Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-93-51

1993

FrIL - A Fractal Intermediate Language

Ron Cytron and David Shields

This document describes the motivation, language description, and experience using FrIL, an intermediate language for a compiler's "middle-end." FrIL has subbessfully supported a two-semester compiler construction sequence, where the first semester included code generation from a C-like language and the second semester included advanced data flow analysis and program transformation.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Cytron, Ron and Shields, David, "FrIL - A Fractal Intermediate Language" Report Number: WUCS-93-51 (1993). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/546

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

FRIL–A Fractal Intermediate Language

Ron Cytron and David Shields

WUCS-93-51

October 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

FRIL—A Fractal Intermediate Language

Ron Cytron*
Washington University
Campus Box 1045
St. Louis, Missouri 63130

David Shields
IBM Research
P.O. Box 704
Yorktown Heights, NY 10598

July 15, 1993

Abstract

This document describes the motivation, language description, and experience using FRIL, an intermediate language for a compiler's "middle-end". FRIL has successfully supported a two-semester compiler construction sequence, where the first semester included code generation from a C-like language and the second semester included advanced data flow analysis and program transformation.

1 Introduction

As evidenced by this issue's table of contents, there is no lack of credible opinion concerning the nature and content of a good IL (intermediate language). Compiler writers have long wrestled with defining an endearing and enduring IL, yet only now do we see an effort to bring various ILs out of their closets and into scrutiny.

Given the plethora of programming languages and computer instruction sets, it's remarkable how few ILs have an associated *architecture*: an exposed language definition, a contractual agreement between the front- and back-ends of a compiler, a reliable and stable interface. As shown in Figure 1, an architected IL carries the promise of multiple front-ends interfacing with multiple back-ends, offering substantial reuse of middle-end optimizing technology. Standards committees can spend a decade settling on extensions to well-established languages; a machine architecture can easily expire in fewer years than its vendors spent defining and benchmarking its instruction set. And yet the mechanism connecting these two worlds—the so-called "middle end" of a compiler—often receives all too little attention. The irony of this situation is evident in Figure 1, from which one predicts that an IL should outlive any given programming language or machine architecture.

Before proceeding to the formal definition of FRIL, it's useful to examine the design considerations that gave birth to FRIL:

*Contact author. phone: (314) 935–7527; e-mail: cytron@cs.wustl.edu

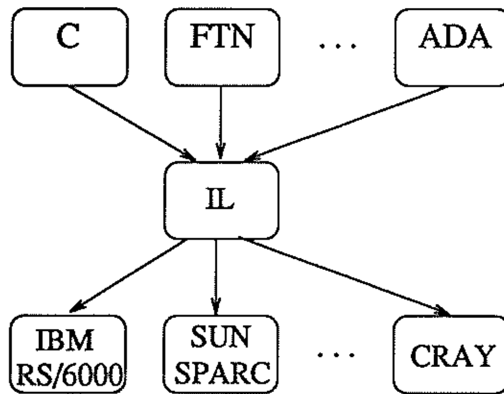

Figure 1. Hypothetically, an IL avoids quadratic work in accommodating multiple languages and multiple architectures

1. **An IL should be a *bona fide* language, and not merely an aggregation of data structures.** With tools such as YACC and LEX at our disposal, we can afford to bestow syntax on our IL, so that its tokens are as readily recognized by humans as by a back-end. Direct encodings of front-end data structures (e.g., a common/equivalence table for FORTRAN) do not constitute a suitable IL.

2. **An IL should have clean and readily apparent semantics.** Where doubtful interpretation exists, trouble will surely follow. For example,

   - An IL should avoid short-circuiting operators like C's "&&"; such control should be explicit in the IL.
   - Evaluation order (or lack thereof) should be explicitly specified in the IL.

3. **The IL representation should not be overly expensive.** Although some expansion is inevitable, one should strive for a terse, meaningful representation, keeping in mind that most optimizing compilers perform many passes over the intermediage language.

4. **The IL should probably not be any particular programming language.** Although persuasive arguments have been made in favor of adopting some programming languages (e.g., ML [App92]), as an IL, the storage model is often inappropriate. Moreover, an IL can demand a level of specificity from its front-ends that would be embarassing to require of humans (e.g., alias relations).

5. **The IL should be sufficiently general to represent important aspects of multiple front-end languages.**

6. **The IL should be sufficiently general to support efficient code generation for multiple back-end architectures.**

7. **The IL should allow easy implementation of machine-independent program optimizations.**

Ideally, an IL has *fractal* characteristics, so that program transformation and optimization can occur at multiple levels. For example, consider the string operation

$$c = concat(a, b)$$

in which strings a and b are concatenated to form string c. There are at least two ways of representing this operation:

High-level approach: The string operation should be preserved intact, so we see a "concat" operation in the IL. This representation allows for efficient data flow summary of the operation's effects and easy relocation of the operation (e.g., out of a loop).

