

January 15, 1988

Annual Fellow Report for 1987

John Backus

This year my report will cover the progress of the Functional Programming project. Last year's report detailed the objectives for the FL language whose manual was completed at the end of last year.

Project personnel

John H. Williams, Edward L. Wimmers, John Backus

Visiting scientists: Prof. John Hughes, Dr. Mary Sheeran, Glasgow University (Summer 1987)

Implementation of FL

During this year John Williams led the development of a Lisp-based implementation of the full FL language. This enables us to translate any FL program into a corresponding Lisp program. We intend to use this program to bootstrap an FL implementation of an FL system. This implementation of the *full FL language* is complete except for some minor changes mandated by recent changes in the language.

During 1988 we plan to begin writing a user-friendly FL compiler in the FL language (this system will provide convenient facilities for writing and running FL programs, but this first version will do very little optimization). It will describe how to translate an FL program into an RT program (with a C program as an intermediate output). We will translate this first into Lisp using the current Lisp implementation, then, using this Lisp version of the new compiler [FL $\rightarrow$ RT] we will input the new compiler again in FL and get out a (not very fast) version of the compiler in RT code. After some testing and experimentation, we will then augment the FL code for the compiler to include more optimization and then recompile to get a fast-executing and optimizing version of the compiler.

The steps of this process can be diagrammed as follows, where a compiler program, written in language M, that translates from language L to language

L' is denoted by the symbol $[L \rightarrow L'](M)$:

<u>Step</u>	<u>Input</u>	<u>Processing compiler</u>	<u>Output</u>
1	$[FL \rightarrow RT](FL)$	$[FL \rightarrow Lisp](Lisp)$	$[FL \rightarrow RT](Lisp)$
2	$[FL \rightarrow RT](FL)$	$[FL \rightarrow RT](Lisp)$	$[FL \rightarrow RT](RT)$
3	$[FL \rightarrow RT](FL)'$	$[FL \rightarrow RT](RT)$	$[FL \rightarrow RT](RT)'$

where $[FL \rightarrow RT](FL)$ is the first $FL \rightarrow RT$ compiler written in FL that does little optimization, and $[FL \rightarrow RT](FL)'$ is the second version with optimization.

The Lisp implementation uses the XBSW parser generator system (modified by John Williams to work under LispVM), an LALR 1 grammar for FL (produced by Ed Wimmers), and semantic routines (written by Williams), for each production in the grammar, that produce the appropriate Lisp code. This code depends on a library of Lisp subroutines upon which the semantic actions are built. Among these subroutines is one for each FL primitive function. These subroutines take account of the way FL objects are represented in Lisp. For example, although the FL primitive `al` is very similar to the Lisp primitive `cons`, the former has one argument $\langle x, seq \rangle$, whereas the latter has two arguments x and `seq`; thus the Lisp code for the FL primitive `al` must take apart its argument and present the two resulting arguments to `cons` in order to produce the result that `al` would have produced.

The Lisp code for the semantic routines and the library subroutines occupies about 2000 lines. The LALR 1 grammar for FL contains about 300 productions.

Language review and modification

In addition to implementation, this year was devoted to a careful review of the language, the addition of several important new features, and the elimination and simplification of a number of other features. The principal changes were (a) the development of a style for defining higher order functions that allowed us to eliminate the most complicated feature of the language, called conditional abstraction; and (b) the development of exceptions (error messages) as semantic entities of the language.

