A Functional Shell

Jon Shultis
Dept. of Computer Science
Univ. of Colorado at Boulder
Boulder, CO 80309

Abstract

One of the best features of the standard UNIX† shell is the use of pipes to compose programs. A C language derivative is used for more complex program combinations involving looping or branching. This paper presents an alternative shell language based on natural extensions of the pipe concept.

"Structured data streams" are introduced as a means of expressing potentially concurrent processing, and "labelled data streams" serve to route data to one of a pool of programs. These complex data streams are hooked together with functional operators much as simple data streams are hooked together with pipes. A generalized notion of "powers" provides for repetition of programs and also for systems that take an arbitrary number of input streams. This provides a uniform way of building complex tools from simple ones, as advocated by Kernighan and Plauger [8]. A major advantage of this new shell language is its program algebra, which facilitates system verification and analysis.

1. Introduction

In the UNIX *C shell* the notation $p_1|p_2$ is used to indicate that the output of program p_1 should be routed to the input of program p_2. The binary operator "|", called a *pipe*, is one of the most useful features of UNIX. If we think of programs as state transition functions, then the pipe operator corresponds to functional composition, which is usually denoted by "$\circ$" in mathematics.

Pipes are useful because they allow a system to be built out of simple combinations of subsystems, as advocated in [8]. The UNIX pipe is in turn made possible by the uniform treatment of files as strings of bytes, having no internal structure of significance to the operating system.

More complicated program combinations are effected by executing a sequence of C shell commands, or by writing a shell *script* that combines programs using a variant of the C programming language. Shell scripts enable the systems programmer to create and destroy temporary files, to handle multiple input and output files, to choose among alternative program sequences, to apply a program to several files, and to repeat program sequences.

Although highly useful, the pipe concept is not always well matched to the C language. In some cases, mismatches are caused by unfortunate restrictions on the use of C language constructs in the shell. For instance, one is not permitted to pipe output to a conditional combination of programs, as in:

$$p_1 \,|\, \textbf{if } b \textbf{ then } p_2 \textbf{ else } p_3 \textbf{ endif}$$

Even with such restrictions removed, however, C is better suited for effecting state transitions by combining assignment statements with control structures than it is for describing the interconnection of programs to form systems.

This paper proposes an alternative to the C shell, called the *F shell*. The F shell takes its cue from the pipe, and is based on a collection of program-combining operators. The F shell operators combine smoothly with pipes, thereby enabling simpler expression of systems.

The F shell is intended to foster structured systems programming by providing a small, highly orthogonal set of powerful primitive program-forming operations (pfo's) that suffice

†UNIX is a Trademark of Bell Laboratories.

for most programming tasks. Of course, all of these operations can readily be defined as extensions to any sufficiently rich language such as ml [6] or Hope [2], just as all of the structured control constructs of von Neumann languages can be simulated with conditional jumps. But the purpose of the F shell's pfo's is to encourage the programmer to use a small set of well-understood programming constructs, so the observation that they introduce no new expressive power is true but irrelevant.

The F shell does have a definition mechanism providing the full power of functional abstraction, but its casual use is discouraged; like the **goto**, it is a powerful but arbitrary and potentially "harmful" feature of the language. It is of course technically possible to supplant abstraction with additional pfo's, but I do not yet see an intuitively appealing way to do it.

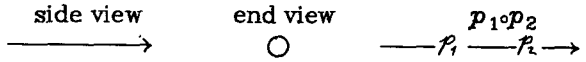
The benefit of writing programs with structured pfo's is that there are simple rules for reasoning about them. In the case of the F shell, the rules express algebraic relationships among programs. The concept of an algebra of programs based on a limited set of pfo's first appeared in a rudimentary form in ISWIM [10] and was further developed in connection with FP [1]. This approach to reasoning about programs is in sharp contrast to most program logics, which express relationships among predicates, states, or modalities which are intended to model the effects of the program's execution on some abstract machine.