Low-level approach: The string operation should be realized as a loop that forms the resulting string. This representation allows for optimization of the code representing "concat". For example, the loop performing the "concat" may be fusable with nearby loops.

Which representation is best? Conceivably, each representation has its advantages at different stages of program optimization. With a fractal IL (such as FRIL), the operation is initially specified at a high-level. The middle-end optimizer can be unaware of the details of "concat", but can perform code motion and value numbering to move or eliminate the operation, since FRIL will require the explicit specification of side-effects. Subsequently, the front-end that provided "concat" may be asked to *lower* the operation, perhaps exposing an iterative loop whose contents can be further optimized. Stated another way, "concat" is initially provided by the front-end as a run-time procedure, whose call sites are subjected to a first round of optimization. Subsequently, the called procedure is expanded (inlined, open-subroutined) at the call site, and the resulting code undergoes a second round of optimization.

From the above discussion we see that a front-end for FRIL doesn't vanish from the compilation scene after constructing the intermediate representation of its source program. The front-end may be called upon to elaborate high-level operations, ultimately rendering them into the lowest level of FRIL prior to target code generation.

2 Overview

The syntax of a FRIL program will look most familiar to those accustomed to LISP, as language boasts numerous left and right parentheses. This notation carries several advantages:

- A FRIL program is amenable to interpretation, perhaps by defining the appropriate LISP macros.
- The abstract syntax tree for a FRIL program is much closer to its parse tree than is the case for most programming languages.
  - FRIL's prefix form of expression specification allows language constructs to be easily specified without the disambiguating productions typically found in grammars for infix-style languages. The FRIL language is easily parsed bottom-up or top-down.

  - Because FRIL is designed as an IL, it need not have the typical redundant syntactic devices associated with most programming languages to assist in syntactic error diagnosis and recovery.

- A FRIL program requires scant semantic processing. Types have already been checked and converted by the front-end. Moreover, implicit references to storage, normally absent from an IL, are explicitly represented in FRIL. For example, if a procedure call potentially modifies a set of storage "names", then this behavior is stated explicitly (and compactly) in FRIL. Thus, even a high-level fractal contains sufficient information to analyze, optimize, and transform the fractal. When more detail is exposed by elaborating the high-level fractal into lower-level fractals, the associated information can also become more precise.

- Where most languages are replete with constructs of similar purpose (cf. `if-then-else` and the "?" operator in C), FRIL has a rather sparse set of operators. There is a general value-choice operator (**CHOOSE**) and only one construct for altering program flow ($\rightarrow$).

While the keywords of FRIL may appear overly verbose, their printed form is not necessarily the form in which they are stored. Once rendered into a "tokenized" format (perhaps using LEX), the cost of representing a keyword is essentially reduced to one computer word. Storage names not requiring external resolution are similarly represented as small integers.

Throughout this paper, a FRIL construct or operation α is denoted $(\alpha\ldots)$; a list of α constructs is denoted $<\alpha>$–list.

Given the rules stated below for storage allocation and name sharing, a front-end is responsible for translating a source program into a <*CompUnit*>–list. Each (**CompUnit**...) is self-contained, with its own symbol table, alias relation specification, and executable expressions. Each alias relation, executable expression, and (practically every) symbol is given a **CompUnit**-specific identifying integer. Expression 0 in each (**CompUnit**...) is the distinguished *entry* expression for the **CompUnit**.

3 Data types

FRIL supports the following basic data types, but others are easily added:

INT represents the usual integer data type.

CHR represents a single character.

PTR represents an untyped pointer.

FLT represents a floating-point number.

Each symbol declared in FRIL must be statically typed. Also, most operations in FRIL are typed.

```
int a1;
extern int a2;
int one;

void main() {
   int i;

   int factorial(X)
   int X;
   {
     int Y;

     Y = X;
     if (Y > 0) Y*factorial(X-1);
     else one;
   }

   one = 1;
   a1 = factorial(i=5);
   a2 = factorial(3);
}
```

Figure 2. Sample C program.

4 Compilation Unit

In support of separate (independent) compilation, each compilation unit provides its own specification of storage references and executable expressions. Access to the data and code of other compilation units is provided through *external* symbols. Figure 2 contains the source of a C program and Figure 3 shows the corresponding (automatically generated) FRIL program. Each **(CompUnit...)** must include the following constructs in their specified order:

(ExternalName <*qstring*>**)** establishes the quoted string <*qstring*> as the name of this compilation unit. The **CompUnit** named "main" is the distinguished *entry* **CompUnit** of a FRIL program. The only mechanism for invoking (expression 0 of) a **CompUnit** is by invoking the **CompUnit**'s external name.

