数理解析研究所講究録
第 482 巻 1983 年 240-254

Treatment of Big Values in an Applicative Language HFP

--- Translation from By-Value Access to By-Update Access ---

Takuya Katayama

Department of Computer Science
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku
Tokyo 152, Japan

1. Introduction

A method for treating big data in an applicative language HFP by converting the way of accessing data from by-value to by-update. It is well-recognized that the applicative or functional style of programming is superior to the procedural one in its clarify, readability and verifiability. It usage in the practical world of computing, however, is blocked by the lack of ways for efficiently executing programs in the style and it comes mainly from the fact that, in applicative or functional languages, every data is accessed through its value and there is no way of naming and updating it through the name. Although this problem may not be conspicuous when data is small in size, it is essential in dealing with big data such as files and databases.

Although there are cases where we can avoid computation involving big values by delaying the operations on the values until they are really needed, i.e., by lazy evaluation technique [2], this technique is not almighty and in many cases we have to do computations about big values, such as updating files, large list structures or arrays. This paper proposes a method of treating big data by converting by-value access to by-update access, which is used in the imlpementation of an applicative language HFP.

HFP is an applicative language which admits hierarchical and applicative programming [3] and is based on the attribute grammar of Knuth [5]. It also has close relationship to Prolog. In the following we first introduce HFP and the discuss its implementation which solve the big data problem by using a simple file processing program.

2. HFP

HFP is an applicative language which supports hierarchical program design and verification. In HFP, a program is considered a module with inputs and outputs which are called attributes of the module. When a task to be performed by the module is complex, it is decomposed into submodules which perform corresponding subtasks and this fact is expressed by a set of equations which hold among attributes of the modules involved in the decomposition. Module decomposition proceeds until no more decompositions are possible.

In short, HFP is an extension of attribute grammar to write programs for general problems which are not necessarily language processing ones for which attribute grammar has been originaly invented.

2.1 Formalism

HFP comprises (1) module, (2) module decomposition, and (3) equation.

(1) Module

Module is a black box with inputs and outputs termed attributes. The function of module is to transform its inputs to outputs. A module M with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$ is denoted diagramatically by

$$\boxed{M}\ \downarrow x_1, \ldots, \downarrow x_n, \uparrow y_1, \ldots, \uparrow y_m$$

and we write

$$IN[M] = \{x_1, \ldots, x_n\},\ OUT[M] = \{y_1, \ldots, y_m\}.$$

Module is completely specified by its input-output relationship and no side effect is permitted.

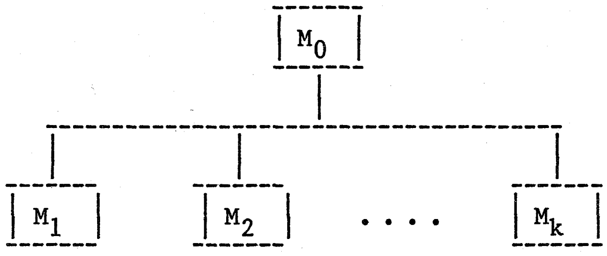
There are special modules: the initial module M_{init} and null module. The initial module corresponds to the main program and whose input-output relationship is what is to be realized by the HFP program under consideration. The null module, denoted by **null**, is used to terminate module decomposition.

(2) Module Decomposition

When a module M_0 has to perform a complex function and is decomposed into modules M_1, M_2, ..., M_k, we express this fact by

$$M_0 \rightarrow M_1\ M_2\ \dots\ M_k$$

or



Decomposition process is terminated by applying a special decomposition with null module as its right side, which is denoted by

$$M \rightarrow \textbf{null} \quad \text{or simply} \quad \boxed{M} \; .$$

It is usually the case that a module is decomposed in several ways. We attach the <u>decomposition condition</u> C to each decomposition d and write

$$d\!: M_0 \rightarrow M_1\ M_2\ \dots\ M_k \ \textbf{when}\ C.$$

The condition C is, in general, specified in terms of attributes, or more accurately, attribute occurrences of M_0, M_1, ..., M_k and it directs when this decomposition may apply. In this paper we only consider determinitic HFP in which (1) this condition C is specified in terms of input attributes of M_0 and (2) there are no two decompositions with the same left side module whose decomposition conditions may become simultaneously true.

