ソフトウェア基礎論 3～10
（1982. 12. 10）

CODE OPTIMIZATION IN A FUNCTIONAL LANGUAGE

Jiro Tanaka and Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

Improvement of the execution efficiency of a functional language is discussed in terms of optimization at an intermediate "assembly code" level. This assembly form is itself a functional language, in the sense that it is an encoding of graphical representations of functions in the program. The optimizations discussed assume a demand–driven block–oriented evaluation model. Two methods, demand preevaluation and block restructuring, are proposed preliminary techniques.

1. Significance of Code Optimization

Although functional languages have been extolled as having a number of benefits, their execution efficiency must be improved before their use will become widespread. Our claim is that the inefficiency is not a defect of the languages themselves but rather of current implementations. We discuss two optimization techniques, the aim of which is efficiency improvement.

2. Function–Equation Language and its Current Implementation

As an example source language, we use Function–Equation Language (FEL) [Keller 82] in which a program is expressed by a set of equations defining functions or other data objects. In the current implementation, the FEL translator first compiles an FEL program to an "assembly form", an encoding of a graphical representation of the program. The assembly code can directly express cyclic graph structures and sharing of common subexpressions [Keller 80].

The "assembly form" of a program is structured as a set of "blocks", with each block corresponding to one non–primitive function. The FEL translator may create anonymous functions in the case of lambda expressions, system functions, or "multi–tiered" (Curried) function definitions. As an example of the latter, the definition f|x|y = g:[x,y], where both | and : represent application (| is left–associative and : is right–associative), is compiled as f|x ={RESULT $LAMBDA, $LAMBDA|x = g:[x,y] }, where $LAMBDA is a compiler–generated name. The braces indicate name scoping; all names used within are either defined at the immediate nesting level of the braces or are imported from outside.

The current implementation of FEL does not employ association lists for binding names to imports or actual parameters. Instead, the values of names brought in "by value–need" at most once from either the surrounding context or the parameters.

Program structures are dynamically allocated and deallocated in units of "blocks". A block can be thought as a user–defined "macro–combinator", since it is expressed without using any variables explicitly. Graph–based reduction is done by the AMPS (Applicative Multi–Processing System) evaluator [Keller 79].

3. Language Features of FEL

FEL is currently implemented by a graph reduction evaluation model. A reduction model consumes its program structure in the process of computation. In the competing class of token–flow models [Davis 82], iteration operators can be implemented by circulating data tokens in a code block. In a reduction model, iteration operators are usually implemented using recursion because code blocks are consumed by evaluation. The judicious introduction of token–flow evaluation within the reduction evaluator is another optimization currently under investigation.

A second aspect of the evaluator is demand–driven computation. Demand–driven computation avoids unnecessary evaluation by evaluating only the demanded parts of a program. This style of evaluation is necessary for supporting conceptually–infinite objects such as infinite streams of data objects, which make certain programs simpler and cleaner.

Another feature of FEL, as well as other functional languages, is the use of higher order functions, i.e. functions which use other functions as arguments or produce functions as results. The current macro–combinator implementation appears to be a fairly efficient technique for supporting such higher order functions, since it does not require saving association lists.

4. Some Optimization Problems

Optimization methods can be classified by two main objectives: time optimization and space optimization. Space optimization can further be divided into static and dynamic space optimization. Static code optimization optimizes the size of the object code, whereas dynamic space optimization considers the code and data space at execution time. *Tail recursion* is an example of a typical dynamic space optimization technique.

In many cases, there is a trade–off between time and space factors. Time optimization is often accompanied by an increase in the code space. The space–time product, which is the integral of space used over the total execution time, is an effective measure of the degree of the optimization in such cases, as discussed in a later section.

The current FEL translator makes some steps toward optimization. It locates opportunities for tail recursion and sharing of common subexpression. For example, function invocation instances which have the same arguments are evaluated only once and the returned value is shared by all their occurrences.

5. Related work

Much work has been done in the optimization of conventional languages [Allen 71, Aho 77, Schaefer 73, Muchnick 81]. Optimization techniques can be classified as local or global. Local optimization is only concerned with a small portion of the program. This includes "constant folding" or "redundant subexpression elimination". Global optimization considers the program as a whole. It requires global control and data flow analysis. "Code motion" and "strength reduction" are representative techniques of global optimizations. All of the above techniques have more–or–less direct counterparts in functional language implementations.

