF I<=J THEN BEGIN
       W:=X[I] ; X[I]:=X[J] ; X[J]:=W ;
       I:=I+1 ; J:=J-1
     END
  UNTIL I>J ;
  IF L < J THEN SORT(L, J) ;
  IF I < R THEN SORT(I, R)
END ;

BEGIN  SORT(1, N)  END
```

Figure 3.2: Quicksort Program Written in PASCAL

array) in nondecreasing order by "quicksort". Figure 3.2 also shows a quicksort program in PASCAL by Wirth [Wirth 76]. They are considered to be "natural implementations" of the same quicksort algorithm in those languages.

In Figure 3.1, "QUICKSORT" at line (1) is the name of this program. Line (2) is a declaration of a sort named ARY which is a one-dimensional array consisting of 5000 integers.

Lines (3) to (10) declare the sorts of the arguments and the value of each defined function. Lines (11) to (18) describe the definitions of defined functions. Line (19) specifies the term, called the **main program term**, which we want to evaluate. If the main program term has variables, then input data must be assigned to them before evaluation.

3.1.2 The Meaning of ASL/F Programs

Let P be an ASL/F program. Let **SORT(P)** denote the set of sorts of the arguments of functions or the values of functions which appear in P, and **T(P)** the set of all possible terms consisting of constants and variables of sorts in SORT(P), and those functions which appear in P.

A rewriting rule t_i -> t_j denotes that a term $\sigma(t_i)$ can be replaced by a term $\sigma(t_j)$ where $\sigma(t_i)$ and $\sigma(t_j)$ are possible terms obtained by substituting terms for variables in t_i and t_j, if any, respectively. The set of the following rewriting rules is denoted by RULE(P).

(1) A rewriting rule $g(x_1,...)$ -> $right_g$ for each defined function in P whose definition is $g(x_1,...)$ == $right_g$.

(2) Rewriting rules $IF_s(TRUE, x_1, x_2)$ -> x_1 and $IF_s(FALSE, x_1, x_2)$ -> x_2 corresponding to the axioms of IFs for each sort s in SORT(P).

(3) (Infinitely many) rewriting rules such as ADD(0, 0) -> 0, ADD(0, 1) -> 1, ADD(1, 0) -> 1, ... which define the values of the primitive functions for all possible constant arguments.

For terms $t, t' \in T(P)$, we write $t \Longrightarrow t'$ iff t' is obtained from t by rewriting a subterm of t by using a rewriting rule in RULE(P).

For term $t_1 \in T(P)$, if (1) $t_1 \Longrightarrow t_2$, $t_2 \Longrightarrow t_3$,..., and $t_{n-1} \Longrightarrow t_n$, (2) there is no t_{n+1} such that $t_n \Longrightarrow t_{n+1}$, and (3) t_n is a constant, then t_n is said to be the value of t_1 (or t_1 has value t_n) in P. For any $t \in TERM(P)$, if t has a value, then it is unique (that is, if t has both values d and d', then d=d'), because RULE(P) satisfies Church-Rosser property [Rosen 73].

Let $t_P[x_1,...,x_n]$ be the main program term with variables x_1, ... , x_n of P. For input data $D=<d_1,...,d_n>$, if a term $t_P[d_1,...,d_n]$ (obtained by the substitution of each d_i for x_i $(1 \leq i \leq n)$ in $t_P[x_1,...,x_n]$) has the value, then we call the value of $t_P[d_1,...,d_n]$ the value of program P for input data D.

3.2 Needed Vertex Sequences

3.2.1 Tree Representation of Terms

For a term t, let tree(t) = (V, E) be the directed tree which represents t, where V is the set of vertices and E is the set of edges. Each vertex v in V has a label denoted by **label(v)**. For a leaf vertex v, label(v) is either a variable name or a constant, and for a vertex v other than a leaf vertex, label(v) is a function name.

Each edge e = (u, w) in E has a positive integer i which specifies that vertex w is the i-th son of vertex u, and w corresponds to the i-th argument of the function, "label(u)". For a vertex $v \in V$, let **term(v)** be the term which corresponds to the subtree with root v. The value of v (with respect to a substitution σ) is defined to be the value of σ(term(v)), where σ(term(v)) is the term obtained from term(v) by substituting constants for variables in term(v), if any, as specified by σ. To simplify the notation, we will not write σ explicitly in the following.

When we use tree representations, we will say "rewriting vertex v" instead of "rewriting term(v)".

3.2.2 Rewriting Order

In tree(t), the first son of a vertex v whose label is an IF function, or each son of a vertex v whose label is a primitive function, is called a **needed son** of v. An edge from a vertex to its needed son is called a **needed edge**. Assume that a vertex v_1 has a son v_2. If v_2 is a needed son, then the rewriting of v_2 must precede the rewriting of v_1,

otherwise the value of v_1 cannot be obtained by the definition of the value as described in Section 3.1.2. If v_2 is not a needed son, then v_1 should be rewritten before v_2 in order to obtain the value of v_1, otherwise infinite or useless rewritings may occur.

For example, suppose that function g(x, y) is defined by g(x, y) == IF(EQ(x, 0), 3, g(x, y)) , and consider a term t = g(0, g(1, 2)). The value of t, 3, can be obtained if we rewrite the outermost occurrence of "g". If the inner "g" is rewritten, however, the resulting term is again g(0, g(1, 2)), and unless we rewrite the outer "g", the rewriting continues forever, and the value of t cannot be obtained.

3.2.3 Needed Vertex Sequences

Let v be a vertex in tree(t)=(V, E) (where t is the right-hand side of a definition or the main program term in P), and let **NEED(v)** be the set of all vertices which are reachable from v via needed edges only.

To obtain the value of v, we need to rewrite all the vertices in NEED(v) in a rewriting order called "leaf-to-root" until we obtain the values of the vertices. Here, we assume that for each vertex u in NEED(v) a total order $<<_u$ among u's needed sons is given (where $<<_u$ is called the **evaluation order** for u). The following (1) and (2) define the rewriting order of all vertices in NEED(v) to obtain the value of v, which is called the **needed vertex sequence** for v, denoted by S_v.

(1) Let $w_1, \ldots, w_k, \ldots, w_m$ be the needed sons of w in NEED(v). If w_k is the last vertex of $w_1, \ldots, w_m$ in S_v (i.e., no other vertices in $w_1, \ldots, w_m$ follow w_k), then w immediately follows w_k. (This restriction of rewriting orders reduces the number of "intermediate

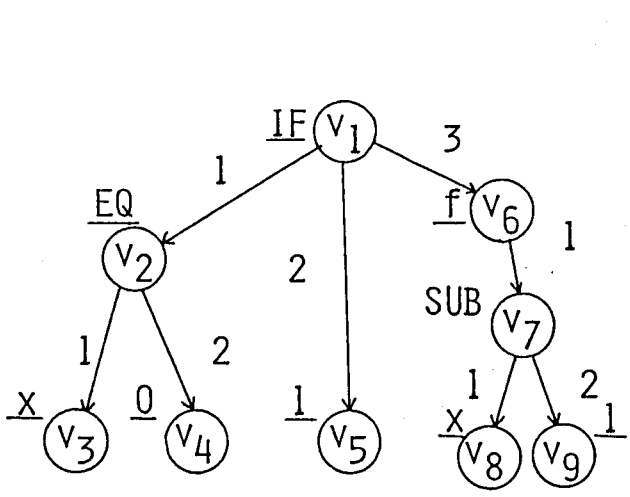
states" of the computation, which need to be saved on the run-time stack at the execution time of the object program that will be described in Section 3.3.)

(2) If $w_i <<_w w_j$ (where w_i and w_j are needed sons of w, and w is in NEED(v)), then w_i precedes w_j in S_v (the order of the needed sons of w in a needed vertex sequence agrees with the evaluation order for w, $<<_w$).

The needed vertex sequence for v defined above can be obtained by the postorder traversal [Aho et al. 83] of the subtree G of tree(t) ($G=(NEED(v), E')$, $E'=\{(u, u') \in E \mid u, u' \in NEED(v)\}$), where the sons of a vertex are visited in the same order as the evaluation order.

For example, let t = IF(EQ(x, 0), 1, f(sub(x, 1))), and for each vertex u in tree(t), let the evaluation order for u be "the i-th son of u" $<<_u$ "the j-th son of u" if i > j. (This evaluation order is called the **right-to-left order.**) Figure 3.3 shows tree(t) and the needed vertex sequence for each vertex in tree(t).

Let EO(P) denote the set of all evaluation orders for the vertices in the tree which represents the right-hand side of each definition in P. In the current version of the compiler, the right-to-left order is adopted as each evaluation order in EO(P). Needed vertex sequences are used for the object program generation (described in Section 3.3) and the optimizations (described in Section 3.4).