(3) Equations

We associate a set of equations to each decomposition d for specifying how the task performed by the main module M_0 is decomposed into subtasks performed by submodules M_1, ..., M_k. This is done by writing equations for (1) what data should be sent to submodules as their inputs and (2) how to combine their computation results to obtain outputs of the main module. Equations are of the form

$$v = F_v (v_1, \ldots, v_t)$$

where (1) $v, v_1, \ldots, v_m$ are attribute occurrences in d. We use the notation $M_i.a$ to denote an occurrence of attribute a of M_i in the decomposition d, (2) $v = M_i.a$ for $a \in IN[M_i]$ $(i=1,\ldots,k)$ or $v = M_0.a$ for $a \in OUT[M_0]$, (3) F_v is an attribute function for computing v from other attribute occurrences $v_1,\ldots,v_t$.

[Computation in HFP]

Given the values of input attributes of the initial module, computation in HFP is carried out in the following steps.

(1) Starting from the initial module, apply decompositions successively until terminated by null module. A tree which is virtually constructed in the above process and whose nodes are module occurrences of the decompositions applied is called a computation tree.

(2) Evaluate the values of attributes of the modules in the computation tree according to the equations associated with decompositions applied.

(3) The values of outputs of the initial module give the results of the computation activated by its input values.

2.2 Example : 91HFP

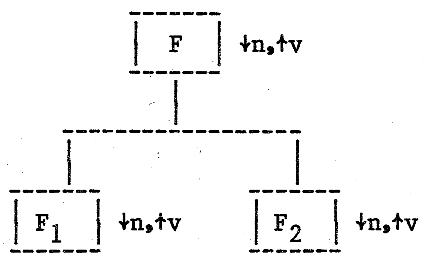
Let us consider a simple example. Although this example is too simple to verify the power of HFP it will be some aid in understanding it.

Let F be the McCarthy's 91 function. This function is defined by

F(n) = **if** n≤100 **then** F(F(n+11)) **else** n-10.

We introduce the module 'F' and associate input n and output v with it. The module F is decomposed into two ways as shown below with equations about its attributes n and v. (Note that '=' in the **with** clauses means 'equal to' and not a symbol for assignment.)

decomposition 1.



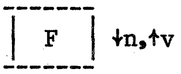
with $F_1.n = F.n+11$

$F_2.n = F_1.v$

$F.v = F_2.v$

when $F.n \leq 100$

decomposition 2.

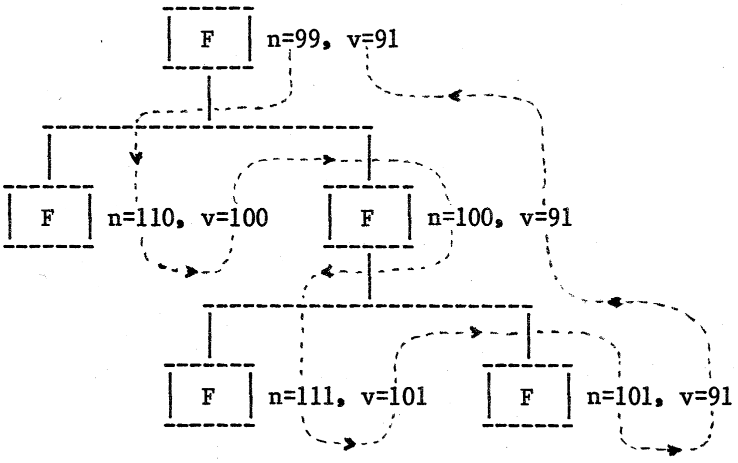


with $F.v = F.n-10$

when $F.n > 100$

In decomposition 1, module name F is indexed to distinguish different occurrences and no submodule exists in decomposition 2 to terminate decomposition. Attributes prefixed by '↓' and '↑' are inputs and outputs respectively. The **when** clause specifies the condition on which the decomposition may apply.

The tree below shows a computation tree for input n=99, where dotted lines indicate data dependency.



2.3 Comparison to Backus'FP and Prolog

[Backus'FP] Applicability of HFP consists in the facts: (1) relationship between attribute occurrences in every decomposition is specified by a set of equations which state that they are computed functionally from other attribute occurrences and (2) every module can be viewed as defining a set of functions which realize input-output relationship of the module. Difference between the Backus' functional programming (BFP) and ours is that BFP defines functions explicitly from primitive ones in terms of functional calculus, whereas HFP forcuses on modules and their decompositions and the form of the functions realized by the module is scattered in the set of equations.

[Prolog] Nondeterministic HFP is very closely related to Prolog in which every of variable is specialized for either input or output. Let d be a decomposition in HFP

$$d : M \rightarrow M_1\ M_2 \ldots M_k \textbf{ when } C.$$

Roughly speaking, this decomposition is equivalent to the following Horn clause in prolog, where [M], in general, means M whose attribute occurrences are substituted by the right hand side of the equations for defining them.

$$[M] \leftarrow [M_1],[M_2],\ldots,[M_k],[C]$$

We consider that this fact does not degrade HFP, because distinction between input and output is essential in many cases and this makes it possible to utilize data dependency analysis for implementing HFP far more efficiently than otherwise and to verify HFP program by the method similar to the one for attribute grammar [3].

