

An optimizing compiler based on program transformation

John H. Williams and Edward L. Wimmers

IBM Almaden Research Center, K53-803

650 Harry Road

San Jose, CA 95120-6099

March 12, 1991

FL [BW89] is a functional programming language designed to facilitate writing high level programs that are clear and concise. Unfortunately, such programs are often extremely inefficient if implemented in a straightforward way. In [Ba85] Backus proposed an approach to optimizing functional programs which consists of introducing new “Fortran-like” constructs into the language and constructing rewrite rules that transform clear, high-level user programs into more complicated but very efficient programs using the new constructs. This paper describes some preliminary results of this approach to building an optimizing compiler for FL.

The paper uses a simple running example, factorial, to motivate and illustrate the techniques involved. Section 1 describes the stylistic differences between programs in FL and conventional languages and motivates the transformational approach to optimization. Section 2 outlines the structure of the compiler and describes the rewrite rules. Section 3 contains a trace of the rewrite rules used in optimizing the factorial program, and section 4 shows how the rules are used in transforming a clear but very inefficient program for matrix multiplication into the usual triply nested loop of conventional languages. Finally, section 5 discusses the present status of the work and the plans for the future.

1 A simple example

The factorial function is a good example to illustrate the stylistic differences between FL and conventional languages.

Specification:

Factorial of a positive integer n is defined to be the product of the integers from 1 to n .

FL program:

```
def fact  $\leftarrow$  isposint  $\equiv$  *  $\circ$  intsto
```

C program:

```
fact(arg)  int arg;  
{int ans,count;  
  ans = 1;  
  for (count = 1; count <= arg; count++)  
    ans *= count;  
  return(ans);}
```

The FL program is written using the three primitive functions **isposint**, *****, and **intsto**, and the program forming operations $\equiv$, $\leftarrow$, and $\circ$. The predicate **isposint** is true if its argument is a positive integer; the function ***** computes the product of a sequence of numbers; and the function **intsto** applied to a positive integer returns the sequence of integers from 1 to n . For example,

isposint:4 = true

intsto:4 = $\langle 1,2,3,4 \rangle$

***: $\langle 1,2,3,4 \rangle$ = 24**

Thus, the FL definition reads, “to compute the factorial of a positive integer n , first build the sequence of integers $\langle 1, \dots, n \rangle$ and then multiply them together.”

Most people would prefer to write the FL program. Being expressed at a higher level, it is shorter and clearer. This can be verified by trying to describe the two programs to someone unfamiliar with both languages. Why then does the world continue to use conventional languages? Efficiency. If executed as written, the FL program first builds a sequence of n integers, allocating n words of memory (perhaps more depending on how sequences are represented) and then makes another pass over the sequence to compute the product (at

which point the sequence has no more use and is “garbage”). This wastes both time and space, since the C program executes in constant space, generating successive integers only as it needs them.

The C programmer opts for efficiency over clarity, explicitly manipulating the variables “ans” and “count”, because doing so is much more time and space efficient. To a large extent it is this trade-off that makes programming so costly and error-prone. If it were possible to write a program like factorial in the FL style and trust that the compiler would perform the necessary optimizations to produce a program like the C version, functional programming would become a viable alternative to conventional languages.

This is the task of the rewriting mechanism of the FL compiler described herein: to transform programs that are written in a modular style and that use (possibly inefficient) intermediate data structures as module interfaces, into efficient programs.

2 The rewriting mechanism and the rewrite rules

To understand the function of the rewriting engine, it is necessary to describe briefly the overall structure of the compiler. There are five phases of compilation: parsing, simplification, type analysis, rewriting, and code generation.

1. The parser removes most of the syntactic sugar and produces an abstract syntactic description of the input program.
2. The program is then converted to a very simple representation in which expressions consist only of atoms and functions closed under sequence construction and function application, and every user-defined symbol is associated with an expression which is guaranteed to be a function. In this phase all environments are flattened into a single, global name space, resulting in a program of the form:

```

expr  where{ def  $f_1 \equiv \text{expr}_1$ 
              ...
              def  $f_n \equiv \text{expr}_n$  }
```

These expressions are stored as trees of “rnodes” (for “rewrite” nodes).

3. The next phase of compilation is “type checking”. Since FL is not a strongly typed language, this is not the usual task of checking that the program satisfies some type constraints imposed by the language. Rather, this phase is more akin to “static analysis” in conventional compilers. For every expression that represents a function,

three “types” (sets of regular tree expressions) are computed: the set of arguments to which the function gets applied, the set of its possible results, and the set of possible errors it can generate. Type analysis is described in much more detail by Alex Aiken and Brian Murphy in [Mu90] and [AM91].