v:needed vertex sequence for v
v_1:v_4,v_3,v_2,v_1
v_2:v_4,v_3,v_2
v_3:v_3
v_4:v_4
v_5:v_5
v_6:v_6
v_7:v_9,v_8,v_7
v_8:v_8
v_9:v_9

Note: f is a defined function.
Right-to-left ordering is adopted for <<.

Figure 3.3 Example of Needed Vertex Sequences

3.3 Object Programs

In this section, we describe an object program which computes, when the input data is given, the value of the main program term.

Let **def(P)** be the set of all defined function names in P. Here, for the main program term t_P, we introduce an additional definition $MAIN(x_1, \ldots, x_n) == t_P$ ($x_1, \ldots, x_n$ are distinct variables in t_P) to P in order to simplify the following discussion. Let **DEF(P)** = def(P)$\cup${MAIN}, and **RIGHT(P)** = $\{right_g \mid g \in DEF(P)\}$, where $right_g$ is the right-hand side of the definition of g (we use this notation in what follows). That is, RIGHT(P) is the set of all right-hand sides of definitions and the main program term in P.

3.3.1 Procedures and the Main Program

We can easily implement a "lazy evaluation" method for arguments of defined functions by introducing subroutines called **master procedures** and **slave procedures.** An object program consists of master procedures, slave procedures, and a main program.

Master procedure : For each defined function $g \in DEF(P)$, we generate a procedure $F_g(p_1, \ldots, p_m)$ which computes the value of $right_g$. $F_g(p_1, \ldots, p_m)$ has formal parameters $p_1, \ldots, p_m$ corresponding to the variables $x_1, \ldots, x_m$ in the left-hand side of the definition of g. This procedure is called a master procedure and we may write F_g instead of $F_g(p_1, \ldots, p_m)$ for simplicity. Each actual parameter passed to a master procedure is an entry address of the slave procedure which computes the value of the argument.

Slave procedure : For each term $right_g$ ($g \in DEF(P)$) and for each

vertex v in tree($right_g$) where v is a son of a vertex u whose label is a defined function e, we generate a procedure H_v which computes the value of v. H_v is called a slave procedure of F_g, or F_g is the master procedure of H_v. H_v has no "explicit" formal parameters. When the value of u is necessary for obtaining the value of $right_g$, master procedure F_e is activated, and the entry address of H_v is passed to F_e as an actual parameter. H_v is activated at the time when the value of v is actually necessary for the computation in F_e or in a slave procedure of F_e.

Main program : The main program executes the following (1) to (3). Let t_p be the main program term with variables $x_1, \ldots, x_n$.

(1) Read input data $d_1, d_2, \ldots, d_n$.

(2) Activate master procedure $F_{MAIN}(p_1, \ldots, p_n)$ with actual parameters $d_1, \ldots, d_n$.

(3) Output the value returned by $F_{MAIN}(d_1, \ldots, d_n)$.

3.3.2 Frame Usage

Activated procedures use memory blocks called **frames.** There are two kinds of frames, an **M-frame** and an **S-frame.** They are allocated on a single LIFO run-time stack. At the time of the activation of a master procedure F_g, an M-frame for F_g is allocated on the top of the run-time stack (see Figure 3.4), and the values needed or obtained in the computation in F_g are referred to or stored into the M-frame (we will describe this referring or storing in Section 3.3.3 (a)-(e)). While F_g is active, the top of the M-frame is pointed by **Stack Pointer (SP),** and the bottom is pointed by **Frame Pointer (FP).**

The M-frame for F_g consists of (1) the value of the frame pointer

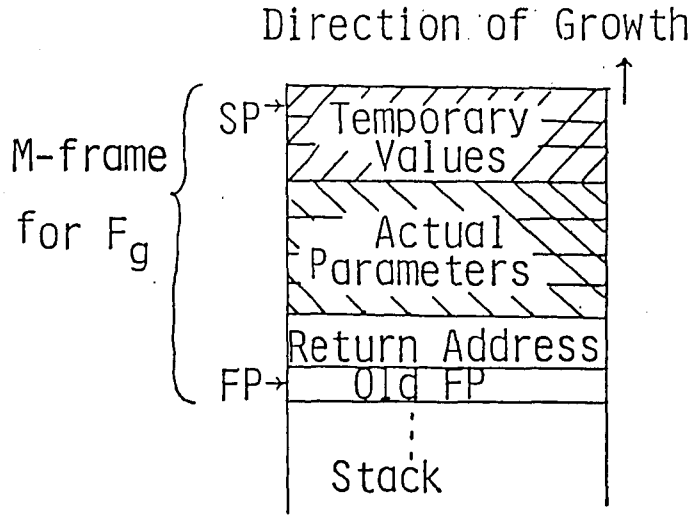


Figure 3.4 Stack when Master Procedure F_g is Active

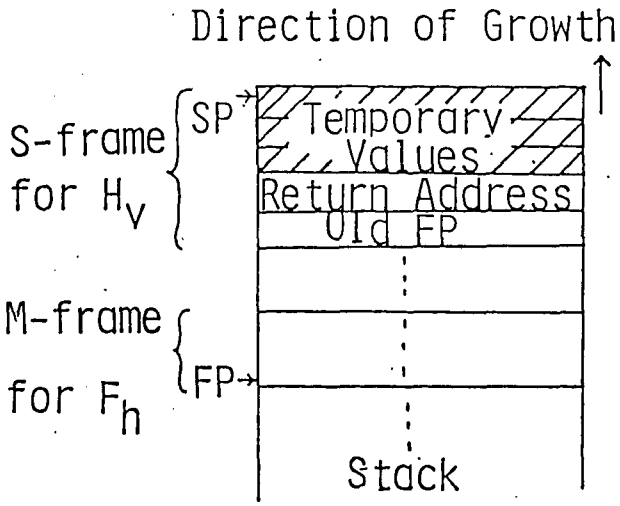


Figure 3.5 Stack when Slave Procedure H_v (of Master Procedure F_h) is Active

FP when the procedure which activated F_g was active (we call this value **old-FP**), (2) the return address, (3) the actual parameters (i.e., the entry addresses of slave procedures of the procedure which activated F_g), and (4) temporary values (the number of the temporary values are not fixed while F_g is active).

On the other hand, at the time of the activation of a slave procedure H_v (of a master procedure F_h), an S-frame for H_v is allocated on the top of the run-time stack (see Figure 3.5). The temporary values generated in H_v are stored in the S-frame for H_v (see Section 3.3.3 (a)-(d)). In addition to the S-frame, H_v uses the M-frame for F_h to refer to the actual parameters of F_h (see Section 3.3.3 (e)). While H_v is active, the top of the S-frame for H_v is pointed by SP and the bottom of the M-frame for F_h is pointed by FP.

The S-frame for H_v consists of (1) old-FP, (2) the return address, and (3) temporary values (the number of the temporary values are not fixed while H_v is active).

3.3.3 Instruction Sequences

Let a vertex r be the root of tree($right_g$) where $g \in DEF(P)$. A master procedure F_g consists of the following instruction sequences,(1) and (2) (CODE(r) will be explained later).

Master Procedure F_g:

(1) An instruction sequence CODE(r) which computes the value of r.

(2) An instruction sequence to return the computed value and the control to the procedure which activated F_g.

On the other hand, slave procedure H_v consists of the following instruction sequences, (1) and (2).

Slave Procedure H_v:

(1) An instruction sequence CODE(v) which computes the value of v.
(2) Instructions to return the computed value and the control to the procedure which activated H_v.

Let u be a vertex in tree($right_g$) ($g \in DEF(P)$) and let $S_u = v_1, v_2, \ldots, v_k$ be the needed vertex sequence for u. **CODE(u)** denotes an instruction sequence $Iv_1, Iv_2, \ldots, Iv_k$ to compute the value of u where Iv_i is an instruction sequence defined by label(v_i) as follows.
(a) If label(v_i) is a primitive function, Iv_i is an instruction sequence which computes the value of the primitive function. The values of the arguments of the primitive function have been already computed and stored as temporary values in the (M- or S-) frame on the top of the run-time stack. As the other instruction sequences from (b) to (e), the execution result is stored to the top of the frame, after the values of arguments (necessary for the computation of the primitive function) have been removed.
(b) If label(v_i) is a constant c, then Iv_i is an instruction sequence to generate the constant value c.
(c) If label(v_i) is an IF function, then Iv_i executes the following: test the value of the first son v_{i1} of v_i (the value of v_{i1} has been already computed by an instruction sequence that precede Iv_i). If it is TRUE, then execute CODE(v_{i2}), and otherwise execute CODE(v_{i3}), where v_{i2} and v_{i3} are the second and the third sons of v_i, respectively (neither v_{i2} nor v_{i3} is contained in S_u). CODE(v_{i2}) and CODE(v_{i3}) are also included in Iv_i.