6. The Impact of the Evaluation Model on Optimization

When the evaluation steps of a program in a *reduction* model are analyzed, the program is seen to expand and shrink in the process of generating output and perhaps converging to a final value, i.e. its *fixpoint*. Often there are intermediate data structures which are much bigger than the initial or final code/data structures. Therefore a critical problem in space optimization is the control of dynamic space needed to execute a program.

In a certain sense, a reduction model is much simpler than a token–flow model [Davis 82]. In a token–flow model, data tokens have to be implemented apart from the code structure, but in the case of a reduction model, data can be handled as nodes of the program structure. Structured data types are therefore handled as program structures and there is no need for supporting structured data apart from code structure.

On the other hand, a reduction model has disadvantages. In a reduction model, copies of code blocks are destroyed by evaluation. It is difficult to reuse the same program structure. For example, in a token–flow computation, iteration can be implemented by circulating data tokens within the same code structure, whereas iteration is usually implemented using recursion in a reduction model. Recursive forms may give maximum concurrency, but they often waste space. In a reduction model, a stream must be expressed as a list–structure [Keller 81]. There must be made as many copies of a program structure as there are stream elements.

One possible solution for space optimization is to combine a token–flow model and a reduction model. In a token–flow model, data tokens flows in the code–structure. It is possible for the same code structure to be reused many times. By combining a token–flow model with a reduction model, it becomes possible to handle streams as token streams flowing inside of code blocks in a pipelined manner. The use of high level primitives allows such a combination. This aspect of optimization is discussed in [Tanaka 82].

Generally speaking, demand–driven computation has more space efficiency but less concurrency than data–driven computation, because an expression is not computed until it is requested. In FEL, we can control concurrency by explicitly introducing concurrency control operators, such as SEQ or PAR [Keller 81].

7. Demand Optimization

7.1. Pre–evaluation of Demands

One aspect of demand–driven computation which may slow things down is that an expression is not computed until its value is requested. One way to speed up the execution time of a program is to preevaluate demand flow at compilation time. Currently, all evaluation of demand flow in FEL is done at execution time. Fortunately, in some cases, e.g. that of strict operators, all arguments can be demanded in advance, as they will always be needed. Hence some demand propagation can be done at compile time, rather than execution time. Preevaluation of demand flow can be done by introducing *demand* operators explicitly. Demand operators serve to indicate where to propagate demand. They transmit demand directly to the specified operators instead of propagating demand layer–by–layer.

Demand preevaluation primarily saves execution time, but its advantages are not limited to the time factor. Normally, a user–defined function is expanded to its definition when its result is demanded, but before the arguments have been evaluated. However, function expansion can be delayed until the arguments have been evaluated by preevaluating demand and transmitting it to the arguments of the function directly. Demand preevaluation also gives a guide for FEL's block restructuring. By restructuring the blocks of the FEL assembly code, a program will become more space–time efficient. This technique will be discussed in a later section.

In demand–driven computation, an expression is evaluated from the outermost operators at execution time. If the outermost operator is reducible, it is replaced with its value. If it is not, it further transmits the demand to the inner operators. Demand flow can be preevaluated at the assembly code level. An example of demand preevaluation is shown in figures 7–1 and 7–2. In this example, f1, f2, f3 and f4 represents primitive strict operators. Small circles and arrows in the figure express demand–transmitters. They work as a bypass, and directly transmit the demand to the operators to which they point. Nodes marked with * are predemanded in the code template. These wait for the notify from their arguments, apply reduction rules, and then notify the parent node.

```
{
Foo: [x,y,z,w] = if x then f1:f2:y else f3:f4: [z,w]

result print:Foo: [false,2,3,4]
}
```

