A SIMPLE OPTIMIZER FOR FP-LIKE LANGUAGES

N. Islam, T.J. Myers, P. Broome*
Department of Computer and Information Sciences
University of Delaware
Newark, Delaware 19711

ABSTRACT

Functional languages provide a framework in which combining existing programs to produce new ones is particularly simple and elegant. However, the penalty usually paid for such simplicity and elegance is poor execution efficiency, especially if the program under consideration is a combination of programs that are more general than required for the problem.

We describe extensions and implementation techniques with which such combinations can be transformed into more specialized and generally faster programs. Our system is based on delaying explicit application for as long as possible by treating applications as compositions of suitable functions. The usual reduction rules then become a subset of the set of "optimization rules" which form the basis of our optimizer. These rules are similar to the identities of the algebra of programs given by Backus, but operate in a common framework in which composition plays a role similar to that of application in the reduction rules.

Keywords: Functional programming, optimization, program transformation, animation.

* Also with USARRADCOM, Chemical Systems Laboratory, DRDAR-CLB-PC, Aberdeen Proving Ground, MD 21010.

INTRODUCTION

Functional languages, advanced in [Bac78], provide very high level support to a programming style in which general purpose functions are combined with the aid of general purpose functional forms. The resulting programs frequently incur an execution-time penalty due to the tenuous connection between programming concepts and machine concepts. Two solutions are possible: adapt the machine to the program, as in [Mag79], or employ a clever translation procedure to produce an equivalent but more efficient program.

Work such as [Bur77] and [Lov77], investigating the second approach, have found that such translation can best be done by using high-level program transformations to optimize before compiling, rather than by first compiling and then attempting to optimize the low level code. This has the advantage that when a special architecture (e.g., parallel reduction machine) becomes available, its capabilities can be relatively easily utilized by transforming the programs appropriately; for example, transformations such as those studied in [Mye81] can increase the potential use of parallelism.

We have found that the algebra of programs presented by Backus forms an excellent basis for such program transformation. Our system is based on delaying explicit application for as long as possible by treating applications as compositions of suitable functions. The usual reduction rules then become a subset of the set of "optimization rules" which form the basis of our optimizer. These rules are similar to the identities of the algebra of programs given by Backus [Bac78], but operate in a common framework in which composition plays a role similar to that of application in the reduction rules. The optimizer transforms programs by rewriting them according to the optimization rules.

We describe the implementation of a simple language for animation, an area in which

such optimization has been found to be particularly beneficial at both a high level (e.g., animation sequence generators) and a low level (e.g., matrix multiplications). We show examples of each of these. The matrix multiplication optimization depends upon the assertion that a particular matrix is diagonal; such assertions are made implicitly by the use of functions which mimic the structural properties being asserted. This makes it possible for us to achieve an optimization similar to, although less general than, that of [Lov77] with a recursive algorithm that is easy to implement.

Notation

The notation used in this paper is similar to that of [Bac78] with a few exceptions. We use an infix '.' to denote function composition (e.g., f.g), and a prefix '#' to denote constant functions (e.g., #x). We also drop the def symbol in function definitions, and extend the syntax to permit multiple definitions in the form

$$[f_1,f_2,\ldots,f_n] = E$$

which is taken as being an abbreviation for

$$f_1 = 1.E$$
$$f_2 = 2.E$$
$$\ldots$$
$$f_n = n.E$$

If an expression e_1 reduces, in one or more steps, to an expression e_2 using the reduction rules of [Bac78], we express this fact as

$$e_1 \Rightarrow e_2$$

Consider the application

```
ADD.[ID,#1]:3                                (1)
```

The sequence of reductions is

```
ADD.[ID,#1]:3  =>  ADD:([ID,#1]:3)
               =>  ADD:<ID:3,#1:3>
               =>  ADD:<3,1>
               =>  4
```

Reduction is program transformation

In reduction languages, as in many other classes of languages, evaluation is source-to-source program transformation. The reduction rules specify how an expression containing an application is to be transformed; there is, in effect, one rule for each primitive function and functional, and a few higher-level rules to deal with metacomposition and user-defined functions and functionals.