(d) If label(v_i) is a defined function q, then Iv_i is an instruction sequence to activate a master procedure F_q, that is, the control is transferred to the entry address of F_q, after (1)the current value of

FP, (2)the return address, and (3)the entry addresses of the slave procedures of F_g (these slave procedures compute the values of the sons of v_i) are saved on the top of the run-time stack.

(e) If label(v_i) is a variable x_j, then Iv_i is an instruction sequence to activate a slave procedure H_{w_j} of a master procedure which activated F_g or a master procedure whose slave procedure activated F_g. The entry address of H_{w_j} has been already passed to F_g as the j-th actual parameter of F_g. That is, the control is transferred to the entry address of H_{w_j} after the current value of FP and the return address are saved on the top of the run-time stack.

3.4 Optimizations

3.4.1 Pre-computing Needed Arguments of Defined Functions

3.4.1.1 Needed-Argument-First Order

If, in order to get the value of $g(t_1, \ldots, t_n)$ (where g is a defined function [$g \in def(P)$]), we always need to obtain the value of t_i regardless of the term rewriting order and the values of $t_1, \ldots, t_n$, then we say that the i-th argument of g is a **needed argument** or simply that the i-th argument of g is **needed.** For example, let the definition of g be

g(x, y, z) == IF(EQ(x, 0), ADD(y, 1), SUB(y, z)).

The value of EQ(x, 0) is necessary to obtain that of $right_g$ (the right-hand side of the definition of g), and so is the value of x to obtain that of EQ(x, 0). Thus, the first argument x of g is needed. The value of either ADD(y, 1) or SUB(y, z) is necessary to obtain that of $right_g$, and so is the value of y to obtain that of any one of ADD(y,1) and SUB(y,z). Therefore, the second argument y of g is also needed.

As described in Section 3.2.2, computing the value of an argument of a defined function g before activating master procedure F_g may cause useless or infinite computation even if the value of g is defined. (To avoid this, we have implemented the lazy evaluation method.) But rearranging the computation order in such a way that the values of needed arguments are computed before activating F_g does not cause useless or infinite computation if the value of g is defined, since those of the needed arguments are strictly necessary for finding the value.

The following computation order for a needed argument x of a

defined function g is called a **needed-argument-first** order for x: The value of x is computed before the activation of master procedure F_g, and the computed value of the argument is passed to F_g as an actual parameter (passed by value).

3.4.1.2 Effects of Needed-Argument-First Order

The object program for an ASL/F program P as described in Section 3.3 is denoted by O_P, and the object program which is executed in the needed-argument-first order for every needed argument in P is denoted by O_P'. In general, O_P' is more efficient than O_P due to the following reasons.

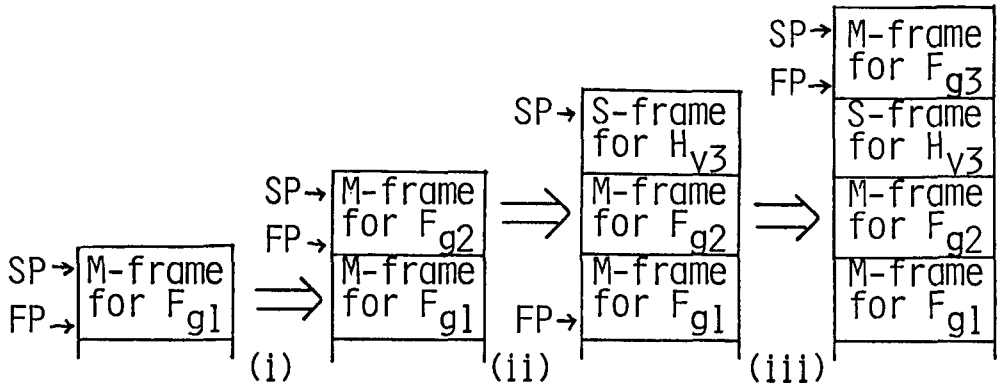
(1) The activations of the slave procedures corresponding to needed arguments in O_P are not necessary in O_P'. Therefore, the overhead due to the activations (e.g., saving and restoring the registers and changing SP and FP) can be saved in O_P'.

(2) The maximum run-time stack length at the execution time of O_P' may be reduced greatly. For example, let $term(v_2)$=g2(g3(...), ...) for a vertex v_2 in $tree(right_{g1})$ where g1, g2, and g3 are defined functions and $right_{g1} \in RIGHT(P)$, and let v_3 be the first son of v_2 (i.e., $term(v_3)$=g3(...)). Suppose that we need the value of v_2 in the computation of master procedure F_{g1}, and the first argument of g2 is needed . By the execution of O_P, the run-time stack will grow as follows (see Figure 3.6-(1)):

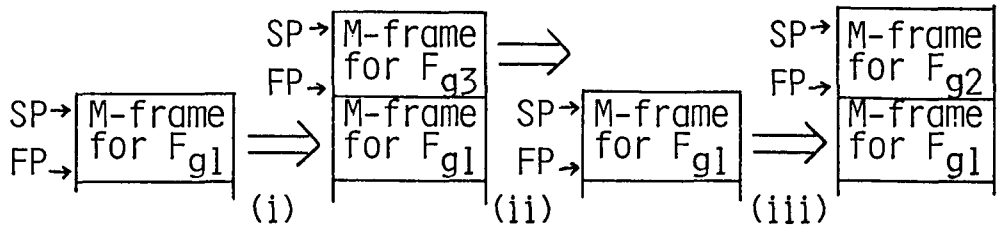
(i) At the time of the activation of master procedure F_{g2} in F_{g1}, an M-frame for F_{g2} is allocated.

(ii) At the time of the activation of slave procedure H_{v_3} (of F_{g1}) in F_{g2}, an S-frame for H_{v_3} is allocated.

(iii) At the time of the activation of master procedure F_{g3} in H_{v_3}, an



(1): Non needed-argument-first



(2): Needed-argument-first

Figure 3.6 Decrease of the Maximum Stack Length by Needed-Argument_First

M-frame for F_{g3} is allocated.

Thus, the M-frames for F_{g2} and F_{g3}, and the S-frame for H_{h3} are on the run-time stack at the same time.

On the other hand, the run-time stack will grow as follows by the execution of O_P' (see Figure 3.6-(2)):

(i) At the time of the activation of F_{g3} in F_{g1}, an M-frame for F_{g3} is allocated.

(ii) At the time of the completion of F_{g3}, an M-frame for F_{g3} is deallocated.

(iii) At the time of the activation of F_{g2} in F_{g1}, an M-frame for F_{g2} is allocated.

Thus, no S-frame for H_{v3} is necessary. Furthermore, the M-frames for F_{g2} and F_{g3} are not on the run-time stack at the same time.

3.4.1.3 Implementation of Needed-Argument-First order

It is not always easy to know whether an argument of a defined function in P is needed or not*. In Appendix A, we show a sufficient condition for an argument of a defined function in P to be needed. The correctness of this condition is shown in [Inoue et al. 82b] [Seki et al. 84]. This condition is so effective that all of the needed arguments in sample ASL/F programs, which will be described in Section 3.5, satisfy this condition. But in general, as mentioned above, it is

---

* The general idea of "needed" comes from [Huet and Lévy 79] and [Sugiyama et al. 82], where only the left-hand sides of the rewriting rules are considered to determine whether they are needed or not. In this sufficient condition, the right-hand sides are also taken into account.

difficult to detect the set (denoted by AN(P)) of all the needed arguments of defined functions in P. Therefore, a subset of AN(P), denoted by SN(P), is assumed to be chosen in the following discussions.

In addition to needed sons described in Section 3.2.3, let us call the i-th son of a vertex whose label is a defined function $g \in def(P)$ a needed son also, if the i-th argument of g is in SN(P). We also define needed vertex sequences in the same way as described in Section 3.2.3. These needed sons and needed vertex sequences are used in what follows.

The generation of the object programs described in Section 3.3 is modified as described below so that the generated object program (with respect to an EO(P) and an SN(P)) is executed in the needed-argument-first order for the needed arguments in SN(P).

(1) For each needed son v corresponding to a needed argument in SN(P), we generate no slave procedure H_v and pass the computed value of v before activation as an actual parameter instead of the entry address of H_v.

(2) For each vertex v_i whose label is a variable x corresponding to a needed argument in SN(P), we replace the instruction sequence Iv_i (which activates a slave procedure) with an instruction sequence which simply refers to the actual parameter.

3.4.2 Avoidance of Duplicate Computation for Common Subterms

The object program described in Section 3.3 may repeatedly compute the values of vertex v_i and v_j in $tree(right_g)$ ($g \in DEF(P)$), such that $term(v_i) = term(v_j)$. Here, we show a method of eliminating such a duplicate computation.