Defining higher order functions

The style for defining higher order functions without using λ -abstraction was developed principally by Prof. John Hughes from Glasgow University, who visited us during the summer. It involves the use of the primitive combining forms curry_n which transform a function f , defined on a sequence of length n , into a curried function $f' = \text{curry}_n:f$ of arity n such that

$$f':x_1:x_2: \dots :x_n = f:\langle x_1,x_2,\dots,x_n \rangle$$

The new style for defining higher order functions is based on the principle of first defining a non-curried version of the desired function and then applying the primitive $\text{curry} = \text{curry}_2$. For example, to define the higher order function cons , where

$$\text{cons}:\langle f_1,\dots,f_n \rangle:x = \langle f_1:x,\dots,f_n:x \rangle,$$

we first define its uncurried version, cons' , where

$$\text{cons}':\langle \langle f_1,\dots,f_n \rangle,x \rangle = \langle f_1:x,\dots,f_n:x \rangle,$$

and

$$\text{Def } \text{cons}' \equiv [\text{seqof}:\text{isfunc},T] \rightarrow \alpha:\text{apply} \circ \text{distr}$$

This definition of cons' checks that its argument is a pair whose first element is a sequence of functions and then pairs its second argument, x , with each of the functions f_i (by distr); it then applies apply to each of these pairs $\langle f_i,x \rangle$ to yield $f_i:x$, giving the desired result for cons' .

Then cons is just $\text{curry}:\text{cons}'$. I.e.,

$$\text{Def } \text{cons} = \text{curry}:([\text{seqof}:\text{isfunc},T] \rightarrow \alpha:\text{apply} \circ \text{distr})$$

If the argument of cons' is not as specified, it (and cons) will produce an exception. Note that the definition of curry is such that its result cannot rewrite until it has two arguments:

$$\text{curry}:f:x:y \text{ rewrites to } f:\langle x,y \rangle$$

but $\text{curry}:f:x$ is in normal form. This property prevents recursively defined higher order functions from "running away" before they actually get applied to their argument.

Exceptions

Exceptions are entities that contain information about applications in which a function is being applied to an inappropriate argument. If x is not appropriate for f (and is not itself an exception), then the result of $f:x$, is an exception that contains x and the name of f . If x is an exception and f uses no exception-recovery primitives, then the result of $f:x$ is just

x ; i.e., all primitives except the exception-recovery ones are strict with respect to exceptions. There are two exception-recovery primitives; one can test for an exception, the other can extract the information contained in an exception.

Having exceptions in the semantic domain of the language presents a difficult problem. We need to report the origin of an exception with respect to the original user's program, whereas the program actually being executed may have been unrecognizably transformed in the process of optimizing it. We believe we have a practical solution to this problem and have the necessary theory for transforming programs without losing the required information for exception reporting. This theory includes means for transforming programs so that they behave identically not only for normal data, but also when exceptions are produced. For example, if a program might produce an exception for some argument, which might report that some function f was applied to some inappropriate argument, then the transform of the program, which might have even eliminated the function f entirely, will produce the same exception with the same report.

Other changes include the following:

New language elements.

Curry combining forms. The combining forms curry_n have been added as primitives. These permit the straightforward definition of higher order functions and allow one to easily provide "curried" functions.

$$\text{curry}_2:f:x:y \equiv f:\langle x,y \rangle$$

Thus $\text{curry}_2:f$ is the curried version of f where f is defined on pairs. For example, if $\text{incr} \equiv \text{curry}_2:+$, then $\text{incr}:5$ is a function that adds 5 to its argument. In general,

$$\text{curry}_n = K^{n-1} \circ 'n \quad ['n \ K:K^{n-1}, K^2:K^{n-2}, \dots, K^{n-1}:K, K^n:\text{id}]$$

where

$K:x:y$ rewrites to x if y is not an exception

$K:x:y$ rewrites to y if y is an exception

$K^n = K \circ K \circ \dots \circ K$ n times

$\circ 'n = \text{lift}^n \circ$

$[^n...] = \text{lift}^n:\text{cons}:<...>$

Truth values. The atoms true and false have been added to represent truth values. Formerly the atoms T and F were used as truth values and could conflict with their use as letters.

The combining form $\Rightarrow$ (function-predicate). The non-computable combining form, function-predicate, written $\Rightarrow$, builds from two predicates p and q, the non-computable predicate $p \Rightarrow q$ that is true for functions that map values satisfying p into those satisfying q. The non-computable predicates built with $\Rightarrow$ can only appear in signature declarations and in assertions. The information they provide will often be useful in type checking.