(SymbolTable...) declares the names and types of all storage symbols.

(AliasTable...) specifies sets of symbols that must or may be aliased with a storage reference.

```
(CompUnit  (ExternalName "main")
  (SymbolTable 6
    (Symbol
      (SymbolID "one")
      (SymbolType INT)
    )
    (Symbol
      (SymbolID "a2")
      (SymbolType INT)
    )
    (Symbol
      (SymbolID "a1")
      (SymbolType INT)
    )
    (Symbol
      (SymbolID 5)    /* i */
      (SymbolType INT)
    )
    (Symbol
      (SymbolID 7)    /* X */
      (SymbolType INT)
    )
    (Symbol
      (SymbolID 8)    /* Y */
      (SymbolType INT)
    )
  )
  (AliasTable 1
    (Alias 1
      (MayAliasSymbols (SymbolID "one"))
    )
  )
  (Expression 0
    (PushLevel 2 (Args ) (Locals ))
    (-> 0 (-> 2) )
  )
  (Expression 2
    (PushLevel 3
      (Args )
      (Locals (SymbolID 5)    /* i */)
    )
    (DefTyped INT (SymbolID "one") 1)
    (DefTyped INT (SymbolID "a1")
      (-> 1
        (DefTyped INT (SymbolID 5)    /* i */
          5
        )
        (HiddenRefs (AliasUse 1))
      )
    )
    (-> 0
      (DefTyped INT (SymbolID "a2")
        (-> 1
          3
          (HiddenRefs (AliasUse 1))
        )
      )
    )
  )
```

```
(Expression 1
  (PushLevel 5
    (LinkExpressionID 2)
    (Args   (SymbolID 7)    /* X */)
    (Locals (SymbolID 8)    /* Y */)
  )
  (DefTyped INT (SymbolID 8)    /* Y */
    (UseTyped INT (SymbolID 7)    /* X */
    )
  )
  (-> 0
    (CHOOSE
      (
        (NE INT
          0
          (GT INT
            (UseTyped INT
              (SymbolID 8)    /* Y */
            )
            0
          )
        )
        (TIMES INT
          (UseTyped INT
            (SymbolID 8)    /* Y */
          )
          (-> 1
            (MINUS INT
              (UseTyped INT
                (SymbolID 7)    /* X */
              )
              1
            )
            (HiddenRefs (AliasUse 1))
          )
        )
      )
      (1 (UseTyped INT (SymbolID "one")))
    )
  )
)
```

Figure 3. FRIL translation.

<*Expression*>–list specifies the executable code. Each expression within a compilation unit is identified by a nonnegative integer. Expression 0 is the *entry expression*, which can only be executed by invoking the (**CompUnit**...)'s external name.

5 Symbol Table

(**SymbolTable** <*num*>...) declares the number of symbols (<*num*>) and contains the <*Symbol*>–list. Note that predeclaring the number of symbols would not be wise for languages in which humans write programs, but such details are quite useful in an IL.

(**Symbol**...) declares a symbol.

(**SymbolID** <*id*>) declares the name of the symbol, where <*id*> is either a positive integer or a quoted string.

(**SymbolType** <*type*>) declares the symbol's type.

There are two forms of a (**SymbolID**...):

Positive Integer: Such symbol names are *private* to the (**CompUnit**...) and thus can only be directly referenced inside the (**CompUnit**...).

Quoted String: Such symbol names are *common* among all (**CompUnit**...)s. For example, each (**CompUnit**...) that references

(**SymbolID** "x")

refers to the same storage name, while

(**SymbolID** 3)

is a different symbol name in different (**CompUnit**...)s.

The explicit association of actual storage with such symbol names is accomplished by a (**Locals**...) or (**Args**...). A symbol can have at most *one* such storage association. If a symbol name is never mentioned in a (**Locals**...) or (**Args**...), then the storage association for that name is *global* (i.e., at level 0 of the display).

6 Alias Table

Each entry of the alias table specifies a pair of sets of symbols $[May, Must]$. When a storage reference is associated with one of these pairs, then the reference *must alias* the symbols in $Must$ and *may alias* the symbols in May.

(**AliasTable** <*num*>...) introduces the alias table with <*num*> entries.

(**Alias** <*id*>... introduces an alias with number <*id*>:

(**MustAliasSymbols** <*SymbolID*>–list) and

(**MayAliasSymbols** <*SymbolID*>–list) declare the aliased symbols.