3. Basic Implementation Technique for HFP

The basic idea for implementing HFP is to associate procedures $P_{M,y}$ to each module M and its output attribute y, and translate the given HFP into a procedural program which consists of these procedures. Here we only skech the technique and for the detail please refer to [3]. We only consider absolutely noncircular HFPs.

Let d_i's(i=1,2,...) be decompositions with left side mode M and decomposition condition C_i. Then the procedure $P_{M,y}$ is of the form

```
procedure $P_{M,y}(x_1,...,x_n;y)$
    if $C_1$ then $H_1$ else
    if $C_2$ then $H_2$ else
          .
          .
end
```

where x_1,..., x_n are input attributes of M on which y is dependent. They are decided by analyzing data dependency DG[M] in the attributes of M. H_i's are sequences of assignment or procedure call statements to calculate the value of y from those of x_i's ,whose forms are determined from data dependencies DG[d_i] among attribute occurrences of d_i's and their associated equations. The principle for constructing H_i is stated in the following way. First we prepare variables for attribute occurrences of d_i. H_i is a sequence of statements to assign values to these variables when the decomposition condition C_i holds. If a variable corresponds to the output attribute y of the main module M or to an input attribute of a submodule M' then the value assigned is computed from the defining equation for it. If it correspons to an output attribute w of a submodule M', the value is obtained by calling a procedure $P_{M',w}$ which is associated with M' and w. These statements are listed in such order as determined by topologically sorting the dependency relation DG[d_i] among attribute occurrences of d_i so that the value of an attribute occurtence v is determined after valuation of attribute occurrences on which v is dependent has been completed.

The next program is obtained from the 91 HFP.

```
program  PROG91
     procedure F(n;v)
          if n > 100 then v ← n-10
                     else n1 ← n+11 ; call F(n1,v1) ; n2 ← v1 ;
                          call F(n2,v2) ; v ← v2
    end
    input(n) ; call F(n,v) ; output(v)
end
```

Of course, this program can be made simpler by folding assignment statements.

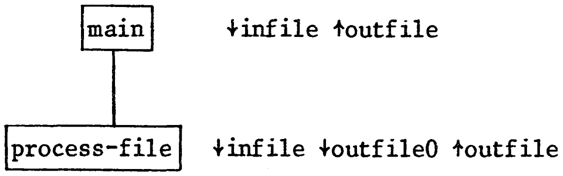
4. Implementing Access to Big Values in HFP

4.1 An Example : A Simple File Processing HFP Program

Let us consider the following simple file processing HFP program which, given an input file of records R_1, R_2,..., R_k arranged in this order, produces an output file of records $F(R_1)$, $F(R_2)$,..., $F(R_k)$, where F is a function operated on input records.

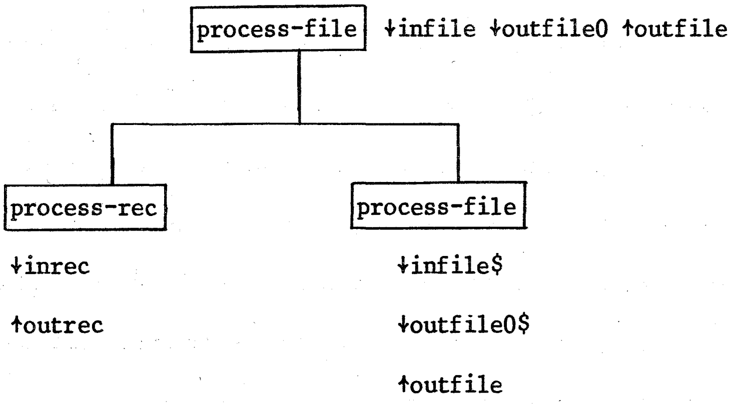
```
attribute infile, outfile, outfile0 : file of record
          inrec, outrec : record
module main has ↓infile ↑outfile
       process-file has ↓infile ↓outfile0 ↑outfile
       process-rec has ↓inrec ↑outrec
module decomposition
```

(1)



```
          with  outfile0 = emptyfile
```

(2)



```
with  inrec = 1st(infile)
      infile$ = rest(infile)
      outfile0$ = appendr(outfile0, outrec)
when not eof(infile)
```