User defined subtypes. A new declaration, Sub A of B $\leftrightarrow$ P, declares that A is a (user-defined) subtype of the user-defined type B and that A's are to be represented by the underlying type that satisfies both P and the predicate, call it Q, that B's underlying type must satisfy. This new declaration implicitly defines the operations isA, mkA, and umA, just as the normal introduction of a type with a Rep declaration does. Thus isA will be true just for tagged objects $\langle A, x \rangle$ or $\langle B, x \rangle$ for which $P \wedge Q : x$ is true; mkA:x will yield $\langle A, x \rangle$ if $P \wedge Q : x$ is true, otherwise it will produce an exception; and finally umA will yield the second element of a tagged pair for which isA is true.

Where clause modified by a where clause. It is now permitted to have a where clause W acting only on another where clause W' by writing (W' W). In this construction, only the definitions in W' can "see" the definitions in W. This construction is useful in extending the domain of definition of a previously defined function.

Parallel function definitions. The facility for defining several functions simultaneously has been added by the following form of definition:

Def P $\equiv$ E

where P is a pattern and E is an expression in which the function names defined by P occur. This one definition is equivalent to all the following definitions, one for each function name f defined by P:

Def f $\equiv$ (P $\rightarrow$ f) $\circ$ E

The f on the right is the (selector) function defined by the pattern P , since it is in the scope of P , whereas the f on the left is just a function name and is not in the scope of P . For example,

Def $[f.,g.] \equiv [r,s]$
 defines $f \equiv r$ and $g \equiv s$ by the above scheme, since

$$\begin{aligned} \text{Def } f &\equiv ([f.,g.] \rightarrow f) \circ [r,s] && \text{by the above scheme} \\ &\equiv ([T,T] \rightarrow s1) \circ [r,s] && \text{eliminating the pattern} \\ &\equiv [T,T] \circ [r,s] \rightarrow s1 \circ [r,s] && \text{by distributive law for condition} \\ &\equiv r && \text{in the domain for which } r \text{ and } s \text{ are defined} \\ &\equiv \text{exception} && \text{in the domain for which } r \text{ or } s \text{ is not defined} \\ &&& \text{since } [T,T] \circ [r,s] \text{ is true in the domain for which} \\ &&& r \text{ and } s \text{ are defined.} \end{aligned}$$

And similarly, $g \equiv s$ by this parallel definition. Note that this scheme makes f and g defined in exactly the same domain, but this is generally what is intended by making parallel definitions.

Full denotational & operational semantics (without simplification). In the preliminary FL manual, the semantics for a core language was given and that of the full language was given by reducing an expression to a core expression. The "simplification algorithm" that accomplished this became quite complex. Ed Wimmers has now provided simplified denotational and operational semantics for the full language, with the result that the semantics is more direct and much easier to understand. The operational semantics no longer relies on the difficult-to-understand multiple recursion fixpoint function `mrfix`. This new semantics is a major technical accomplishment of Ed Wimmers.

Changed language elements.

Strings. In the preliminary FL manual strings were a separate class of atoms. There were a set of primitive functions on strings: to convert them to sequences of chars, to convert other entities into strings, to find substrings, etc.

In the present language, strings are just sequences of chars, so that all operations on sequences apply to strings. But strings have abbreviations; thus 'abc' abbreviates the sequence $\langle "a,"b,"c \rangle$, where "a is a char;

consequently, for example, $s1:'abc' = 'a'$ and $t1:'abc' = 'bc'$. We may provide some special primitive functions for string manipulation, but this would be just for convenience, since clearly all of them could easily be defined.

Composition of patterns. The preliminary manual allowed the formation of a new pattern by composing two patterns $P_1 \circ P_2$. Now a new pattern can be formed by composition, but only by composing a pattern with an expression $P \circ e$. This change has eliminated a number of complexities.

Language elements that have been eliminated.