Figure 7–1: An FEL program example

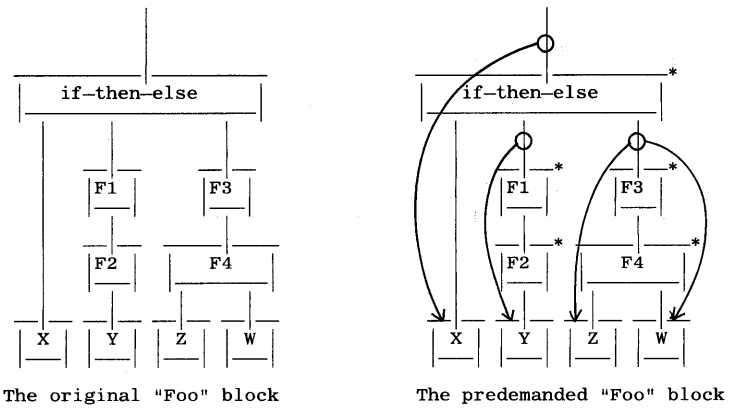


Figure 7–2: Demand preevaluation of a conditional expression

7.2. FEL Assembly Form

Figure 7–3 shows the assembly form for the FEL program of figure 7–1. Two blocks, MAIN and FOO, are shown, each block corresponding to a function–equation at the source code level. The block header consists of global–id, result arc, argument–id and import–id. Each line of text inside of a block represents a node by specifying its output arc number, function, and input arc number.

```
DEF FOO 9; $TEMP1;
1 SELECTC 4 $TEMP1
2 SELECTC 3 $TEMP1
3 SELECTC 2 $TEMP1
4 SELECTC 1 $TEMP1
5 F2 3
6 F1 5
7 F4 2 1
8 F3 7
9 COND 4 6 8
ENDDEF
```

```
DEF MAIN 7;;
1 FALSE
2 NUMB 2
3 NUMB 3
4 NUMB 4
5 CONS 1 2 3 4
6 APPLY 8 5
7 PRINT 6
8 CLOSURE FOO
ENDDEF
```

Figure 7–3: Assembly form of FEL

7.3. Implementation of Demand Preevaluation

In the implementation of demand preevaluation, demand–transmitters are expressed by "demand" operators. The "demand" operator takes n arguments. If demand:[x,y,z,...] is demanded, it transmits demand to the operators which are pointed to by y,z, ... and does not propagate the demand to the first argument x. An example of preevaluated assembly code is shown in figure 7–4. In demand–driven computation, a node must be demanded first before it is reduced to its output value. When a node is demanded, notifiers are set to indicate which operators are to be notified when the result is computed. In demand preevaluation, predemanded nodes notifiers are preset. We indicate their presence in the code by numbers surrounded by ( ).

In this example, block Foo is evaluated in the order of 9 –> 4 => 9 –> 8 –> 7 –> 2,1 => 7 => 8 => 9, assuming that the predicate part of Foo is false. Here –> shows demand propagation and => shows notification. In contrast, in the preevaluated version, the expressions are evaluated in the order of 12 –> 4 => 9 –> 11 –> 2,1 => 7 => 8 => 11 => 9.

```
DEF FOO 9; $TEMP1;
1 SELECTC 4 $TEMP1
2 SELECTC 3 $TEMP1
3 SELECTC 2 $TEMP1
4 SELECTC 1 $TEMP1
5 F2 3
6 F1 5
7 F4 2 1
8 F3 7
9 COND 4 6 8
ENDDEF
```

The original assembly code

```
DEF FOO 12; $TEMP1;
1 SELECTC 4 $TEMP1
2 SELECTC 3 $TEMP1
3 SELECTC 2 $TEMP1
4 SELECTC 1 $TEMP1
5 F2 3 (6)
6 F1 5 (10)
7 F4 2 1 (8)
8 F3 7 (11)
9 COND 4 10 11 (12)
10 DEMAND 6 3
11 DEMAND 8 2 1
12 DEMAND 9 4
ENDDEF
```

The predemanded assembly code

Figure 7–4: Predemanded assembly code

7.4. Fixpoint Demand Flow Analysis at the Inter–block Level

Demand prevaluation can be extended to the inter–block level. A simple example is shown below.

```
{
add2:x = add1:add1:x

add1:x = add:[1,x]

result print:add2:4
}.
```

Add2 calls add1. Pre–demands of add1 must be evaluated first. Then pre–demands of add2 are evaluated using the information of add1. In this example, we first know add1 is a strict operator. Then we know add2 is also strict.