OPTIMIZATION

We rephrase applications as compositions, thus viewing (1) as the composition

```
ADD.[ID,#1].#3                               (2)
```

This is similar to the manner in which data is represented as constant functions within FP function expressions. With an appropriate set of transformation rules, the composition (2) "reduces" to #4, which is the constant function corresponding to the object to which the application (1) reduced. We call such transformation rules "opt" rules, and use the terms "reduce" and "reduction" to refer to such transformations as well as to the usual reduction rules for application. The sequence of reductions for the composition (2) is as follows.

```
ADD.[ID,#1].#3  =>  ADD.[ID.#3,#1.#3]
                    ADD.[#3,#1]
                    #4
```

The rules used in the above derivation were

$$[f_1,f_2,\ldots,f_n].g \Rightarrow [f_1.g,f_2.g,\ldots,f_n.g]$$
$$ID.g \Rightarrow g$$
$$\#x.g \Rightarrow \#x$$
$$ADD.[\#x,\#y] \Rightarrow \#(x+y)$$

(Here the f_i and g are arbitrary functions, and x and y are arbitrary constants. We use '=>' also to separate the left and right parts of rules.)

Notice that each of the above rules has a central composition on the left hand side; i.e., all rules are of the form

```
f.g  =>  E
```

where E is a function expression. This is true of all opt rules. Having opt rules "driven" by compositions yields a close resemblance between them and the reduction rules which are similarly driven by application. However, there is a major difference: composition is associative, but application is not, i.e.,

$$f.(g.h) = (f.g).h$$

but

$$f:(g:x) \neq (f:g):x \quad \text{generally}$$

As a result, there are generally more composition subexpressions candidate for reduction than there are application subexpressions. (In our current system we have not exploited this property as much as we could have.)

We can now usefully include opt rules like

```
ADD.[g,#0]  =>  g
ADD.[#0,g]  =>  g
```

to assert the fact that

```
ADD:<x,0>  =>  x
ADD:<0,x>  =>  x
```

independently of the value of x.

These rules can properly be called optimization rules because they transform non-constant functions into loosely equivalent functions, and also improve their efficiency. We use the qualifier "loosely" to mean that the resulting function may be more defined than the original. In other words, we may lose strictness and have a call-by-need semantics instead of FP's strict call-by-value semantics. Except for this possible loss of strictness, our rules are similar to those of the algebra of programs given in [Bac78].

We have developed a set of such rules (see the Appendix), and have incorporated it into an interpreter that permits the user to define efficient new functions as specializations of existing general functions.

To illustrate this, we take the example of matrix multiplication in which one argument is known to be diagonal. This example is considered in [Lov77], where optimization is accomplished by substituting, for each occurrence of the diagonal argument, an if-then-else statement asserting the diagonal property, and then propagating this through the body of the procedure. We do something similar, except that our assertion takes the form of a construction function which is simply composed onto the original function. This composition is then optimized, the effect of the assertion being propagated by the optimizer using the opt rules.

Matrix multiplication

Consider the definitions

```
IP = (/ADD).(*MULT).TRANS
MM = (*(*IP)).(*DISTL).
                    DISTR.[1,TRANS.2]
ID3X3 = [[1.1,2.1,3.1],
         [1.2,2.2,3.2],
         [1.3,2.3,3.3]]
DIA3X3 = [[1,#0,#0],
          [#0,2,#0],
          [#0,#0,3]]
MMDIA = MM.[DIA3X3.1,ID3X3.2]
```

Here IP and MM are the inner product and matrix multiplication functions as given in [Bac78]. ID3X3 is the identity function for 3x3 matrices. DIA3X3 is a function that takes a vector of three values and produces a 3x3 matrix with those values on the diagonal. MMDIA is a function that is defined as a specialization of MM to the case where the first argument is a diagonal matrix.