7 Expressions

The remainder of a **(CompUnit...)** contains the <*Expression*>–list. Each expression declares its expression ID, static link, and storage associations. The rest of an expression contains executable code. An expression finishes by "returning" some value to its caller, as described below.

(Expression <*num*>... introduces expression with nonnegative ID <*num*>.

> **(PushLevel** <*depth*>... declares the static properties and storage associations of this expression. The expression is nested at positive level <*depth*>.
>
> > **(LinkExpressionID** <*outer*>**)** specifies which expression is statically scoped just outside the current expression. If this declaration is missing, then this expression is outermost.
> >
> > **(Args** <*SymbolID*>–list**)** declares a list of symbols whose storage associations are stack locations on entry to this expression.
> >
> > **(Locals** <*SymbolID*>–list**)** declares a list of symbols whose storage assocations are local to this expression.

When an expression is invoked, the runtime *display* is established to provide access to non-local storage. Storage associations are established between the expression's arguments and locals and their allocated storage in the expression's dynamic *stack frame*.

7.1 Transfer of Control

The *invoke* operator (→...) is the only mechanism in FRIL for diverting the flow of expression execution. Constructs involving conditional branching, unconditional branching, iteration, procedure call, or procedure return must inevitably execute an invoke operator. There are three syntactically distinguishable forms for (→...):

(→ <*positive*>...) invokes a particular expression of the currently executing compilation unit. Expressions invoked in this manner are essentially *internal procedures.* A front-end might create such procedures to correspond with actual internal procedures, inline procedures, or code that might be invoked from multiple sources.

(→ <*qstring*>...) invokes expression 0 of the specified compilation unit. Such expressions are typically externally callable functions.

(→ 0...) invokes the *continuation* of the currently executing expression. This mechanism's primary use is to *return* control flow to the expression's caller, with the supplied arguments placed on stack as returned values.

The remaining operands of (→) are *arguments*, evaluated and placed on the execution stack, in reverse order, prior to executing the invoked expression.

7.2 Conditional Execution

The operator **(CHOOSE...)** is the only mechanism in FRIL for selection of alternatives. The syntax is:

(CHOOSE <*choice*>–list),

where each <*choice*> is specified as

$$(<\textit{expression}> <\rho_1> <\rho_2> \ldots <\rho_n>)$$

and executes as follows:

1. The <*expression*> is evaluated.
2. If the result is an integer i, $1 \leq i \leq n$, then this <*choice*> succeeds by executing $<\rho_i>$.
3. Otherwise, the <*choice*> fails and the next <*choice*> is considered.

(CHOOSE...) *must* succeed on some <*choice*>; otherwise, a runtime exception is raised.

Horizontal **(CHOOSE...)** resembles case selection, while vertical **(CHOOSE...)** resembles nested conditional execution.

7.3 Arithmetic and Logical Operations

The format of an arithmetic or logical operation is:

(<*operator*> <*type*> <*operand*>–list)

where

<*operator*> is one of the operators listed below;

<*type*> is a FRIL basic type;

<*operand*>–list are the operands of the operation.

Operator	Description
PLUS	addition
MINUS	subtraction
TIMES	multiplication
DIVIDE	division
REMAINDER	remainder of division
GT	greater-than
GE	greater-than-or-equal
LT	less-than
LE	less-than-or-equal
EQ	equal
NE	not equal
AND	conjunction of two integers
OR	disjunction of two integers

7.4 Input and Output

Basic input and output are accomplished by

(PUT <*char*>**)** writes a single character into the file "output".

(GETC) reads a single character from the file "input".

There is currently no operator for opening, closing, or switching among files, but these could easily be provided.

Higher-level input and output routines have been generated by writing C functions that read and write integers, strings, etc. The resulting FRIL code is essentially a primitive run-time library, that need only be concatenated with other FRIL code passed into the interpreter.

7.5 Storage Operations

(Def <*type*> <*address*> <*value*> <*alias*>**)** is a more general form of (Def...). The expression returns <*value*>, storing the result at <*address*>. The type of <*value*> must agree with <*type*>.

(Use <*type*> <*address*> <*alias*>**)** returns the current value at storage <*address*>, interpreted as <*type*>.

(Storage <*type*>**)** returns the number of storage units associated with <*type*>.

(Alloc <*units*>**)** returns the address of a fresh block of <*units*> storage locations.

(HiddenRefs <*refs*>–list**)** does not explicitly reference storage, but instead declares a set of (unordered) implicit references. Each of the **(refs...)** is of the form:

(AliasDef <*id*>**)** for definitions, or

(AliasUse <*id*>**)** for uses.

where <*id*> is an alias relation.