4. During the rewriting phase, the rewrite engine searches the rnode trees for redexes, substitution instances of the left-hand-sides of the rewrite rules, and replaces them with their corresponding right-hand-sides. It repeats this process until no further redexes remain. It is crucial that this process be both fast, since many transformations may be needed to produce the desired optimization, and sophisticated, in order to accommodate the powerful sorts of rules described in the next section. The rewrite engine is described in detail in a forthcoming report by Paul Tucker and Ed Wimmers [TW91].
5. Finally, the code generator produces target code from the rewritten rnode representation. Presently, the target code is CLisp; a C backend is currently being written.

2.1 The Fortran-like combining forms

As explained in Section 1, the purpose of the rewriter/optimizer is to transform programs into functionally equivalent versions for which efficient code can be generated. This task is facilitated by introducing certain “Fortran-like”, higher-order functions into the language, functions that map readily into efficient code on a conventional sequential machine. Since they reduce the clarity of programs that employ them, the intention is that they not be used by programmers but rather as a convenience for the rewriter. The forms are Fortran-like only in that they are patterned after Fortran DO loops; unlike their Fortran counterparts, they are pure functions without side-effects. Consequently, they can’t be used to transform algorithms whose correctness depends on the cumulative effect of repeated assignments, but they are useful in eliminating the creation of intermediate data structures.

Reduce

The form **reduce** is used to compute a single result by bounded iteration. It takes three arguments, a function to be applied to an accumulator and successive values, a function to initialize the accumulator, and a function to determine the number of iterations. For example, **reduce**: $\langle(\text{isposint} \circ \text{sel} \circ \text{s2} \rightarrow \text{ar} \circ [\text{s1}, \text{sel} \circ \text{s2}]; \text{s1}), [], \text{len}\rangle$ is a function that selects those elements of its input sequence that are positive integers. Note that **reduce** can be defined in FL:

```

def reduce(⟦E.isfunc, initial.isfunc, lim.isfunc⟧) ≡
    /l:E ∘ al ∘ [initial, (intsto ∘ lim)  distr'  id]

```

For

for is a form that is useful for computing a sequence of predetermined length. It takes two arguments, a function for computing the *i*th element of its result, and a function for determining the length of its result. For example, **for**: $\langle \mathbf{sel} * \mathbf{sel}, \mathbf{len} \rangle$ is a function that squares every element of a sequence. As with **reduce**, **for** can be defined in FL:

```

def for(⟦E.isfunc, lim.isfunc⟧) ≡ aa:E ∘ ((intsto ∘ lim)  distr'  id)

```

Examples of the use of these functions appear below.

2.2 The rewrite rules

This section describes the rewrite rules that are used to transform FL programs. Some of the rules (eg, **id_elim** and **intsto_elim**) express particular properties of primitive FL functions. More general rules (eg, **reduce_intro** and **cond_dist_in**) express mutually distributive properties of the FL combining forms. Wherever possible, the strategy is to include a few general rules rather than many special purpose rules, even though this can increase the number of transformations required to optimize a particular program. This is done to keep the size of the rule libraries manageable.

To guarantee termination of the rewriter and to help understand the effect of optimization, the rules are organized into different classes, some of which are:

1. Simplification rules
2. **for** and **reduce** manipulation rules
3. Rules to remove higher-order functions
4. Distribution rules
5. Flag manipulation rules

The flag manipulation rules are included not to contribute to program optimization but merely to facilitate the operation of the rewrite engine. In the rewriting traces that follow these steps are not shown.

The following are examples of each of these classes together with a brief justification for their validity. It is important to note that all of the rules strictly preserve the meanings

of programs, including their error behavior. This desideratum and its implications are discussed in [AW90]. The basic format of a rewrite rule is $\mathbf{lhs} \Longrightarrow \mathbf{rhs}$, which means that anything that matches (a substitution instance of) the expression $\mathbf{lhs}$ can be replaced by the expression $\mathbf{rhs}$; $\$f$ denotes a variable that can match any FL function; and (following [AW90]) $\mathbf{expr_safe}$ denotes a function valued expression that is guaranteed not to produce an error when it is applied.

1. typical simplification rules

$$(\mathbf{id_elim}) \quad \mathbf{id} \circ \$f \Longrightarrow \$f$$

$\mathbf{id}$ is the identity function and hence can be eliminated

$$(\mathbf{tl_al_elim}) \quad \mathbf{tl} \circ \mathbf{al_safe} \Longrightarrow \mathbf{s2}$$

$\mathbf{al}$ needs to be safe to preserve error behavior

$$(\mathbf{signal_collapse}) \quad \$f \circ \mathbf{signal} \Longrightarrow \mathbf{signal}$$

all functions are strict with respect to errors and hence the function $\$f$ need not be called.