3.4.2.1 Common Vertices in Trees

Let tree($right_g$) = (V, E) ($g \in DEF(P)$), and v_1, ..., v_n ($n \geq 2$) be distinct vertices in V such that

(1) $term(v_1) = \ldots = term(v_n)$ (i.e., each subtree whose root is v_1, ..., v_n is identical to each other),

(2) for any $u \in V$ other than v_1, ..., v_n, $term(u) \neq term(v_1)$, and

(3) there exist vertices v_i and v_j ($i \neq j, 1 \leq i \leq n, 1 \leq j \leq n$) such that $term(u_i) \neq term(u_j)$ where u_i and u_j are parents of v_i and v_j, respectively.

Then, the vertex set $\{v_1, \ldots, v_n\}$ is called a **common vertex set** and vertex set $\{v_1, \ldots, v_n\}$ is denoted by **COMMON(v_i)** for any v_i ($1 \leq i \leq n$). A vertex in a common vertex set is called a **common vertex.**

3.4.2.2 Elimination Method

We modify the instruction sequence of F_g and its slave procedures in the following way:

For each distinct common vertex set C in tree($right_g$), extra space a_C and b_C described below are added to the M-frame for F_g, which F_g and its slave procedures use.

a_C : a space for the value of a vertex in C.*

---

* If the following (1) and (2) hold, there is no need to add the new space a_C but the space for an actual parameter p_k in the M-frame can be used for a_C: (1) The label of each vertex in the common vertex set C is a variable. (2) The size of a_C does not exceed the size of the space for p_k.

b_C : a space for a flag indicating whether the value to be stored in a_C is "already computed" or "not yet computed". Initially b_C is set to "not yet computed".

(2) Before executing an instruction sequence CODE(v), which computes the value of vertex v in C, the flag in b_C is checked. (i) If the flag is "not yet computed", then CODE(v) is executed, and after the execution of CODE(v), the value of v is stored in a_C, and b_C is set to "already computed". Otherwise CODE(v) is not executed but instead a_C is referred to.

Let CMN-CODE(v) denote the modified instruction sequence of CODE(v) as described above.

3.4.3 Elimination of Redundant Flag Tests

Let O_P'' be the object program of P which, for each common vertex set, computes the values of the common vertices only once in the way described in Section 3.4.2. For vertices in tree($right_g$) ($g \in DEF(P)$), let COMMON(v_1) = $\{v_1, \ldots, v_m\}$ = C where $m \geq 2$. We say that v_i is "not preceded", if the result of the flag test in the instruction sequence CMN-CODE(v_i) (which obtains the value of v_i by computing or referring as mentioned in Section 3.4.2) in O_P'' is always "not yet computed" for any input data D of program P.

On the other hand, we say that v_i is "preceded" if the result of the flag test in the instruction sequence CMN-CODE(v_i) in O_P'' is always "already computed" for any D of P.

Sufficient conditions for v_i to be "not preceded" or "preceded",

discussed in [Seki et al. 84], are shown in Appendix B. If a common vertex v_i satisfies the sufficient conditions, we can modify an instruction sequence CMN-CODE(v_i) as follows.

If v_i is "not preceded", then the instructions for the flag test are deleted from CMN-CODE(v_i), so that the computation of the value of v_i can start immediately. If v_i is "preceded", then the instructions for the flag test, the computation of the value, and the assignment of the computed value are deleted from CMN-CODE(v_i), so that a reference only to the value computed already is made.

If each common vertex of C is either "not preceded" or "preceded", then the space for the flag is not necessary.

3.4.4 Globalization of Sorts

Let $\underline{O}_P$ be the object program (with respect to an EO(P) and an SN(P)) generated according to the needed-argument-first order described in Section 3.4.1 and the avoidance of duplicate computation for common subterms described in Section 3.4.2. (The other optimizations do not affect the argument here directly.)

If, at a point q of the execution of $\underline{O}_P$, a value A of a sort s has been already obtained and is used for subsequent computation, then we say that A is live at q; otherwise A is dead at q. If the number of live values of sort s does not exceed one at any point of the execution of $\underline{O}_P$ for any input data D, we say that s is **globalizable** in $\underline{O}_P$.

In Appendix C, we show a sufficient condition, presented by [Seki et al. 84], for a sort s in a program P to be globalizable (with respect to EO(P) and SN(P)). The sort ARY appearing in the quicksort program

shown in Figure 3.1 satisfies this condition.

If a sort s is globalizable in P, the object program which is obtained by modifying $\underline{O}_P$ as described below in (1) to (4) also computes the value of program P. This modification of the object program is called a **globalization** of sort s.

(1) At compile-time, we statically allocate space W_s, which is sufficient to hold any value of sort s, out of the run-time stack.

(2) For each instruction which refers to a value of sort s, we read the value of sort s from W_s, and for each instruction whose execution generates a value of sort s, we store the result into W_s.

(3) For each (master or slave) procedure which computes a value of sort s, we return the value of sort s not on the run-time stack but in W_s.

(4) For each needed argument of g in SN(P) described in 5.1, whose sort is s, we store the value of sort s (as an actual parameter corresponding to the needed argument) not on the run-time stack but in W_s.

3.4.5 Elimination of Tail Recursions

Let the definition of g be $g(x_1, \ldots, x_n) == right_g$. Let r be the root of $tree(right_g)$, and u be a vertex in $tree(right_g)$ whose label is g. If g and u satisfy the following conditions, (1) and (2), we call u a **tail recursive vertex** of g [Aho et al. 83].

(1) The label of every vertex except for u on the path from r to u is an IF function, and each edge on the path connects a vertex with either its second or third son.

(2) For each son v_i of u (let v_i be the i-th son of u), either v_i is a needed son of u (i.e., the i-th argument of g is needed), or the label of v_i is a variable x_i (i.e., $term(v_i) = x_i$).

Consider an instance I of a master procedure F_g, which uses an M-frame FR. After the value of tail recursive vertex u is computed by the activation of another instance I′ of F_g whose M-frame is denoted by FR′, no instruction sequences except for the instruction sequence which terminates I are executed in I. Furthermore, any value in FR is not necessary for I′. Therefore, at the time of the activation of I′ , FR on the top of the run-time stack can be deleted and FR′ can be allocated on the same location where FR was (old-FP and the return address in FR′ are the same as those in FR). If we do so, the maximum stack length and the execution time of the object program will decrease greatly.

3.4.6 Elimination of Auxiliary Functions

Let the following modification of the source program P be called **elimination of auxiliary functions** P :

If a subterm of each term in RIGHT(P) can be rewritten by the rewriting rules in RULE(P) without knowing the values assigned to variables, we rewrite (expand) it repeatedly before compilation (except for the rewritings of the functions defined recursively so that the rewritings will terminate) and replace the right-hand side of the function definition or the main program term of P with the terms obtained by such rewritings.

The advantages of the elimination of auxiliary functions are as follows :

(1) The number of procedure activations at the time of execution may decrease.

(2) The number of common subterms in each term in RIGHT(P) may increase and duplicate computation may be avoided by the optimization in Section 3.4.2.

(3) In order to increase readability of a program, we can introduce auxiliary defined functions without causing inefficiency.

The elimination of auxiliary functions enlarges the size of the object programs. And sometimes it may cause some inefficiency at the time of execution, because of the increase of the frame size corresponding to the increase of the number of common vertices in the term in RIGHT(P).

3.5 Sample ASL/F Programs

3.5.1 Outline of the System

The optimizing ASL/F compiler runs under the UTS/VS operating system on a MELCOM COSMO 900-II (it executes about 6 million instructions per second), and it compiles an ASL/F program into an object program in an assembly language META-SYMBOL. The compiler was written in PASCAL, and is about 4000 lines long (including parsing, optimizing, and code generation routines). It took about 7 man-months to design and implement.

In the current version of the compiler, primitive functions on integers, booleans, arrays, and tuples have been implemented as shown in Table 3.1, and the right-to-left order is adopted as each evaluation order in EO(P) as mentioned in Section 3.2.3.

It always avoids the duplicate computation for common subterms as described in Section 3.4.2. And the compiler generates an object program only when the sorts of arrays in the source program are globalizable as described in Section 3.4.4.* Programmers can specify whether or not each of the other optimizations described in Section 3.5 is to be performed. In addition to these optimizations, the compiler always carries out

---

* In ASL/F, we can declare more than two sorts for the same data type, an array (i.e., the dimension, the number of elements, the sort of each element are identical). Thus, when as ASL/F program P containing a sort of arrays is not globalizable, we may be able to modify P by using more than two sorts of the arrays so that each sort of the arrays can be globalizable.

minor improvements of the object programs such as the removing of obviously redundant transfer instructions.

3.5.2 Effects of Optimizations

We examined the effects of the optimizations by executing several sample programs. Algorithms to solve the following problems have been programmed in ASL/F:

(1) Sorting (Quicksort): program is shown in Figure 3.1.