(3)

process-file ↓infile ↓outfile0 ↑outfile

```
with outfile = outfile0
when eof(infile)
```

(4)

process-rec ↑inrec ↓outrec

```
with output = F(inrec)
when always
```

[1] The **attribute** clause defines data types of attributes. **file** is a data type with operations

emptyfile	:	→ file
1st	: file	→ record
rest	: file	→ file
appendr	: file x record	→ file
eof	: file	→ boolean

, where 1st(f) means a first record of a file f, rest(f) is a file obtained from f by deleting its first record , appendr(f,r) is a file resulted from writing the record r at the end of f, and eof(f) is true iff f is emptyfile. Details of the data type 'record' is irrelevant here. [2] **module** clause declares module with their attributes. Input attributes are prefexed by ↓ and output attributes by ↑. [3] **with** clause specifies equations associated with a decomposition and **when** clause defines its decomposition condition. [4] Note that notation for attribute occurrence is different from what is given in section 2. In this HFP program, for the sake of clarity, attribute occurrence is not prefixed by module name and instead positional notation is used. Also note that when the same attribute name appears more that once in a decomposition this means that equations for copying the attribute values are omitted.

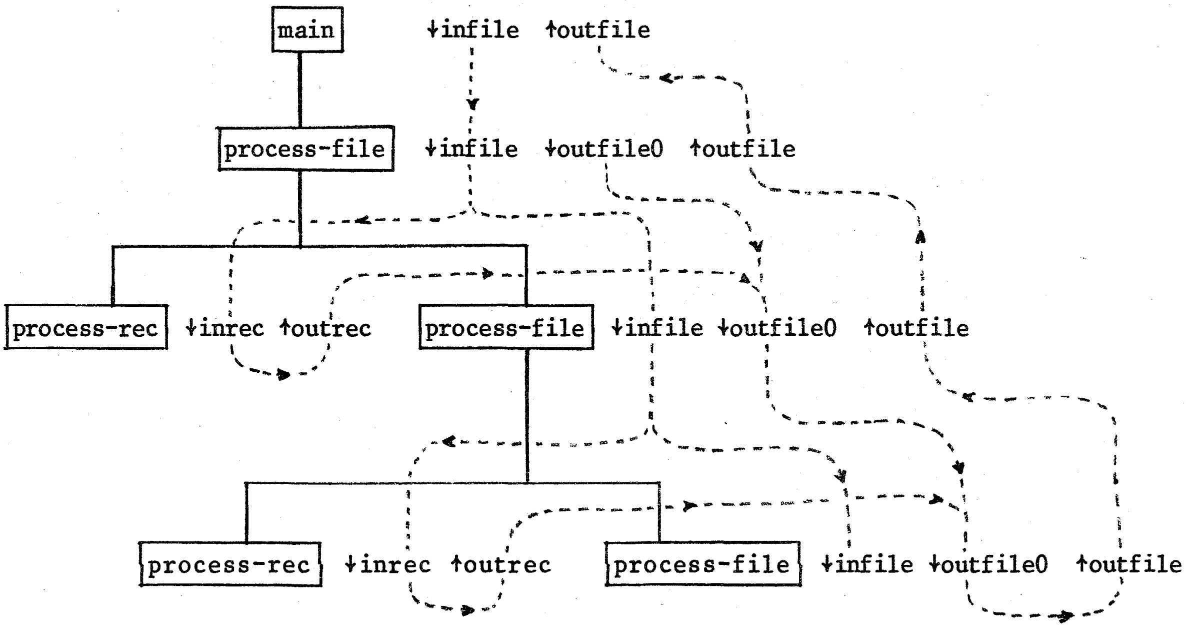
Suppose we apply the basic implementation technique to this HFP, we get the following procedural type program

```
program main(infile; outfile)
    procedure process-file(infile, outfile0; outfile)
         if not eof(infile) then
            inrec := 1st(infile)
            infile$ := rest(infile)
            call process-rec(inrec; outrec)
            outfile0$ := appendr(outfile0, outrec)
            call process-file(infile$, outfile0$; outfile)
         else
            outfile := outfile0
    end
    procedure process-rec(inrec; outrec)
         outrec := F(inrec)
    end
    outfile0 := emptyfile
    call process-file(infile, outfile0; outfile)
end
```

This program is, of course, unacceptable because big values of files are copied to variables and passed to and from procedures.

[From By-Value Access To By-Update Access]

Let's consider a computation tree for this HFP program, which illustrates how the input file 'infile' of the main module is transformed into the output file 'outfile'.



From this tree, we can see the following (1) and (2) hold.