Conditional abstraction. Formerly we had the complex construct called conditional abstraction to model and extend the notion of λ -abstraction. This construct complicated the language and its semantics considerably. As mentioned above, our work with our visitor John Hughes from Glasgow University enabled us to develop a style for defining higher order functions without the use of λ -abstraction, a style much more in the spirit of the FL programming approach. Thus we have eliminated conditional abstraction from the language.

Identifiers. Formerly identifiers were a special class of atom with special syntax to distinguish them from function names. We have simply eliminated them; we use strings where they were used.

Templates. Formerly we had two forms of definitions that modelled conventional definitions; in the latter the left side shows the application of the defined function to an argument: $f:\langle x,y \rangle = E(x,y)$. Templates were used as a function level model of an argument of a defined function on the left side of a definition. The intent was to make function level definitions look more like conventional ones.

We found that the special syntax and rules associated with these definitions involving templates were hard to remember and added unnecessary complexity to the language without adding any real power. Without loss of clarity, we now use patterns on the left side of definitions to model arguments; thus we have eliminated the template construct. In place of the former enhanced definitions using templates, we now have enhanced defini-

tions of the following form:

Def $f \leftarrow P \equiv E$

where P is a pattern. It is exactly equivalent to the regular definition

Def $f \equiv P \rightarrow E$

For example,

Def $f \leftarrow [x., y.] \equiv [y, x]$

defines the function f that reverses pairs and is undefined on all non-pair arguments.

Function extensions.

Formerly we had a special declaration for extending the domain of an already defined function. We have found that such extensions can be done without this special declaration and have eliminated it.

Simplification algorithm.

As explained above, Ed Wimmers' new semantics for the full FL language no longer requires the simplification algorithm, so it has been eliminated.

Theory, papers, lectures.

In addition to the work on reviewing and revising the FL language, we also devoted a fair amount of time to some other theoretical studies, to writing papers, and to giving lectures about FL and its properties. One lecture on FL was video-taped by University Video for distribution to 22 universities.

Papers.

A paper by Williams and Wimmers, "Sacrificing simplicity for convenience: Where do you draw the line?" compares the differences between the FL "convenient" approach and the "pure" approach to handling I/O used in a number of other functional languages (the "convenient" approach maintains a chronological "history" of all I/O events, whereas the "pure" approach represents inputs/outputs to each device as an infinite stream of data). This paper was accepted by POPL (January 1988). The problem they discuss is a central one for designers of functional languages. They show that these two fundamentally different approaches to the treatment of I/O have the following properties:

- 1) The pure approach requires a more complex functionality for interactive programs than that required in the convenient approach. (Each "pure" program must have all the device streams to which it refers as explicit arguments and results; in the convenient approach all I/O devices are implicit potential arguments/results of every program.)
- 2) In the pure approach, special analogues of the combining forms are required for building interactive programs (in addition to the regular combining forms for building non-interactive programs, thus, e.g., interactive composition is different from regular composition), whereas in the convenient approach the same combining forms serve for both interactive and non-interactive programs.
- 3) The algebra of interactive programs under the interactive combining forms (in the pure case) and their algebra under the regular combining forms (in the convenient case) are virtually identical. This fact indicates that the convenient approach has lost little of the generality of the pure approach while keeping the convenience implied by (1) and (2).

A second paper, "Full Abstraction and Expressive Completeness for FP" by J. Y. Halpern and E. L. Wimmers, was presented by Wimmers at the Logic in Computer Science Symposium (LICS) in June. It is a theoretical investigation of the question: when does a system have enough primitive functions to ensure the definability of all possible (recursive) functions? This question is of great practical importance; this is indicated by the question Williams raised which began the investigation: could the combining form `apply` to all be defined in FP84? (One would expect that to be easy.) The answer is that it is impossible to define it, along with `concatenate` and `reverse` (this fact arises from the existence in FP84 of sequences that can be infinite on both ends). Among the most interesting results of this paper are the following facts: (1) If `append right` is eliminated from FP84 (giving only one-sided infinite sequences) and `parallel OR` is added, then it has enough primitives to describe all the finite elements of the semantic domain (including all the finite functions), but not all the recursive elements of the domain. (2) If the primitive function `exists` is added, then all recursive elements can be described. (Results (1) and (2) are extensions of known results.) (3) Full FP84 with `parallel OR` added cannot describe all finite domain elements (this

proof is quite new). (4) Full FP84 with the new primitive interp can describe all finite elements.