(2) Sorting (Bubblesort)

(3) Towers of Hanoi

(4) The computation of the base of natural logarithm, "e"

(5) Matrix multiplication

The programs are written in such a way that they can be considered to be "natural implementations" of the algorithms in ASL/F.

Table 3.2 shows the execution time and the maximum run-time stack length* of those ASL/F programs. These experimental results demonstrate that all of the optimizations adopted here are useful in reducing the execution time and/or the maximum run-time stack length. Especially, (1) pre-computing needed arguments of defined functions is effective to reduce both of the execution time and the maximum run-time stack length greatly, and (2) in some cases, the elimination of tail recursions reduced the maximum run-time stack length considerably.

---

* In addition to the memory space for the run-time stack, the space for the object program itself and for holding globalizable arrays as mentioned in Section 3.4.4 are required.

Table 3.2: **Effects of Optimizations**

Execution time (milli sec.)

Program \ Optimization method		a,b	a,b,c	a,b,c, d	a,b,c, d,e	a,b,c, d,e,f
Quicksort	5000 integers	3200	990	830	670	430
	50 integers	53	18	14	10	6
Bubblesort	1000 integers	[20000]	7000	5900	3800	2400
Towers of Hanoi	10 stairs	160	20	20	15	10
	15 stairs	[5000]	630	630	460	350
Calculation of e	100 digits	250	64	52	48	23
	1000 digits	[14000]	3500	3200	2800	1300
Multiplication of two (50,50)-matrices		4700	1100	650	640	640

The maximum run-time stack length (word = 32 bits)

Program \ Optimization method		a,b	a,b,c	a,b,c, d	a,b,c, d,e	a,b,c, d,e,f
Quicksort	5000 integers	73700	49900	529	433	433
	50 integers	31600	225	27	24	17
Bubblesort	1000 integers	[$1.2*10^7$]	4030	27	24	17
Towers of Hanoi	10 stairs	37800	194	194	163	162
	15 stairs	[$1.2*10^6$]	285	285	238	242
Calculation of e	100 digits	44300	525	33	31	20
	1000 digits	[$2.8*10^6$]	3670	33	31	20
Multiplication of two (50,50)-matrices		44100	818	36	36	31

Square brackets denote that the object programs could not be executed because of memory shortage. The values in them are our estimates.

Optimization method:
a: Avoidance of duplicate computation for common subterms.
b: Globalization of arrays.
c: Pre-computaing needed arguments of defined functions.
d: Elimination of tail recursions.
e: Elimination of redundant flag tests.
f: Elimination of auxiliary functions.

For example, when pre-computing needed arguments of defined functions is not performed for Towers of Hanoi, the maximum run-time stack length is an order of exponential of n, where n is the number of stairs. On the other hand, when it is performed, the maximum run-time stack length is linear in n as shown in Table 3.2. (In both case, optimizations (a) and (b) are performed.)

When the elimination of tail recursions is not performed for Bubblesort, the maximum run-time stack length is approximately linear in the number of integers to sort. On the other hand, when it is performed, the maximum run-time stack length is fixed to a constant, 27 words. (In both cases, optimizations (a), (b), and (c) are performed.)

3.5.3 Comparison Between ASL/F and PASCAL Programs

ASL/F programs mentioned above, and PASCAL programs, which implement the same algorithms as ASL/F programs were executed on the MELCOM COSMO 900-II. The execution time is shown in Table 3.3. These PASCAL programs are natural implementations of the algorithms in PASCAL, where, for example, iterations such as FOR, WHILE, etc. are used effectively, and redundant computation is eliminated by using variables to store temporary values. ASL/F programs were compiled according to all the optimizations described in Section 3.4. The quicksort program in PASCAL used here is shown in Figure 3.2 and is by Wirth [Wirth 76]. PASCAL programs were compiled by a MELCOM PASCAL 8000 compiler which generates machine codes directly. The time required for compiling the quicksort program, for example, was as follows:

(a) From ASL/F to the object program in the assembly language META SYMBOL: 0.6 second (including 0.1 second for optimizations),

(b) From PASCAL to machine codes : 0.3 second.

Table 3.3: **Execution Time of ASL/F and PASCAL Programs**

(milli seconds)

Program (data size)	ASL/F	PASCAL
Quicksort (5000 integers)	430*	320**
Bubblesort (50 integers)	6	8
(1000 integers)	2400	3200
Towers of Hanoi (10 stairs)	10	11
(15 stairs)	350	370
The calculation of the base of natural logarithm (100 digits)	23	28
(1000 digits)	1300	1500
Multiplication of two matrices (50*50)	640	820

* This program is shown in Fig. 1.
** This program is by Wirth [Wirth 76] and shown in Figure 2.

3.5.4 Writing an Interpreter in ASL/F

As an example of fairly long ASL/F programs, we have written an ASL/F program which interprets (parses and executes interpretively) ASL/F programs. (In source programs to be interpreted, only integer and/or boolean sorts can be used.) The source program to be interpreted is given as a form consisting of an integer array of character codes, since the current version of our compiler does not support the sorts such as characters or strings. The parser transforms the character code sequence in the array into integer sequences each of which represents the term on the right-hand side of a definition. The execution part rewrites the integer sequences according to the rewriting rules. Table 3.4 shows the size characteristics of this interpreter program. (If functions on characters or string are implemented in the compiler, each size may be reduced.)

It took about 2 man-months to design and complete this interpreter program. At the beginning of its development, we had some syntactic errors but only two logical errors. Several programs were executed by this interpreter. The execution time of Ackermann function ACK(3, 3), for example, was 3.6 seconds. (The program of the Ackermann function was compiled by the optimizing compiler and executed also. It took 0.01 second to compute ACK(3, 3)).

Table 3.4: **Size Characteristics of the ASL/F Interpreter Written in ASL/F**

	Parsing part	Execution part
The number of defined functions	110	70
The maximum number of arguments of defined functions	10	6
The average number of arguments of defined functions	4	4
The maximum nesting depth of functions in the right-hand sides of definition statements	7	11
The average nesting depth of functions in the right-hand sides of definition statements	3	3
The number of lines in the text	350	200

CHAPTER 4

CONCLUSION

We have described two purely functional languages, methods of interpreting or compiling their programs on conventional machines, and execution results of several sample programs by using the systems constructed for those languages.

In Chapter 2, we have shown two systems for a functional programming language FPL based on the Backus' FP System. Both of them have been developed with little effort. The efficiency of both is satisfactory for systems to perform small-scale computation.

While FPL is very simple, it has particular properties such that every function has a single argument and no other data types except for tree can be processed. By these properties, it becomes difficult to increase the efficiency of the systems as described in Section 2.6.

For the sake of the general applicability of the system, the language must be extended by, for example, increasing the number of arguments and adding data structures such as arrays.

In Chapter 3, we have discussed the compiling and optimizing method for a functional programming language ASL/F and shown that those methods are effective to increase the time and space efficiency. The execution time of an ASL/F program is about 75 to 135% of that of a PASCAL program using the same algorithm.

If optimization techniques adopted in compilers of procedural languages are used to improve the object program, the time and space efficiency will be increased further.

It would be possible to improve the sufficient conditions for arguments to be needed, for flag tests to be eliminated, and for sorts to be globalizable. Although in the current version of the compiler each evaluation order in EO(P) is the right-to-left one as described in Section 3.3, the execution time will be reduced further if a better EO(P) can be found.

We are developing a programming system for ASL/F with some syntactic sugar on a VAX-11/780, which consists of an editor, a debugger and a new compiler. The new compiler is written in ASL/F itself and generates an object program in language C.

APPENDIX A

A SUFFICIENT CONDITION FOR AN ARGUMENT TO BE NEEDED

Let the definition of each defined function g in P (i.e., $g \in def(P)$) have the form $g(x_1, x_2, \ldots, x_{n_g}) == right_g$. Now, we introduce boolean variables

$Z[g, i]$ for each $g \in def(P)$ and integer i , $1 \leq i \leq n_g$ (if $Z[g, i]$ is TRUE, then the i-th argument of g is needed), and

$Y[g, t, x_i]$ for each $g \in def(P)$, each subterm t of $right_g$ and integer i , $1 \leq i \leq n_g$ (if $Y[g, t, x_i]$ is TRUE, then the value of x_i is necessary to obtain the value of t).

We set up equations as follows.

(1) for each $g \in def(P)$ and integer i , $1 \leq i \leq n_g$:

$$Z[g, i] = Y[g, right_g, x_i]$$

(2) for each $g \in def(P)$, each subterm t of $right_g$ and integer i , $1 \leq i \leq n_g$:

a. if t is a constant, $Y[g, t, x_i] = FALSE$

b. if t is a variable x_i, $Y[g, t, x_i] = TRUE$

c. if t is a variable other than x_i, $Y[g, t, x_i] = FALSE$

d. if t is $IF(t_1, t_2, t_3)$,

$$Y[g, t, x_i] = Y[g, t_1, x_i] \vee \{Y[g, t_2, x_i] \wedge Y[g, t_3, x_i]\}$$

(If the value of x_i is necessary for t_1, or both for t_2 and t_3, then it is necessary for t.)