In the above, <*alias*> is optional, but can be specified as:

(AliasWith <*id*>**)** where <*id*> is an alias relation.

8 Status of the system

First, a disclaimer: in writing this paper, some of FRIL's syntax has been improved from what is currently accepted; the affected software will shortly be upgraded to accept exactly the syntax put forth in this paper. Figure 4 shows the status of development to date. There are actually two front-ends for FRIL:

MIME is the C-like front-end, which produces FRIL exactly as described in this paper.

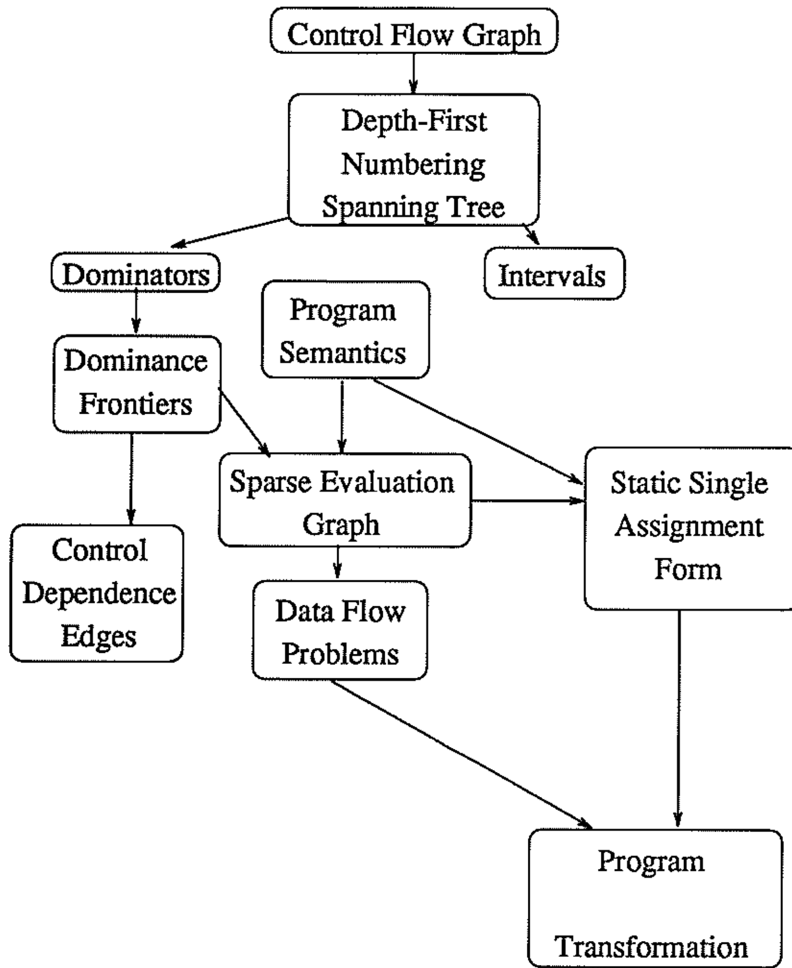

Figure 4. Status of the system.

LEMON is a FORTRAN front-end that produces a control flow graph and program references, but no executable code. This front-end is useful for measuring statistics of real FORTRAN programs. The parser for this front-end is actually the PTRAN system at IBM Research.

The program optimizations implemented to date include

- Constant propagation [WZ91].
- Register allocation [CAC+81].
- Dead code elimination [ASU86].
- Incremental may-alias accommodation [CG93].

A FRIL interpreter can execute a FRIL program, whose behavior is monitored by: construct

(Debug *<level>***)** sets the interpreter trace level to nonnegative *<level>*. Level 0 produces no output; level 9 traces stack and storage activity. Levels 8 – –1 produce increasing amounts of trace information.

9 Conclusions

FRIL has served well as the target of a first course in compiler construction and as a language for program analysis and optimization. The storage model (display levels and SymbolIDs) is sufficiently high-level to allow easy IL generation from languages such as C and PASCAL. The fractal nature of FRIL allows optimizations to proceed on large-grained code before trying to optimize the lower-level code. This approach should allow great efficiency and power in a middle-end optimizer.

Future work might consider the "fractalization" of data as well as code. For example, a symbol name may refer to a complex object composed of other names.

Acknowledgements

The authors thank Fran Allen, whose ideas and discussions on levels of IL greatly influenced this paper.

References

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CG93] Ron Cytron and Reid Gershbein. Efficiently accommodating may-alias information in ssa form. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1993.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.