(1) To all occurrences of the attribute 'infile' can be assigned a single global variable instead of a stack which is realized by activation records of the procedure 'process-file' in the above program. This is because after 'infile' is referred at a level to compute values of 'inrec' and 'infile' for the next level it is not referred any more.

(2) So are 'outfile' and 'outfile0'. Furthermore, we can assign the identical global variable to these attributes just by the similar reason.

Taking these facts into consideration, the generated program for our file processing HFP becomes as follows. The procedure 'process-file' has no parameters in this version.

```
program main(infile, outfile)
        procedure process-file
             if not eof(infile) then
                inrec := 1st(infile)
                infile := rest(infile)
                call process-rec(inrec; outrec)
                outfile := appendr(outfile, outrec)
                call process-file
        end
        procedure process-rec(inrec; outrec)
             outrec := F(inrec)
        end
        outfile := emptyfile
        call process-file
    end
```

Note that we omotted a useless assignment 'outfile := outfile' and changed the **if-then-else** statement to **if-then** statement.

Now, what shoule be done next is to translate 'by-value' file access to 'by-update' access. This is accomplished by the help of translation axioms given below.

Axiom 1

```
inrec := 1st(infile)
infile   := rest(infile)
       ---→ read(infile, inrec)
```

Axiom 2

```
outfile := appendr(outfile, outrec)
       ---→ write(outfile, outrec)
```

Axiom 3

```
outfile := emptyfile
       ---→ rewrite(outfile)
```

The read, write, and rewrite statements are such that (1) read(f,r) transfers the record of f at the cursor position to r and move the cursor right, (2) write(f,r) writes the data in r at the cursor position and move it right and (3) rewrite(f) initializes f.

After applying these axioms to the generated program, i.e., replacing texts which matches the right side of any axiom by its left side, we have the following program in which files are accessed through the usual updating operations 'read' and 'write'. We also changed tail recursion to iteration.

```
program main(infile, outfile)
    rewrite(outfile)
    while not eof(infile) do
       read(infile, inrec)
       outrec := F(inrec)
       write(outfile, outrec)
end
```

4.2 General Scheme for Translation

Translating a given HFP program into a procedural type program in which big values are accessed through updating is performed in the following steps. Input to the translator is the given HFP program and a database of axioms which state what pattern of by-value data access statements can be converted to by-update access statements.

1. Calculate data dependency relations DG[d] and DG[M] among (1) attribute occurrences of each decomposition d and (2) attributes of each module M rspectively.

2. Determine groups of attributes to which single global variables can be assigned. These groups can be determined from dependency relation DG[d] obtained in step 1. A simple and useful criterion that a group of attributes satisfies the above condition is that every deoendency relation DG[d] does not contain 'branching' with respect to the attributes. More specifically, this is stated in the following way. Let DG[d]=(V,E), where V is a set of attribute occurrences of d and E={(v1,v2)|there is an attribute function f_{v2} such that $v2=f_v(\dots,v1,\dots)$}. If there is no tripple (v1,v2,v3) such that both of (v1,v2) and (v1,v3) are in E and v1, v2 and v3 are among $M_1.a_1,\dots,M_k.a_k$, then

this criterion assures that these attributes $a_1,...,a_k$ can be globalized. Distinct local variables are assigned to other attribute occurrences.

3. For each module M and its output attribute y, construct a procedure $P_{M,y}$ by the method sketched in section 3.

4. Inspect the body of $P_{M,y}$ and find a group of statements for big data access. If it matches to the right side of some conversion axiom, replace the group by its left side which is a sequence of statements for by-update data access.

5. Apply, if possible, other optimization techniques such as folding of assigment statements and recursion elimination.

5. Concluding Remarks.

We have proposed a method of treating big values which is used in an applicative language HFP. The underlying principle is to convert by-value data access to by-update data cccess with the aid of convertion axiom database. An exampe is given to support this method. This principle may be applied to other applicative systems.

References

1. Backus, J. Can programming Be Liberated from the von Neumamm Style ? A Functional Style and Its Algebra. Comm. ACM, 21,8(Aug.1978), 613-641

2. Henderson,,P. & Morris,J.H. A Lazy Evaluator. Proc. of 3rd ACM Smp. on Principle of Programming Languages (1976)

3. Katayama, T. HFP: A Hierarchical and Functional Programming Based on Attribue Grammar,Proc. 5th Int. Conf. On Software Engineering (1981)

4. Katayama,T., and Hoshino, Y. Verification of Attribute Grammars. Proc. 8th ACM Symposium on Principles of Programming Languages (1981)

5. Kunth, D.E. Semantics of context-free languages Math. Syst. Theoty J.2(1968),127-145