e. if t is $f(t_1, \ldots, t_m)$ where f is a primitive function,

$Y[g, t, x_i] = Y[g, t_1, x_i] \vee \cdots \vee Y[g, t_m, x_i]$

(If the value of x_i is necessary for any of t_1, ... , t_m, then it is necessary for t.)

f. if t is $g'(t_1, \ldots, t_m)$ where $g' \in def(P)$,

$Y[g, t, x_i] = \{Z[g', 1] \wedge Y[g, t_1, x_i]\} \vee \cdots \vee$

$\{Z[g', m] \wedge Y[g, t_m, x_i]\}$

(If there exists an integer j ($1 \leq j \leq m$) such that the value of x_i is necessary for t_j and the j-th argument of g' is needed, then the value of x_i is necessary for t.)

A sufficient condition for the k-th argument of g in P to be needed is that there is a solution which satisfies Z[g, k]=TRUE.

APPENDIX B

SUFFICIENT CONDITIONS FOR A VERTEX TO BE "NOT PRECEDED" AND THAT TO BE "PRECEDED"

Let tree($right_g$) = (V, E) ($g \in DEF(P)$). Let $COMMON(v_1) = \{v_1, \ldots, v_m\} = C$ where $m \geq 2$.

For each pair v_a and v_b in C, let the lowest common ancestor of v_a and v_b be denoted by r_{ab} as shown in Figure B.1, and let u_a and u_b be the sons of r_{ab} which are ancestors of v_a and v_b, respectively. (Note that a vertex w is an ancestor of w itself. u_a and v_a may be the same vertex or u_b and v_b may be the same vertex.)

(1) A sufficient condition for v_i to be "not preceded" in O_P'':

For each $v_k \in C$ other than v_i, at least one of the following, (1a), (1b) and (1c), holds (see Figure B.2) :

(1a) $label(r_{ki})$ is an IF function and u_k and u_i are the second and third (or third and second) sons of r_{ki}, respectively. (The computation for u_k is exclusive of that for u_i, or vice versa, i.e., if the computation for u_i is executed, that for u_k has never been executed.)

(1b) u_i is contained in needed vertex sequence $S_{r_{ki}}$ for r_{ki}, and u_k is not contained in $S_{r_{ki}}$. (If the computation for u_i is executed, it always precedes that for u_k, or the computation for u_k may not be executed.)

(1c) Both u_k and u_i are contained in $S_{r_{ki}}$ and u_i precedes u_k in $S_{r_{ki}}$. (If the computation for u_i is executed, it always precedes that for u_k.)

(2) A sufficient condition for v_i to be "preceded" in O_P'':

There exists at least one $v_k \in C$ which satisfies at least one of

the following, (2a) and (2b) (see Figure B.3) :

(2a) v_k is contained in $S_{r_{ki}}$, and u_i is not contained in $S_{r_{ki}}$.

(2b) Both v_k and u_i are contained in $S_{r_{ki}}$ and v_k precedes u_i in $S_{r_{ki}}$.

(Both (2a) and (2b) imply that if the computation for v_i is executed, then there exists a vertex v_k such that the computation for v_k always precedes that for v_i.)