The "innovation" of the F shell language, if any, is its integration of the concepts of structured functional programming and types. But the main purpose of the F shell is not language innovation; instead, I wanted to explore one way in which functional programming concepts could be embedded in an operating system, which is fundamentally concerned with state transitions. In this connection it should be noted that I am suggesting only that individual transitions be constructed as pure functions of the state, not that the entire system be described functionally.

The basic plan of this paper is as follows. §2 introduces structured and labelled data streams, which are used to combine simple UNIX-style data streams into "cables" for hooking up complex program combinations. §3 shows how operator polymorphism is used to retain uniformity of notation, beginning with how "|" can be used to compose any two subsystems having similar cable connections. §4 presents the F shell operators, and §5 describes "definitions" and their relationship to scripts. Examples of their use are given in §6. A significant advantage of the F shell is its support for formal reasoning about composite systems, as described in §7. §8 remarks on the programming language origins of the F shell, its implementation (including possibilities for parallelism), and directions for future research. Sections marked with an asterisk can be skipped on a first reading without loss of continuity.

2. Data Streams

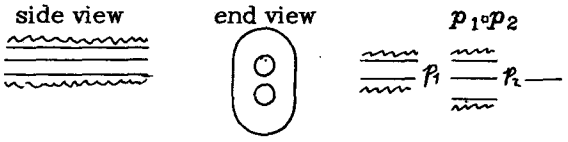
As mentioned earlier, UNIX achieves simplicity of program composition by having a simple and uniform view of files as byte strings. Each program has associated with it a "standard input" and a "standard output". The standard input of one program is connected to the standard output of another using a pipe. Thus data are passed from one program to the next in a *simple data stream*, pictured thus:



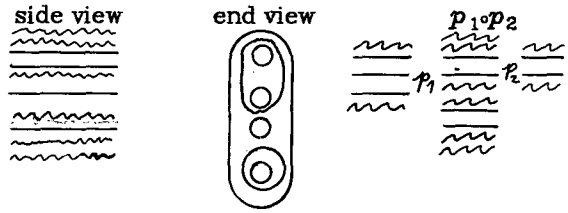
(Note: here and henceforth we shall use ∘ instead of | to denote composition, read from left to right (i.e., in "diagrammatic order"), just like UNIX pipes.)

2.1. Structured Data Streams

A program that has multiple inputs or multiple outputs can be viewed as having a single input stream and a single output stream, each of which is a *structured data stream* having some number of components. Here is a picture of a structured data stream having three simple stream components:



The wiggly line indicates the boundary of a structured data stream. A structured data stream can also have other structured data streams as components, as shown below:



2.2. Labelled Data Streams

It is sometimes convenient to be able to put a label on a data stream. The label on a labelled data stream is used to carry conditional routing information. In drawings, a stream labelled l will be drawn with the label on top:

For simplicity, we shall assume in the sequel that labels are drawn from the set of natural numbers, although in general we would be happy with any enumeration type. The use of

labelled streams will be further clarified in §4.3.

2.3. * Data Stream Semantics

The F shell's data streams are defined by the domain equation

$$\mathbf{stream} \equiv \mathbf{simple\_stream} \oplus \mathbf{stream}^{\otimes^*} \oplus \mathbf{stream}^{\oplus^*}$$

where $\oplus$ is the *separated sum*, $\otimes$ is the *Cartesian product*, and d^{Θ^*} is the domain of arbitrary sums (products) of elements from d, depending on whether Θ is $\oplus$ or $\otimes$. Thus the class of data streams is closed under the formation of (Cartesian) products and (separated) sums. The exact nature of the domain of streams shall not concern us but will, of course, depend on whether the solution is sought in terms of lattices, cpo's, or some other appropriate construction.

3. Polymorphic Operators

Notice that in the above drawings a single operator symbol, viz. "∘", has been used to compose data streams of arbitrarily complex structure. The only requirement is that the structure of the streams being composed is the same.