The optimizer transforms the definition of MMDIA into

```
MMDIA' = [[MULT.[1.1,1.1.2],
              MULT.[1.1,2.1.2],
                MULT.[1.1,3.1.2]],
           [MULT.[2.1,1.2.2],
              MULT.[2.1,2.2.2],
                MULT.[2.1,3.2.2]],
           [MULT.[3.1,1.3.2],
              MULT.[3.1,2.3.2],
                MULT.[3.1,3.3,2]]]
```

The results of counting the number of CONSes that the interpreter executed while applying the unoptimized and optimized forms of MMDIA are given below.

```
Unoptimized:
    Each call                729 CONSes
Optimized:
    First call              1245 CONSes
    Each subsequent call     193 CONSes
```

Here, the "First call" count includes the cost of optimization. Note that 516 CONSes were expended (once only) in improving the program by 536 CONSes (for each subsequent call) -- an increase in speed by a factor of about 3.7 .

Our diagonal assertion also fixes the dimensions of the argument matrix. This may be undesirable in some applications, but in low level graphics programming it is quite acceptable.

A better example of the improvement that our system can achieve is multiplication of a sparse matrix with a vector -- a problem very common in low-level graphics algorithms. In such problems, the matrix is usually a transformation matrix (for rotation, scaling or translation) and the vector a point to be transformed. Consider the following definitions (for a two-dimensional homogeneous coordinate system)

```
SCL = [[1,#0,#0],
       [#0,2,#0],
       [#0,#0,#1]]
PNT = [[1],[2],[3]]
SCLPNT = MM.[SCL.1,PNT.2]
```

The function SCLPNT, which takes a scaling transformation matrix and a point vector (actually a 3x1 matrix) and yields the transformed point vector, optimizes to

```
SCLPNT' = [[MULT.[1.1,1.2]],
           [MULT.[2.1,2.2]],
                 [3.2]     ]
```

which is a considerably improved form of the original definition which was based on a too-general MM.

The CONS counts for the unoptimized forms of SCLPNT were

```
Unoptimized:
    Each call                317 CONSes
Optimized:
    First call               550 CONSes
    Each subsequent call      55 CONSes
```

Again, the "First call" count includes the cost of optimization. Here, 233 CONSes were expended in improving the program by 262 CONSes, increasing speed by a factor of about 5.7.

Type checking

The optimizer also serves as a somewhat limited type-checker. There is no static typing in the language; all functions are expected to do dynamic type-checking of their own. However, some types can be inferred. For example, a function construction of length n must, when applied to any object, yield either $\underline{|}$ or a sequence of length n. If the individual components of the function construction were themselves function constructions, the individual components of the resulting sequence will again either be $\underline{|}$ or sequences of corresponding lengths, and so on. If a function expression optimizes to $\#\underline{|}$ (the everywhere-undefined function), then it means that the original function would yield $\underline{|}$ when applied to any object. Thus, $\underline{|}$-returning functions, when optimized, would terminate more quickly with the error notification.

Examples of $\#\underline{|}$-returning transformation rules are

```
i.[]  =>  #|
i.[g1,...,gn]  =>  #|  if i>n
MULT:[g1,...,gn]  =>  #|  if n≠2
DIV.[g1,#0]  =>  #|
etc.
```

Of course, f.g may diverge for all x even if f.g $\not\Rightarrow \#\underline{|}$.

Graphics

We have developed a simple FP-like language for describing time-varying pictures. Each graphic object is a function that maps a time sequence (0,1,2, ...) into a sequence of wire-frame pictures representing the object at corresponding times. They may be specified to have arbitrary and independent motion.

The programs are manipulatable and can thus be subjected to our kind of optimization transformations fairly easily. These transformations are particularly useful in animation because often large parts of the pictures are constant in time. These constant parts get optimized into constant functions producing the corresponding data structures, while the time-varying parts get optimized into somewhat better (although not constant) functions because some computations are done at transformation time. Since the framework for constant and time-varying objects is the same, it is extremely easy to redefine a constant object as a time-varying one (with immediate gain in speed) or vice-versa.