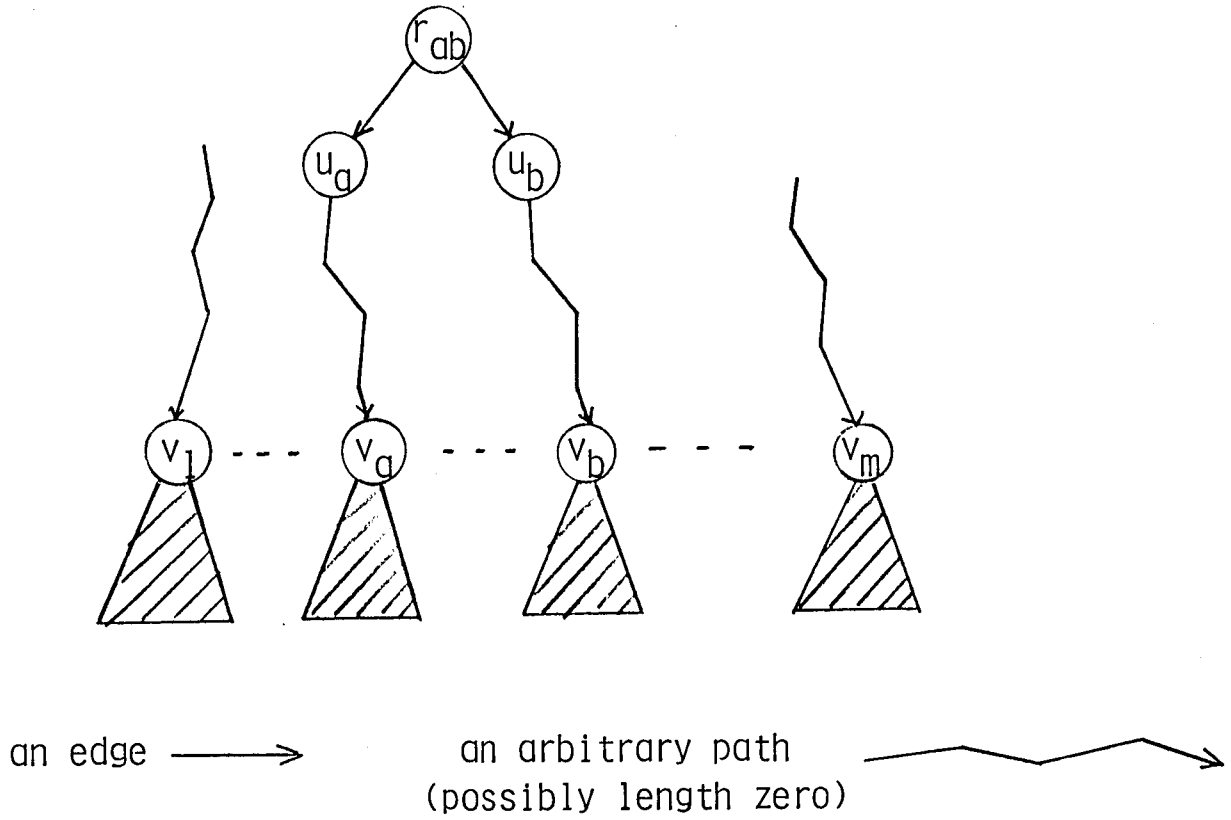


Figure B.1 Lowest Common Ancestor r_{ab} of v_a and v_b

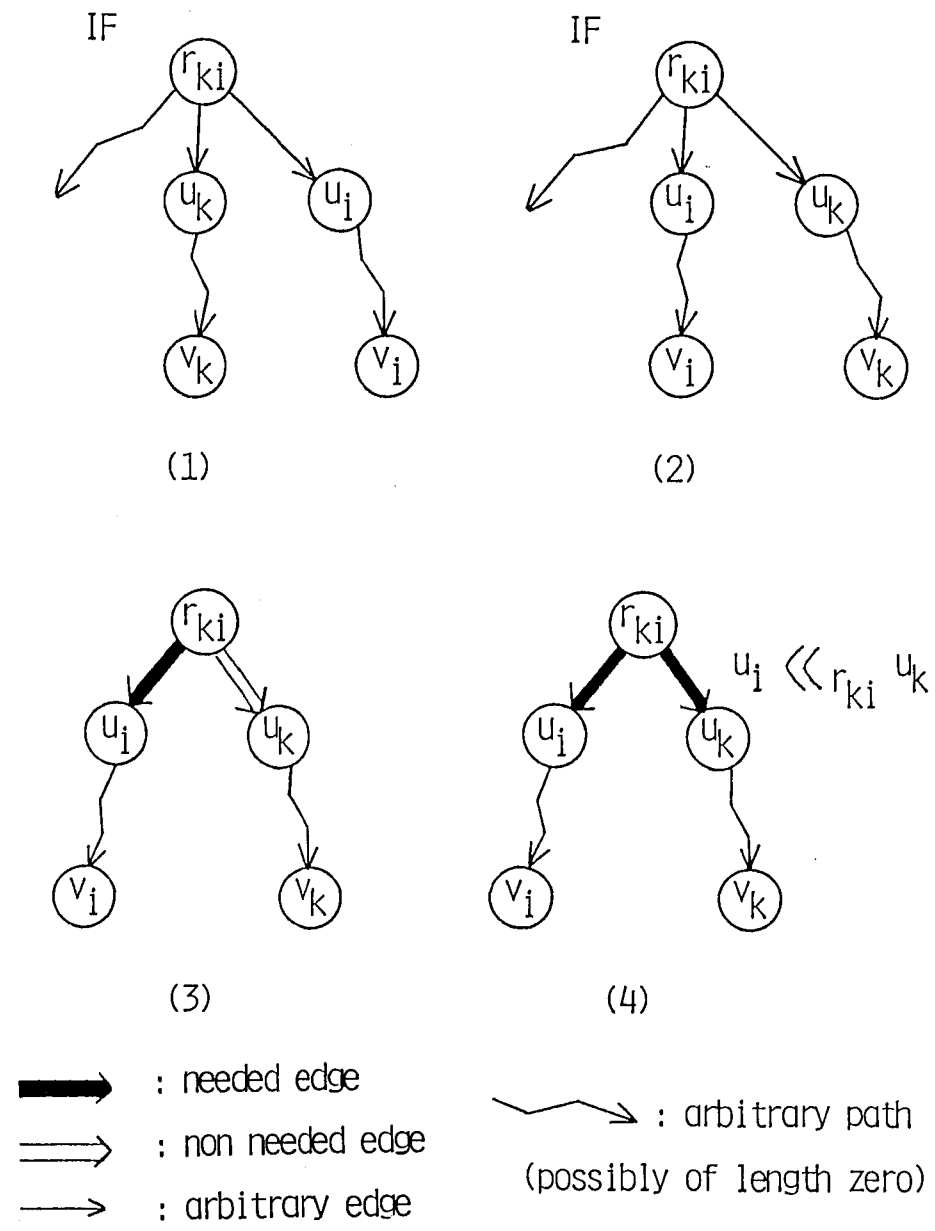


Figure B.2 Illustrations for (1a),(1b) and (1c) in Appendix B

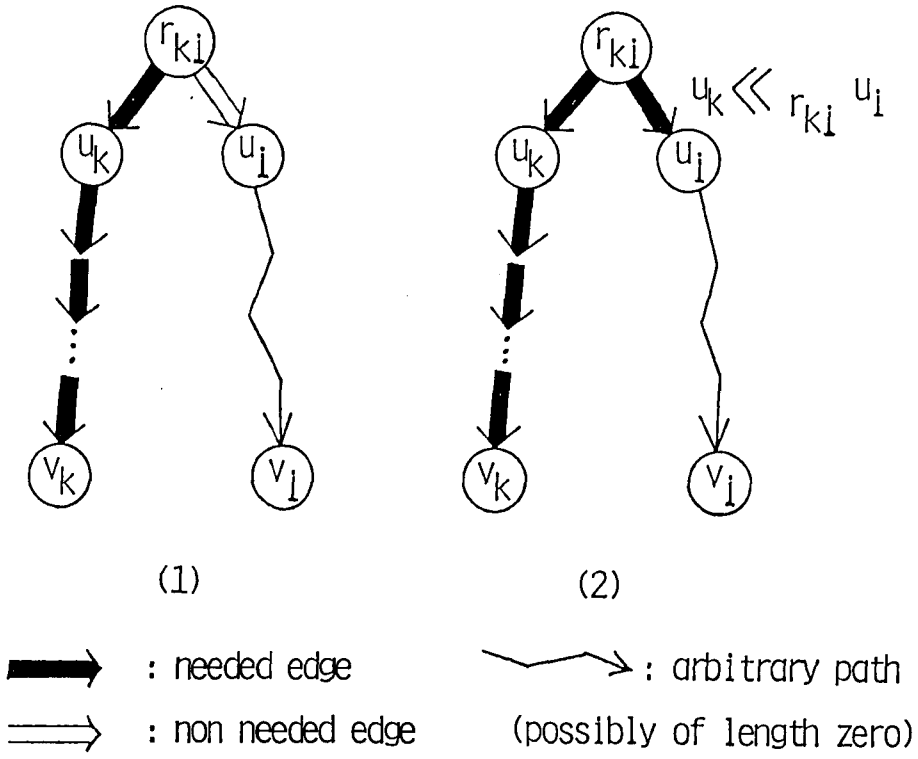


Figure B.3 Illustrations for (2a) and (2b) in Appendix B

APPENDIX C

A SUFFICIENT CONDITION FOR A SORT TO BE GLOBALIZABLE

Let $tree(right_g) = (V_g, E_g)$ $(g \in DEF(P))$. For a defined function g and a sort s, let CHANGE(g, s) denote the smallest subset of V_g satisfying the following:

(1) If the label of v in V_g is of sort s and is not a variable name, then v is in CHANGE(g, s).

(2) If the label of v in V_g is a defined function and CHANGE(label(v), s) is not empty, then v is in CHANGE(g,s).

In an object program $\underline{O}_P$, the instructions corresponding to the vertices in CHANGE(g, s) may cause storing of values of sort s on the run-time stack.

If the following both (1) and (2) hold, then a sort s is globalizable (with respect to EO(P) and SN(P)) in $\underline{O}_P$.

(1) Any primitive or defined function g satisfies (1a) or (1b).

(1a) If g is a primitive function of sort s, g has exactly one argument of sort s.

(1b) If g is either a primitive function of a sort other than s, or a defined function then g has at most one argument of sort s.

(2) For each defined function g and for each common vertex set $C = \{v_1, \ldots, v_m\}$ in V_g where $label(v_l)$ is of sort s $(1 \leq l \leq m)$, each pair v_i and v_j in C satisfies at least one of the following (a) to (d).

Let u_i and u_j denote the parents of v_i and v_j, respectively. Let r_{ij} be the lowest common ancestor of u_i and u_j, and let w_i and w_j be the sons of r_{ij} where w_i is an ancestor of u_i and w_j is an ancestor of u_j. (Note that u_i and u_j may be the same vertex.)

(a) $term(u_i) = term(u_j)$

(b) label(r_{ij}) is an IF function, and w_i and w_j are the second and third (or third and second) sons of r_{ij}, respectively. (Hence, the computation for u_i is exclusive of that for u_j, or vice versa.)

(c) Neither u_i nor u_j is in CHANGE(g, s). (The computation for u_i and u_j cause no storing of values of sort s on the run-time stack.)

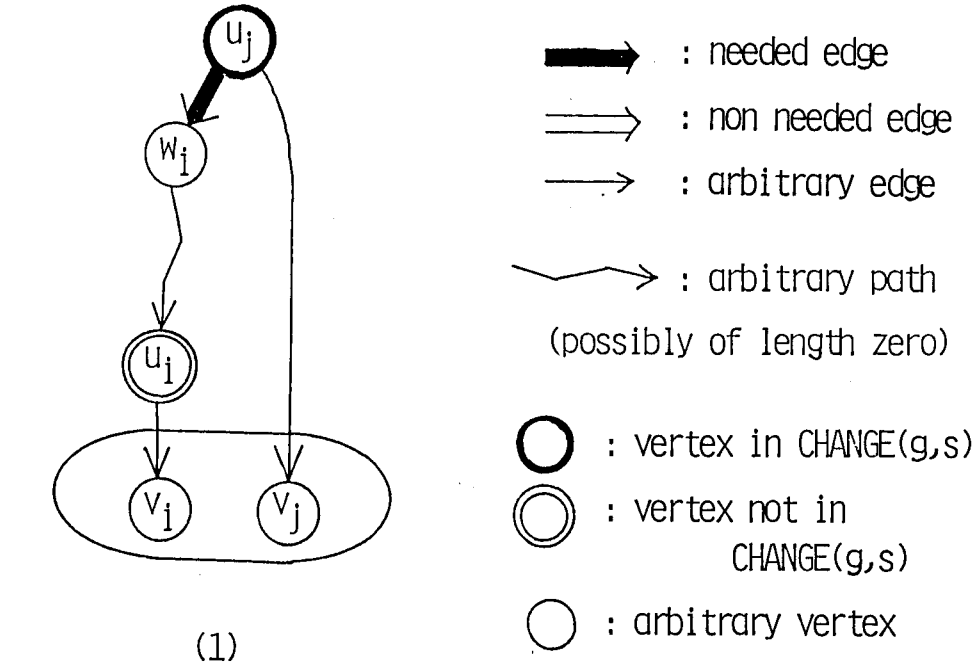
(d) u_i and u_j are distinct vertices, exactly one of u_i and u_j is in CHANGE(g, s), and all the following hold. (Here we assume that u_j is in CHANGE(g,s).) (see Figure C.1)

(d-1) u_i is not an ancestor of u_j.

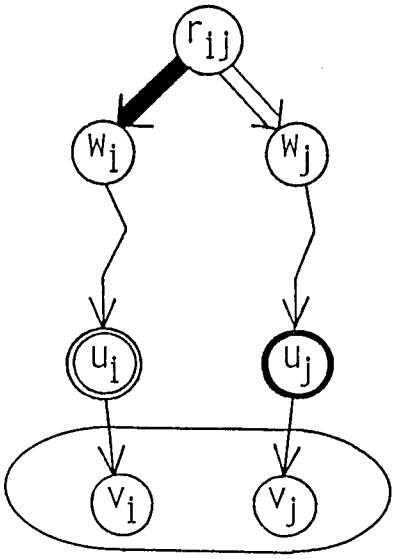
(d-2) If u_j is an ancestor of u_i (i.e., if $r_{ij}=u_j$), then w_i is a needed son of r_{ij}.