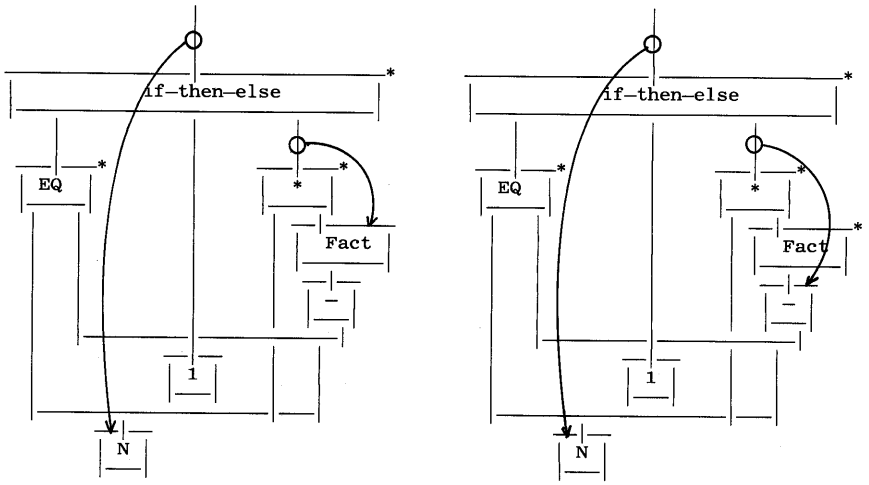
It is obviously possible to predemand a function at the inter–block level if a program has such a hierarchical structure. A call–graph which expresses a caller–callee relation can be constructed for a given FEL program. If this call–graph has a tree structure, user–defined functions can be preevaluated from leaf nodes to the root node.

If the call–graph has a loop, it means user–defined functions are called recursively or mutually recursively. The example below shows the recursive case.

```
{
Fact:n= if n=1 then 1 else n * Fact:(n–1)

result print:Fact:2
}
```

On the surface, Fact cannot be preevaluated because it is called recursively and it is *not* known whether Fact demands its argument or not. But demand can be preevaluated assuming that Fact does not demand its argument at first. Then it becomes known that Fact demands the argument n. Using this information, Fact is preevaluated again and again until it arrives at the least *predemand fixpoint.* In this example, this occurs within two steps. It is easily proved that there exists a unique least predemand fixpoint and that it can be computed in a finite number of steps. This fixpoint demand flow analysis is quite effective for recursive or mutually recursive FEL programs.



The first preevaluation

The second preevaluation

Figure 7–5: Demand Preevaluation of Fact

8. Block restructuring

Usually functions are defined to enhance the readability of a program at the source code level. Thus their corresponding block structures are not necessarily optimal for program execution at the assembly code level. If the block size is small, much execution time will be spent for the linkage of functions. Instead, if the block size is too large, it may lead to the waste of space, and contribute unnecessarily to the space–time product. Block restructuring aims at a more space–time efficient execution.

8.1. Block Splitting and Block Merging

The simplest example of block splitting concerns the evaluation of if–then–else operators. In demand–driven computation, the predicate argument of the if–then–else operator is evaluated first. Then the second or third argument is computed, depending upon the result of the predicate. Therefore by splitting the second and third arguments into separate blocks, space efficiency will be increased. Only one argument block will be created at execution time. In addition, tail recursion can be used if the block begins with an if–then–else operator. Garbage collection can take place with the original block when either argument block is called.

Conversely, if a block A is sure to demand its argument block B, block B probably should be merged with block A. Here, we make the qualifying assumption (not always valid in the current evaluator) that no long–term storage is allocated within blocks A and B. Block merging can be done to expand a function to its definition. Block merging primarily saves block loading time. The relation of block merging and splitting is shown in figure 8–1.