2. rules for manipulating the **for** and **reduce** constructs

$$(\mathbf{intsto_elim}) \quad \mathbf{intsto_safe} \Longrightarrow \mathbf{for}:\langle \mathbf{s1}, \mathbf{id} \rangle$$

$$\mathbf{intsto}:\mathbf{n} = \langle \mathbf{1}, \dots, \mathbf{n} \rangle = \langle \mathbf{s1}:\langle \mathbf{1}, \mathbf{n} \rangle, \dots, \mathbf{s1}:\langle \mathbf{n}, \mathbf{n} \rangle \rangle = \mathbf{for}:\langle \mathbf{s1}, \mathbf{id} \rangle:\mathbf{n}.$$

$$(\mathbf{reduce_intro}) \quad \mathbf{*_safe} \circ \mathbf{for}:\langle \$f_safe, \$1 \rangle \Longrightarrow \mathbf{reduce}:\langle (* \circ [\mathbf{s1}, \$f \circ \mathbf{s2}])_safe, \sim \mathbf{1}, \$1 \rangle$$

Multiplication over a sequence can be performed by initializing an accumulator to **1** and then multiplying successively the accumulator by each element in the sequence.

3. higher-order function removal

$$(\mathbf{K_elim}) \quad \mathbf{apply} \circ [\mathbf{K}, \$f_safe] \Longrightarrow \mathbf{id}$$

$$\mathbf{K}:\mathbf{x}:y = \mathbf{x} \text{ and therefore } (\mathbf{apply}:[\mathbf{K}, \$f]):\mathbf{x} = \mathbf{apply}:\langle \mathbf{K}:\mathbf{x}, \$f:\mathbf{x} \rangle = \mathbf{K}:\mathbf{x}:(\$f:\mathbf{x}) = \mathbf{x} = \mathbf{id}:\mathbf{x}.$$

4. distributing in

$$(\mathbf{cond_dist_in}) \quad \$f \circ (\$p \rightarrow \$g_1; \$g_2) \Longrightarrow (\$p \rightarrow \$f \circ \$g_1; \$f \circ \$g_2)$$

Any function can be distributed from the left into a conditional. Note that this rule results in two copies of $\$f$.

5. flag manipulation

(flag-cons) $(\$p_safe \rightarrow \$g1_safe; \$g2_safe) \implies (\$p \rightarrow \$g1; \$g2)_safe$

If all three functions in a conditional are safe, then the entire conditional is itself safe.

It is important to note that the effectiveness of the rewrite phase depends heavily on the quality of the information deduced during static analysis. Many of the rules are valid only if it is known that the rewritten functions are “safe”, i.e. guaranteed not to produce errors.

3 Optimizing factorial

This section traces the essential rewriting that occurs when the compiler is applied to the FL version of factorial (i.e. flag manipulations have been deleted). In each of the following lines, the redex is shown underlined, and between each pair of lines, the matching rule is identified.

The first line is the result of “desugaring” the program from section 1 by replacing the $\leftarrow$ construct with a conditional.

isposint $\rightarrow * \circ \underline{\text{intsto}}$; **signal** $\circ [\sim \text{fact}, \sim \text{arg1}, \text{id}]$

using the intsto-elim rule $\text{intsto_safe} \implies \text{for} : \langle \text{s1}, \text{id} \rangle$

$= \text{isposint} \rightarrow * \circ \underline{\text{for} : \langle \text{s1}, \text{id} \rangle}$; **signal** $\circ [\sim \text{fact}, \sim \text{arg1}, \text{id}]$

using the reduce-intro rule

$_safe \circ \text{for} : \langle \$f_safe, \$l \rangle \implies \text{reduce} : \langle (* \circ [\text{s1}, \$f \circ \text{s2}])_safe, \sim 1, \$l \rangle$ with

$\$f = \text{s1}$

$\$l = \text{id}$

$= \text{isposint} \rightarrow \text{reduce} : \langle \text{mul} \circ [\text{s1}, \text{s1} \circ \text{s2}], \sim 1, \text{id} \rangle$; **signal** $\circ [\sim \text{fact}, \sim \text{arg1}, \text{id}]$

Thus, the FL program is transformed into a program that executes like the C program in Section 1 when applied to a positive integer and returns the proper error value otherwise.

4 Matrix multiply

It is interesting to see how the rewrite rules perform when applied to a larger example. The following is a program that first creates two test matrices and then multiplies them.

Both the creation and the multiplication functions are written in a straightforward but very inefficient way, employing several large temporary data structures.