One way to ensure that the structure of the streams being composed is the same is to have only one kind of stream. The systems programmer can then enjoy the flexibility of working in a unityped or "typeless" command language. Another way to ensure compatibility between "job steps" is to specify the structure of the data streams in minute detail, using "data definition" statements. A third alternative is to use polymorphic typing and type inference, thereby combining the security and added expressivity of a typed system with the flexibility and convenience of a unityped system.

The use of a single operator symbol to denote a class of related operations is generally known as (parametric) *polymorphism*. Such operators are parameterized on both the types and values of their arguments. Minimal constraints on the structure (type) of the arguments to an operator can almost always be inferred from the way the corresponding formal parameters are used in the body of the operator's definition.

As an illustration, the first paragraph of this section can be restated formally as follows. If $p_1{:}t_1{\to}t_2$ and $p_2{:}t_2{\to}t_3$, then $p_1{\circ}p_2{:}t_1{\to}t_3$. So, if we let τ_1, τ_2, and τ_3 be type variables, then we can ascribe a polymorphic type to the composition operator, viz.

$$\circ{:}(\tau_1{\to}\tau_2){\times}(\tau_2{\to}\tau_3){\to}(\tau_1{\to}\tau_3)$$

The initial motivation for having polymorphic operators in the F shell is to enable a single conceptual operation, e.g. composition, to be used to combine programs having an arbitrary number of inputs and outputs without having to specify their structure. But the real benefit of introducing polymorphism is that it gives us the power to write generic systems, i.e. systems that take a number of subsystems as arguments and generate complete systems according to some rule. This increase in expressive power over the C shell is a natural result of introducing the programming language notion of polymorphism into the operating systems arena.

3.1. * Remarks on the semantics of F shell types

The F shell types correspond to Scott's category of restricted equivalence relations on $P\omega$ [15], which is closed under both products and coproducts, and type inference basically proceeds by unification of syntactical terms built from the type operators + × → as in ml [12]. Thus the word "type" as used here is closer to Scott's "functionality"; F shell types are *not* retracts.

An important point to note is that the type operator + is a coproduct in the category of types, whereas the domain operator $\oplus$ is *not* the coproduct in the category of retracts. The connection between these operators will be clarified at the end of the following section.

4. F Shell Operators

The F shell language comprises a collection of operators for combining programs in various ways. These operators fall into four classes: composers, structurers, selectors, and powers. The set of F shell programs is the closure under the F shell operations of a set of primitives.

Every primitive program is a function taking a (possibly labelled) structured data stream as input and producing a (possibly labelled) structured data stream as output. Primitive programs cannot create or modify files (i.e., they have no side effects). Nor can primitive programs read files (i.e., they have no free variables). Consequently, a primitive program is a pure function of its inputs.

4.1. Composers

There are three primitive composers: composition, source, and sink. Composition is the polymorphic extension of the UNIX pipe introduced in §2, and is denoted by the infixed operator symbol "∘". The F shell programs form a category under composition (we assume that there exists a polymorphic primitive program **id** that simply transmits any data stream unchanged). The input stream type of an F shell program is its *domain*, and the output stream type is its *codomain*.

The notation *p<f* in the C shell indicates that program *p* takes its input from the file *f*. In the F shell, the postfixed operator symbol "<" is called the *source* operator. Its effect is

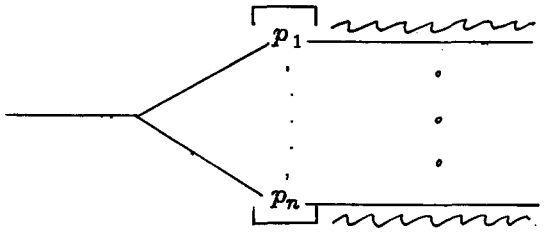
to create a simple data stream from the file *f*. Hence the C shell expression above is recast in the F shell as *f<p* (note the interchange of *f* and *p*). The identifier *f* takes its meaning from a tree-structured environment known as the system *directory*.

Sources supply files as inputs to F shell programs; sinks record the results of F shell programs, updating the system state. The sink operation is denoted by the prefixed operator symbol ">". The effect of *>f* is to record a simple data stream in the file *f*. Hence the expression *f<>g* in the F shell copies the file *f* into the file *g*.