A call-by-need mechanism is used in our formulation of graphic sequences because our time sequence is an infinite stream and the corresponding graphic pictures also form a stream that must be evaluated (and displayed) in a lazy fashion.

The following program describes a bicycle (with square wheels!) moving uniformly with time, with the wheels rotating in synchronism.

```
BIKE = PICTURE.[FRAME,BWHEEL,FWHEEL]
FRAME = MOVE.[PICTURE.
                    (*LINE).[[FR1,FR2],
                             [FR2,FR3],
                             [FR4,FR5],
                             [FR5,FR6]],
              BAXLE]
[FR1,FR2,FR3,FR4,FR5,FR6] =
                (*POINT).[[#0,#0],
                          [#100,#200],
                          [#24,#200],
                          [#15,#300],
                          [#200,#300],
                          [#300,#0]]
BAXLE = POINT.[MULT.[ID,#5],#200]
FAXLE = MOVE.[BAXLE,POINT.[#300,#0]]
[FWHEEL,BWHEEL] =
                (*MOVE).[[ROTWHL,FAXLE],
                         [ROTWHL,BAXLE]]
ROTWHL = ROTATE.[WHEEL,ANGLE]
ANGLE = MULT.[ID,#(.1)]
WHEEL = PICTURE.(*LINE).[[LB,RB],
                         [RB,RT],
                         [RT,LT],
                         [LT,LB]]
[LB,RB,LT,RT] =
              (*POINT).[[#-100,#-100],
                        [#100,#-100],
                        [#-100,#100],
                        [#100,#100]]
```

(Note how optimization allows us to use mapping in our function definitions without penalty.)

The functions PICTURE, LINE and POINT are primitive constructor functions, and MOVE and ROTATE are primitive transformation functions. The function BIKE defines a picture whose components are generated by the functions FRAME, BWHEEL, and FWHEEL. FRAME defines the frame consisting of lines joining certain pairs of points with the frame's origin "attached" to the point BAXLE. BAXLE, which represents the rear axle, defines a point whose y coordinate is fixed at 200 and whose x coordinate increases at the rate of 5 coordinate units per time unit. FAXLE, which represents the front axle, defines a point relative to the moving BAXLE. FWHEEL AND BWHEEL define instances of ROTWHL attached

to the points FAXLE and BAXLE respectively. ROTWHL represents a rotating wheel and is a function of ANGLE, which, in turn, is a function of time. WHEEL defines a square with corners at the points LB, RB, LT, and RT.

The CONS counts for the unoptimized and optimized forms of BIKE were

Unoptimized:	
Each call	1715 CONSes
Optimized:	
First call	2303 CONSes
Each subsequent call	1143 CONSes

As before, the "First call" count includes the cost of optimization. Here, 588 CONSes were expended in improving the program by 572 CONSes, increasing its speed by a factor of about 1.5.

This is a substantial speedup in view of the fact that all parts of the picture vary with time. In cases where large parts of a picture are time-independent, we can expect correspondingly large improvements beyond those shown in this example.

Optimizing functions

Associated with each function is an optimizing function, called its opt function, which is invoked when a composition with that function on the left is chosen for reduction. The opt function can then examine the form of the function expression on the right using predefined meta-functions, and can construct new functions when a reduction can be done. The opt function is usually a conditional expression on the structure of the right hand function expression. If no predicate of the conditional expression is satisfied, the composition is returned unchanged.

Strategies

"Strategy" determines the order in which compositions are picked for optimization. A non-deterministic (and possibly parallel) optimization of the various subexpressions could be performed.