4.1 The FL version

The program is the composition of **build**, a program that builds a pair of test matrices given their dimensions, and **mm**, a program that multiplies two conformable matrices. Briefly, the following code reads, “ **build**, when applied to integers **m**, **n**, and **p**, uses **buildmat** to produce $m \times n$ and $n \times p$ matrices, and produces a “build error” if applied to other than a triple of positive integers. **buildmat**, applied to a pair **i** and **j**, produces a matrix (a sequence of equal length sequences) of **i** rows, each consisting of the integers from **1** to **j**. **copies_of** uses two very inefficient data rearranging operations, **distl** and **aa:s1** (‘select the first element of each pair’), to produce **n** copies of its **data** argument.”

```
mm o build
where
{
Def build ← [[m.isposint,n.isposint,p.isposint]] ≡ aa:buildmat o [[m,n],[n,p]]
  Def buildmat ← [[i.,j.]] ≡ i copies_of' intsto o j
  Def copies_of ← [[n.,data.]] ≡
    aa:s1 o (data distl' intsto o n)
Def mm ≡ aa:(aa:IP) o scart o [s1,trans o s2]
  Def scart ≡ (aa:distl) o distr
  Def IP ≡ + o (aa:*) o trans
}
```

The matrix multiply function **mm** works by making several copies of its input matrices and restructuring these copies to form the proper row-column pairs for inner product:

after:	the shape is:
(input)	(mxn,nxp)
[s1, trans@s2]	(mxn,pxn)
distr	mx(n,pxn)
aa:distl	mxx2xn
aa:(aa:___)	
trans	mxx x nx2
aa:*	mxx x n
+	mxx

4.2 The transformed program

Using the same rewrite rules and strategy used for the factorial example, the above program is transformed to:

```

E0 ≡
  (isseq → (lenis:3 → and ◦ aa:isposint;ff);ff)
  → for
    :⟨for
      :⟨mul
        ◦ [reduce
          :⟨add ◦ [s:1, s:1 ◦ s:2],
            ~0,
            s:2 ◦ s:2 ◦ s:2⟩,
          s:1],
        s:3 ◦ s:2⟩,
      s:1⟩
  ;signal ◦ [~"build", ~"arg1", id])

```

Modulo the code to produce the proper error values, this is essentially the familiar triply nested for loop (with an additional factoring of the common multiplier for each inner loop that results from the particular test matrices used).

5 Status and future work

This is definitely work in progress. At present the optimizer consists of about 400 rules and a rather complex strategy for applying them. One of the complications in developing a strategy can be seen by considering the “law” $[f \circ h, g \circ h] \equiv [f, g] \circ h$. There are two rewrite rules possible, $[f \circ h, g \circ h] \rightarrow [f, g] \circ h$ and $[f, g] \circ h \rightarrow [f \circ h, g \circ h]$. Both are potentially useful; the first to remove common sub-expressions, and the second to distribute programs into contexts where they are used, thus providing grist for the rewriting mill.

Presently, the rules are divided into eight different collections each of which can be applied exhaustively without any possibility of putting the rewriter into an infinite loop. These are then applied iteratively until no rules are fired during a round of matching. Much work remains to be done to develop more clever strategies which can adapt to the particular forms of the programs being rewritten.

The examples considered herein only scratch the surface of what can be done using this ambitious approach to optimization. An enormous amount of development and study is

required to characterize the optimizations done by a particular strategy and its associated rewrite rules.

6 Acknowledgements

The FL compiler represents the cumulative efforts of the FL group at Almaden: Alex Aiken, John Backus, Thom Linden, Peter Lucas, Brian Murphy, Paul Tucker, and the authors.

References

- [AW90] A. Aiken, J. H. Williams, and E. L. Wimmers. *Program Transformation in the Presence of Errors*. Seventeenth POPL proceedings, San Francisco, Jan. 1990, pp. 210-217.
- [AM91] A. Aiken and B. Murphy. *Static Type Inference in a Dynamically Typed Language*. Eighteenth POPL proceedings, Orlando, FL, Jan. 1991, pp. 279-290.
- [Ba78] J. Backus. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*. *Communications of the ACM*, 21(8):613–641, August 1978.
- [Ba85] J. Backus. *From Function Level Semantics to Program Transformation and Optimization*. Research Report RJ 4567, IBM, 1985.
- [BW89] J. Backus, J. Williams, E. Wimmers, A. Aiken, and P. Lucas. *FL Language Manual, Parts 1 and 2*. Research Report RJ 7100, IBM, 1989.
- [Mu90] B. R. Murphy. *A Type Inference System for FL*. Master’s thesis, MIT, 1990.
- [TW91] P. Tucker and E. Wimmers. *The FL Rewrite Engine*. Research Report (in preparation).