A third paper, by J. Y. Halpern, J. H. Williams, and E. L. Wimmers, "Completeness of Rewrite Rules and Rewrite Strategies for FP", has been accepted for publication by JACM. This big paper completes a series that investigates questions such as: When are there enough rewrite rules in the operational semantics to rewrite any expression to (a) one corresponding to its meaning (if that is a finite object)? Or (b) to one that has the right form (if the meaning is an infinite sequence or a function)? Or (c) to a sequence of expressions that more and more closely approximate the meaning (if that is not finite)? (These three questions correspond to three notions of *completeness*.) For each notion of completeness, what rewrite strategy will assure that an expression will actually get rewritten as required (and not get caught in a loop, or stop prematurely)? These questions are all answered.

A fourth paper, an extensive revision of parts of the FL language manual, entitled "An introduction to the FL programming language" will be included in a book to be published by the University of Texas containing papers from the lecturers in the Institute of Declarative Programming of UT's Year of Programming. This paper contains a description of most of the changes we have made in the FL language as discussed above.

Video-taped lecture for use at universities.

"Function level programming and the FL language" Video-taped lecture (University Video) to be used in computer science courses at 22 universities (in several courses at some universities), including Carnegie-Mellon, Cornell, Harvard, MIT, Stanford, and U of Texas.
John Backus.

(The preparation and taping of this lecture was very time consuming, since it required high quality graphics prepared on a plotter. Also, the constraints of the medium required a very different style of presentation than that to which I am accustomed.)

External Lectures.

"Introduction to Functional Programming" SHARE Meeting, San Francisco, March 1, 1987, John Williams.

"What does a program mean?" University of California, Los Angeles, March 1987, Ed Wimmers.

"What does a program mean?" Emory University, Atlanta, Georgia, April 1987, Ed Wimmers

"Higher-order functions and I/O in a strict function-level language" WG2.2 Meeting, Sophia-Antipolis, France, June 4, 1987, John Williams.

"Data types in FL" WG2.2 Meeting, Sophia-Antipolis, France, June 4, 1987, John Backus.

"Higher order functions and I/O in a strict functional language" Invited lecture, Institute of Declarative Programming, The Year of Programming (organized by C A R Hoare) at the University of Texas, Austin, Texas, August 28, 1987, John Backus. (Ed Wimmers was also an invited participant.)

"The function level language FL", Tektronix Research Laboratory, Beaverton, Oregon, September 17, 1987, John Williams.

Miscellaneous activities.

Here I shall list only my own activities that have not been covered above.

Report on Software Research and Development in IBM

In January I completed a long study of Research's problems in supporting the development of software in IBM. The resulting report was sent to J. A. Armstrong and R. E. Gomory.

Vacation

In accordance with the Company's wishes, I took 45 days of vacation. This is a fair hunk of time out of my working year.

Internal talks given

"The FL language and function level programming" One of a group of review lectures by K53 members for the Almaden lab., January 30, 1987

"Function level programming" Presentation for visiting French professors, Almaden Research Center, June 25, 1987

"The FL language and function level programming" Presentation for visiting group of Bellcore people, July 16, 1987

Meetings attended

National Academy of Sciences Roundtable meeting, University of San Francisco, January 12, 1987

CSLI (Center for the Study of Language and Information) workshop on natural and computer languages (by invitation), Princeton, California, March 19-21, 1987

Functional Programming and Computer Architecture Conference (Sponsors: IFIP TC2, ACM Sigplan, ACM Sigarch, INRIA), Portland, Oregon, September 14-16, 1987