Our current strategy is a kind of demand propagation from the left (top) going to the right (down), in a manner similar to evaluation, i.e., the token of control for optimization is passed up and down in a manner similar to the behavior of ':' in [Bac78]. Each (successful or unsuccessful) optimization step specifies where the optimization should be attempted next, in a manner much less general than in Loveman's system [Lov77]. In our case, the points of application are always specified to be within the resulting expression. For example, having used

$$[f_1,f_2,\ldots,f_n].g \Rightarrow [f_1.g,f_2.g,\ldots,f_n.g]$$

the opt function for function construction will cause all compositions in the r.h.s. to undergo optimization. The opt functions explicitly call the optimizer to attempt to optimize subexpressions which are components of the structure that they return. Care is taken not to recurse where doing so might cause infinite recursion. Since we have not yet formally proven our set to be loop-free, we have omitted these specifications from the rules given in the Appendix.

Space-time tradeoff

There is a tradeoff between space and time for several of the optimizations. For example, in

$$*f.[g_1,\ldots,g_n] \Rightarrow [f.g_1,\ldots,f.g_n]$$

the l.h.s. is better in space while the r.h.s. is better in time. In

$$[f_1,\ldots,f_n].g \Rightarrow [f_1.g,\ldots,f_n.g]$$

the r.h.s. is worse in both space and time unless some of the $f_i.g$ can be reduced. Our system applies these reductions out of optimism.

THE INTERPRETER

The interpreter is a demand-driven simulation of the underlying reduction system. It accepts definitions in the form

```
f = E
```

where f is a unique function name (there is no scoping of names whatsoever) and E is a function expression. It also accepts queries in the form

```
?f
```

There is no explicit application. When a query is encountered, the interpreter optimizes f (if not already optimized) and all subordinate functions (those not already optimized) and then applies f to 0,1,... successively and displays the result after each application. One can think of the application as being even further delayed and say that the system optimizes successively f.#0, f.#1, ... and applies the resulting functions (which must then be constant functions) to an arbitrary object.

As mentioned earlier, with each function the interpreter associates an opt function; it also associates an evaluation function. (We are attempting to combine these functions in such a way that the evaluation function body becomes

just another clause in the conditional expression of the opt function.)

In addition, for each user-defined function, the interpreter associates an unoptimized definition and an optimized definition. It also maintains a graph of function dependencies in order to delete optimized definitions when any subordinate function is redefined.

CONCLUSIONS

We have shown that a fairly simple optimization scheme, based on Backus's algebra of programs, can effect substantial improvement in execution speed of functional programs. This encourages the programmer to build new programs from existing general-purpose programs without too much concern for efficiency. This process can be viewed as one of successive specialization of general-purpose functions from a suitable base library.

We have shown that this scheme is quite successful in graphics programming, and we expect similar results in areas where the structure of the data involved provides implicit assertions which obviate the need for bidirectional transformations, goal direction and, thus, planning.

We are investigating several aspects of this approach. We are attempting to extend the system to include inference of attributes other than structure. We are also exploring new strategies, particularly those with little or no heuristic search, so as to strike a good balance between simplicity of the optimizer and the improvement it yields. We are also updating the library of optimization rules and extending the kinds of functions that could form a suitable library of assertion functions.

Comparing our system with Loveman's, we find that Loveman had to contend with, in effect, three languages -- assertions, expressions, and statements. On the other hand, FP-like languages provide a much more hospitable medium for such transformations. The absence of side effects and variables (free and bound) eliminates many of the problems of environments and binding, while the property that all functions take exactly one argument simplifies considerably the interfacing of functions and provides more freedom in their transformation.

ACKNOWLEDGEMENTS

This paper is based on part of Noorul Islam's doctoral dissertation work currently in progress. We would like to thank Dr. Hatem Khalil, Dr. Toni Cohen, and the referees for pointing out errors and suggesting numerous improvements.

