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

December 3, 1990

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 [Bac85] 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 ← isposint ≡ * ∘ 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 $\langle 1,...,n \rangle$ 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:

$$\begin{aligned} \texttt{expr} \quad \texttt{where\{ def } \ \texttt{f}_1 \equiv \texttt{expr}_1 \\ \cdots \\ \texttt{def } \ \texttt{f}_n \equiv \texttt{expr}_n \texttt{\}} \end{aligned}$$

   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 [Mur90] 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, these forms are added to the language because they can be implemented efficiently on existing machines. They are intended to be used not by the programmer but by the rewriter/optimizer. 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 are useful in eliminating the creation of intermediate data structures, but they can't be used to transform algorithms whose correctness depends on the cumulative effect of repeated assignments. [Note: The motivation and much of the informal description of these forms is deleted from this extended abstract.]

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**: $\langle(\texttt{isposint} \circ \texttt{sel} \circ \texttt{s2} \rightarrow \texttt{ar} \circ [\texttt{s1}, \texttt{sel} \circ \texttt{s2}]; \texttt{s1}), [], \texttt{len}\rangle$ is a function that selects those elements of its input sequence that are positive integers. Note that **reduce** can be defined in FL:

$$\texttt{def reduce}([\![\texttt{E.isfunc}, \texttt{initial.isfunc}, \texttt{lim.isfunc}]\!]) \equiv$$
$$/\texttt{l:E} \circ \texttt{al} \circ [\texttt{initial}, (\texttt{intsto} \circ \texttt{lim}) \quad \texttt{distr}' \quad \texttt{id}]$$

For

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

$$\texttt{def for}([\![\texttt{E.isfunc, lim.isfunc}]\!]) \equiv \texttt{aa:E} \circ ((\texttt{intsto} \circ \texttt{lim}) \quad \texttt{distr}' \quad \texttt{id})$$

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 combing forms. Wherever possible, the strategy is to include a few general laws rather than many special purpose laws, 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.

The following are some of the rules that are used in optimization. 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 [AWW90].
[The full paper contains a complete description of the rule format. Briefly, `lhs` $\Longrightarrow$ `rhs` means that anything that matches (a substitution instance of) the expression `lhs` can be replaced by the expression `rhs` ; `$f` denotes a variable that can match any FL function; and (following [AWW90]) `expr_safe` denotes a function valued expression that is guaranteed not to produce an error when it is applied.]

`id_elim`	`id` $\circ$ `$f` $\Longrightarrow$ `$f`
`ind_def`	`ind` $\Longrightarrow$ `s1`
`intsto_elim`	`intsto_safe` $\Longrightarrow$ `for:`$\langle$`ind, id`$\rangle$
`reduce_intro`	`*_safe` $\circ$ `for:`$\langle$`$f_safe, $1`$\rangle$ $\Longrightarrow$ `reduce:`$\langle(*\circ$ `[s1, $f` $\circ$ `s2])_safe, ~1, $1`$\rangle$
`signal_collapse`	`$f` $\circ$ `signal` $\Longrightarrow$ `signal`
`cond_dist_in`	`$f` $\circ$ (`$p` $\rightarrow$ `$g`$_1$; `$g`$_2$) $\Longrightarrow$ (`$p` $\rightarrow$ `$f` $\circ$ `$g`$_1$; `$f` $\circ$ `$g`$_2$)

[The full paper will contain a discussion of these rules. Very briefly, the `id_elim` rule is valid because the identity function `id` can be eliminated from any calculation. The `ind_def` rule is valid because `ind` is just a mnemonic alias for `s1` within a `reduce` or `for`. The `intsto_elim` rule is used to rewrite `intsto` as a for loop that generates the sequence $\langle$`1`$,\ldots,$`n`$\rangle$. The `reduce_intro` rule is valid because multiplication over a sequence can be performed by

initializing an accumulator to 1 and then multiplying the accumulator successively by each element in the sequence. The `signal_collapse` rule is valid because all functions are strict with respect to errors and thus can be deleted when applied to values that are guaranteed to be errors. The `cond_dist_in` rule is valid because any function composed with a conditional ca be distributed into the arms of the conditional.]

3 Optimizing factorial

This section traces the rewriting that occurs when the compiler is applied to the FL version of factorial. [Description abbreviated for this abstract.] In each of the following lines, the redex is shown underlined, and between each pair of lines, the matching rule is identified.

$$\texttt{isposint} \rightarrow * \circ \underline{\texttt{intsto}}; \texttt{signal} \circ [\tilde{}\texttt{"fact"}, \tilde{}\texttt{"arg1"}, \texttt{id}]$$

using the intsto-elim rule $\texttt{intsto\_safe} \Longrightarrow \texttt{for:}\langle \texttt{ind}, \texttt{id} \rangle$

$$= \texttt{isposint} \rightarrow * \circ \texttt{for:}\langle \underline{\texttt{ind}}, \texttt{id} \rangle; \texttt{signal} \circ [\tilde{}\texttt{"fact"}, \tilde{}\texttt{"arg1"}, \texttt{id}]$$

using the ind-def rule $\texttt{ind} \Longrightarrow \texttt{s1}$

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

using the reduce-intro rule

$$*\texttt{\_safe} \circ \texttt{for:}\langle \$\texttt{f\_safe}, \$\texttt{1} \rangle \Longrightarrow \texttt{reduce:}\langle (* \circ [\texttt{s1}, \$\texttt{f} \circ \texttt{s2}])\texttt{\_safe}, \tilde{}\texttt{1}, \$\texttt{1} \rangle \text{ with}$$

$$\$\texttt{f} = \texttt{s1}$$

$$\$\texttt{1} = \texttt{id}$$

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

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

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

[This abstract omits a detailed description of the program. Briefly, 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              mxpx2xn
 aa:(aa:___)
  trans                mxp x nx2
  aa:*                 mxp x n
  +                    mxp                    ]
```

4.2 The transformed program

```
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])
```

5 Status and future work

[Deleted from this abstract.]

6 Acknowledgements

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

References

[AWW90] 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.* Research Report RJ 7804, IBM, 1990 (to appear in POPL 1991).

[Bac78] 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.

[Bac85] 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.

[Mur90] 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).