(d-3) If r_{ij} does not coincide with u_i or u_j, then either <1> w_i is a needed son of r_{ij} and w_j is not, or <2> both w_i and w_j are needed sons and w_i precedes w_j in needed vertex sequence $S_{r_{ij}}$ for r_{ij}.

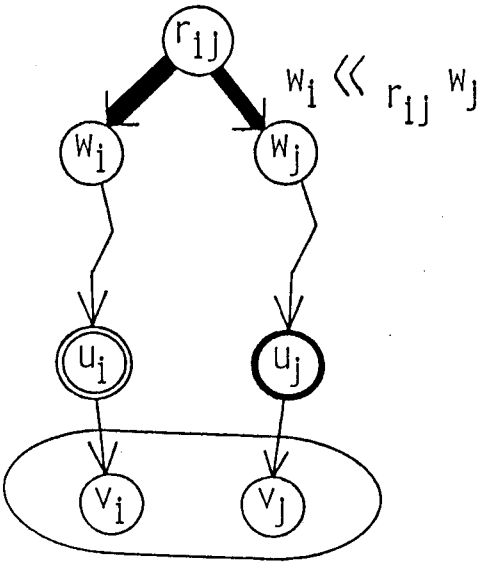
(The computation for u_j may cause storing a value of sort s on the run-time stack, and that for u_i may cause a reference of a value of v_i, which is of sort s. If the computations for both u_i and u_j are executed, then that for u_i must precede the computation for u_j.)



(1)



(2)



(3)

Figure C.1 Illustrations for (2d) in Appendix C

REFERENCES

[Aho and Ullman 77]

Aho, A. V., and Ullman, J. D., "Principles of Compiler Design", Addison-Wesley, 1977.

[Aho et al. 83]

Aho, A. V., Hopcroft, J. E., and Ullman, J. D., "Data Structures and Algorithms", Addison-Wesley, pp.66-67 and pp.78-82, 1983.

[Backus 78]

Backus, J., "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programming", Comm. ACM, Vol.21, No.8, pp.613-641 Aug. 1978.

[Berkling 76]

Berkling, K. J., "Reduction Languages for Reduction Machines", Interner Bericht ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, Sep. 1976.

[Bobrow and Wegbreit 73]

Bobrow, D. G., and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", Comm. ACM, Vol.16, No.10, pp.591-603, Oct. 1973.

[Burstall and Darlington 77]

Burstall, R. M., and Darlington, J., "A Translation System for Developing Recursive Programs", J.ACM, Vol.24, No.1, pp.46-67, Jan. 1977.

[Chiarini 80]

Chiarini, A., "On FP Languages Combination Forms", SIGPLAN Notices, Vol.15, No.9, pp.25-27, Nov. 1980.

[Friedman and Wise 76]

Friedman, D. P., and Wise, D. S., "CONS Should Not Evaluate Its Arguments", Proc. of 3rd international Colloquium on Automata, Languages and Programming, Edinburgh, pp.257-281, 1976.

[Henderson 80]

Henderson, P., "Functional Programming, Application and Implementation", Prentice-Hall, pp.218-223, 1980.

[Henderson and Morris 76]

Henderson, P., and Morris, J. H., "A Lazy Evaluator", Proc. of 3rd Symposium on Principles of Programming Languages, pp.95-103, Jan. 1976.

[Hosomi et al. 76]

Hosomi, T., Hagihara, K., Imase, M., Toyotaka, Y., and Okamoto, T., "A General Purpose Emulator Using a Microprocessor", Papers of Technical Group on Electric Computers, IECE Japan, EC76-10, May 1976 (in Japanese).

[Huet and Lévy 79]

Huet, G., and Lévy, J.,"Call by Need Computations in Nonambiguous Linear Term Rewriting Systems", INRIA Research Report, No.359, Aug. 1979.

[Inoue et al. 80]

Inoue, K., Ito, M., Sugiyama, Y., Taniguchi, K., Tanizawa, T., and Okamato, T., " An Implementation of a Functional Programming Language on a Microprogrammable Computer", Papers of Technical group on Electronic Computers, IECE Japan, EC80-36, Sep. 1980 (in Japanese).

[Inoue et al. 82a]

Inoue, K., Tanizawa, T., Taniguchi, K., and Okamoto, T., "Implementations of a Functional Programming Language FPL", Trans. of IECE Japan, Vol.J65-D, No.5, May 1982 (in Japanese).

[Inoue et al. 82b]

Inoue, K., Seki, H., Sugiyama, Y., and Kasami, T., "Code Optimization at Compilation of Functional Programming Language ASL-F Programs", Papers of Technical Group on Electronic Computers, IECE Japan, EC82-18, June 1982 (in Japanese).

[Inoue et al. 83]

Inoue, K., Seki, H., Sugiyama, Y., Taniguchi, K., and Kasami, T., "Functional Programming Language ASL-F Compiler", Proc. of 26-th National Convention of Information Processing Society of Japan, 3D-7, pp.13-14, March 1983 (in Japanese).

[Inoue et al. 84]

Inoue, K., Seki, H., Taniguchi, K., and Kasami, T., "Functional Programming Language ASL/F and Its Optimizing Compiler", Trans. of IECE Japan (to appear).

[Ishihara and Yonezawa 80]

Ishihara, H., and Yonezawa, N., "On an Implementation and Use of Backus' Functional Programming Language", Proc. of 21st National Convention of Information Processing Society of Japan, 1B-6, May 1980 (in Japanese).

[Keller 82]

Keller, R., "FEL: An Experimental Applicative Language", WG on Software Foundation of Information Processing Society of Japan, 3-11, Dec. 1982.

[Nakanishi et al.79]

Nakanishi, M., Sugiyama, Y., Taniguchi, K., Kasami, T., and Peterson, W., "A System supporting Program Design --Program Derivation from an Algebraic Specification--", 1979 National Convention Record on Information and System Section, IECE Japan, pp.402, Oct. 1979 (in Japanese).

[Magó 79]

Magó, G. A., "A Network of Microprocessors to Execute Reduction Language, Part 1 and 2", Int. J. Compt. & Inf. Sci., Vol.8, No.5 and No.6, pp.349-385 and pp.435-471 Oct. and Dec. 1979.

[Rosen 73]

Rosen, B. K., "Tree Manipulating Systems and Church-Rosser Theorems", J.ACM Vol.20, No.1, pp.160-187, Jan. 1973.

[Sasaki and Katayama 83]

Sasaki, H., and Katayama, T., "A Procedure to Check Globability of Attributes for Hierarchical Functional Programming HFP", Papers of Technical Group on Automata and Languages, IECE Japan, AL82-97, Feb. 1983 (in Japanese).

[Seki et al. 82]

Seki, H., Inoue, K., Sugiyama, Y. and Kasami, T., "Development of Functional Programming Language ASL-F Compiler", Proc. of 25-th National Convention of Information Processing Society of Japan, 1C-4, pp.321-322, Oct. 1982 (in Japanese).

[Seki et al. 84]

Seki, H., Inoue, K., Taniguchi, K., and Kasami, T., "Optimization of Functional Language Programs -Optimizing Compiler for ASL/F-", in preparation for submitting to Trans. of IECE Japan (in Japanese).

[Sugiyama et al. 82a]

Sugiyama, Y., Suzuki, I., Taniguchi, K., and Kasami, T., "Efficient Execution in a Class of Term Rewriting Systems", Trans. of IECE Japan, Vol.J65-D, No.7, July 1982 (in Japanese).

[Sugiyama et al. 82b]

Sugiyama, Y., Taniguchi, K., and Kasami, T., "Algebraic Specification Language ASL/1 - The Syntax and Its Semantics - ", Papers of Technical Group on Automata and Languages, IECE Japan, AL82-62, Nov. 1982 (in Japanese).

[Tanaka and Keller 82]

Tanaka, J., and Keller, R., "Code Optimization in a Functional Language", WG on Software Foundation of Information Processing Society of Japan, 3-10, Dec. 1982.

[Tanizawa et al. 80]

Tanizawa, T., Inoue, K., Ito, M., Sugiyama, Y., Taniguchi, K., and Okamoto, T., "A Method of Direct Execution of Programs in a Functional Programming Language", Proc. of 1980 National Convention of IECE Japan, 1471, 6-121, March 1980 (in Japanese).

[Wirth 76]

Wirth, N., "Algorithms + Data Structures = Programs", Prentice-Hall, pp.76-82, 1976.