APPENDIX

Metacomposition rule

(m f).g => m.[[#m,#f],g]

Opt rules for some primitive functions

Identity

ID.g => g

Selectors

i.[] => #⊥

$i.[g_1,\ldots,g_n] \Rightarrow g_i$ if $1 \le i \le n$

Tail

TL.[] => #⊥

$TL.[g_1] \Rightarrow$ #[]

$TL.[g_1,g_2,\ldots,g_n] \Rightarrow [g_2,\ldots,g_n]$

Length

LEN.[] => #0

$LEN.[g_1,g_2,\ldots,g_n] \Rightarrow$ #n

Null

NULL.[] => #T

$NULL.[g_1,\ldots,g_n] \Rightarrow$ #F

Distribute

$DISTL.[g_1,[g_{21},g_{22},\ldots,g_{2n}]]$
$\Rightarrow [[g_1,g_{21}],[g_1,g_{22}],\ldots,[g_1,g_{2n}]]$

$DISTR.[[g_{11},g_{12},\ldots,g_{1n}],g_2]$
$\Rightarrow [[g_{11},g_2],[g_{12},g_2],\ldots,[g_{1n},g_2]]$

Arithmetic

$ADD.[\#0,g_2] \Rightarrow g_2$

$ADD.[g_1,\#0] \Rightarrow g_1$

$SUB.[\#0,g_2] \Rightarrow g_2$

$MULT.[\#0,g_2] \Rightarrow$ #0

$MULT.[g_1,\#0] \Rightarrow$ #0

$MULT.[\#1,g_2] \Rightarrow g_2$

$MULT.[g_1,\#1] \Rightarrow g_1$

$DIV.[g_1,\#0] \Rightarrow$ #⊥

$DIV.[\#0,g_2] \Rightarrow$ #0

DIV.$[g_1,\#1]$ => g_1

ADD.$[g_1,\ldots,g_n]$ => $\#\underline{!}$ if $n\neq 2$

etc.

Equal

EQUAL.$[g_1,g_2]$ => #T

if $g_1=g_2$ (intensionally)

EQUAL.$[[g_{11},\ldots,g_{1m}],$

$[g_{21},\ldots,g_{2n}]]$

=> #F if $m\neq n$

Transpose

TRANS.$[g_1,$

$\ldots,$

$[g_{i1},\ldots,g_{in}],$

$\ldots,$

$g_m]$

=> $[[1.g_1,\ldots,g_{i1},\ldots,1.g_m],$

$[2.g_1,\ldots,g_{i2},\ldots,2.g_m],$

...

$[n.g_1,\ldots,g_{in},\ldots,n.g_m]]$

Constant

#x.g => #x

Construction

$[f_1,f_2,\ldots,f_n].g$

=> $[f_1.g,f_2.g,\ldots,f_n.g]$

Conditional

$(p\rightarrow f_1;f_2).g$

=> $f_1.g$ if p.g => #T

$f_2.g$ if p.g => #F

$(p\rightarrow f_1;f_2).g$ otherwise

Apply-to-all

(*f).[] => []

(*f).$[g_1,g_2,\ldots,g_n]$

=> $[f.g_1,f.g_2,\ldots,f.g_n]$

REFERENCES

[Bac72] Backus, J. Reduction Languages and Variable-free Programming, IBM Research Report RJ1010, Yorktown Heights, NY, April 7,1972.

[Bac73] Backus, J. "Programming language semantics and closed applicative languages", Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71-86.

[Bac78] Backus, J. "Can Programming be Liberated from the Von-Neumann Style? A Functional Style and its Algebra of Programs", CACM, 21, 8, August 1978.

[Bur77] Burstall, R.M. and Darlington, J. "A transformation system for developing recursive programs", JACM, 24, pp. 44-67, Jan. 1977.

[Lov77] Loveman, D.B. "Program Improvement by Source-to-Source Transformation", JACM, 24, 1, 1977, pp. 121-145.

[Mag79] Mago, G.A. "A Network of Microprocessors to Execute Reduction Languages, Part 1", Int. J. Comptr. and Inf. Sci., 8, 5, 1979.

[Mye81] Myers, T.J. "Operator-directed Program Transformations" (in preparation).

[Poz77] Pozefsky, M. "Programming in Reduction Languages", Ph.D. dissertation, U. of N. Carolina, Chapel Hill, 1977.