8.2. A Cost Function for Block Restructuring

For simplicity, we assume number of processors is infinite and the communication time between processors is negligible. The following factors can be considered as cost functions:

- t(p): time factor of process p. t(p) = te(p) + tl(p).
- te(p): execution time of process p.
- tl(p): loading and connecting time of a block. tl can be estimated as tl(p) = l1 + l2*length(p), where l1 is constant factor and l2*length(p) is the factor which is proportional to the length of a block.
- s(p): space factor for block p. s(p) = sk + s'(p).
- sk: sk expresses the block overhead. Block overhead is assumed to be block independent and always equal to some constant.
- s'(p): s' expresses the space factor for pure coding. s'(p) = s1*length(p).

An simple example for block restructuring:

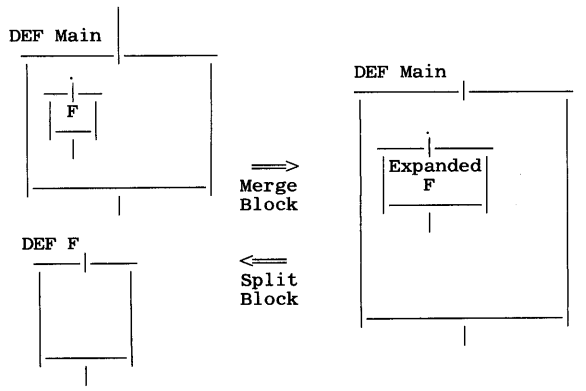


Figure 8–1: Block restructuring: block merging and splitting

program: if p then A else B

case 1: The whole program is packed to one block.

time factor: t(main) = te(main) + tl(main)
= te(main) + l1 + l2*length(main)
space factor: s(main) = sk + s'(main)
= sk + s1*length(main)

case 2: The if–then–else and predicate part form block main'.
The "then" part and "else" part are separated to the other blocks.

time factor
= t(main') + or (t(A), t(B))
= te(main') + tl(main') + or(te(A)+tl(A), te(B)+tl(B))
space factor
= s(main') + or(s(A), s(B))
= 2sk + s'(main') + or(s'(A), s'(B))

relation between two:

te(main) = te(main') + or(te(A), te(B))
s'(main) = s'(main') + s'(A) + s'(B)

It is not known which argument of "or" is needed. A worst case analysis can be made by using "max" instead of "or". By comparing the space–time product of these two cases, an estimate can be made as to whether or not the block restructuring is good. This kind of estimate is also applicable to other cases of block restructuring.

9. Summary and Future Research Plan

Two optimization methods, i.e. demand preevaluation and block restructuring, has been introduced for the optimization of FEL. The demand preevaluation algorithm has sketched. As for block restructuring, in most cases there are trade–offs between the space and time factors. How to harmonize these two factors is the important problem. The cost function approach mentioned here must be expanded and investigated further. These aspects are being considered in the planning of a FEL optimizer.

10. References

[Aho 77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison–Wesley, 1977.

[Allen 71] Frances E. Allen and John Cocke. A Catalogue of Optimizing Transformations. In Randall Rustin (editor), *Design and Optimization of Compilers*, pages 1–30. Courant Computer Science Symposium, 5th, New York, 1971, Prentice–Hall, Englewood Cliffs, New Jersey, 1971.

[Davis 82] A.L. Davis and R.M. Keller. Dataflow program graphs. *Computer* 15(2):26–41, February, 1982.

[Keller 79] R.M. Keller, G. Lindstrom, and S. Patil. A loosely–coupled applicative multi–processing system. In *AFIPS*, pages 613–622. AFIPS, June, 1979.

[Keller 80] R. M. Keller. *Semantics and Applications of Function Graphs.* Technical Report UUCS–80–112, University of Utah, Computer Science Department, 1980.

[Keller 81] R.M. Keller and G. Lindstrom. Applications of feedback in functional programming. In *Conference on functional languages and computer architecture*, pages 123–130. October, 1981.

[Keller 82] R.M.Keller. *FEL (Function–Equation Language) Programmer's Guide.* AMPS Technical Memorandum No. 7, Dep. of Computer Science, Univ. of Utah, March, 1982.

[Muchnick 81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice–Hall, 1981.

[Schaefer 73] Marvin Schaefer. *A Mathematical Theory of Global Program Optimization.* Prentice–Hall, 1973.

[Tanaka 82] Jiro Tanaka. Code Optimization in Applicative Architectures. 1982. A dissertation proposal for Doctor of Philosophy.