Notice that sources and sinks create and destroy only simple data streams. Were this otherwise, UNIX would have to know something about the internal structure of files.

4.2. Structurers

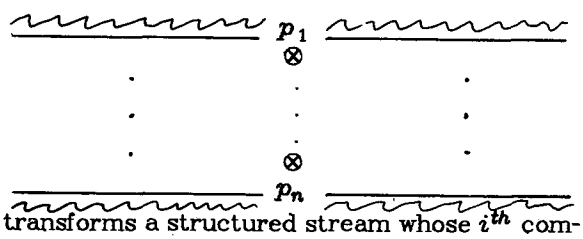
There are three stream structuring operators, or "structurers", called construction, product, and projection. Simple streams are created by the source operator. Structured streams are created by the construction operator. Let $p_1, \cdots, p_n$ be F shell programs having the same domain[†]. Then the program



creates a structured data stream with n components. The i^{th} component stream is obtained by applying p_i to the input stream ($1 \le i \le n$). The input stream and the components of the output stream may be simple, structured, and/or labelled. Notice that a structured stream embracing the contents of several files may be created by a construction of sources: */dev/null<*[f_1<, $\cdots$, f_n<].

Interestingly, there is a crude version of the construction operator in UNIX, called *tee*, that makes n copies of its standard input. This is analogous to the F shell program **[id,...,id]** (the construction of n **id**'s). Although *tee* is intended for "pipe fitting", its output is actually sent to n named files, because there are no operators in the C shell for building programs that act on structured streams.

Programs that act on structured streams are built in the F shell using the binary infixed operator symbol "⊗", called the *product* operator. Let programs p_i have domains δ_i ($1 \le i \le n$). Then the program



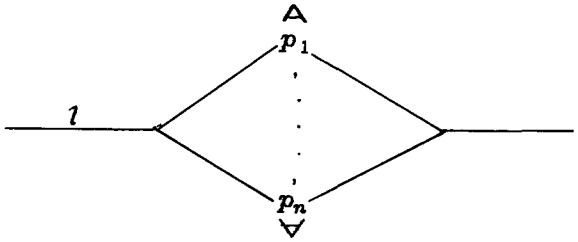
transforms a structured stream whose i^{th} component has type δ_i into a structured stream whose i^{th} component has type κ_i by applying p_i to the corresponding component of the input stream ($1 \le i \le n$).

In order to record the results of F shell programs that act on and produce structured streams, there must be a way to select simple components and sink them. This is done using one of the *projectors* $1^{st}, 2^{nd}, 3^{rd}, \cdots$, which project a structured stream onto its first, second, third, ... component stream. In other words, the i^{th} projector transmits the i^{th} component of a structured stream and discards all other components (effectively sinking them to /dev/null).

4.3. Selectors

There are also three selectors: alternation, sum, and labelling[†]. These operators allow conditional combinations of F shell programs.

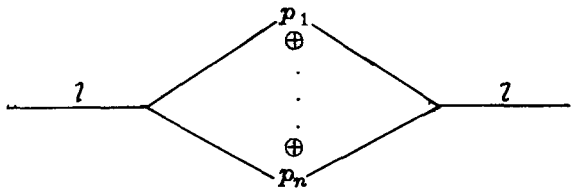
Whereas construction creates structured streams, alternation destroys labelled streams. Let $p_1, \cdots, p_n$ be F shell programs having the "same" domain (in the above sense). Then the program



produces an output stream obtained by applying program p_l to the input stream (with the label l removed). Recall that we assumed that labels were drawn from the set of natural numbers. Clearly, the "syntactic sugar" of labels drawn from arbitrary enumeration types would be beneficial in practice, but would unnecessarily complicate the discussion here.

Sums of labelled streams are analogous to products of structured streams. The former build programs that act on labelled streams, just as the latter build programs that act on structured streams. Let programs p_i have domains δ_i ($1 \le i \le n$). Then the program

[†]More precisely, the constructed function has a polymorphic domain that unifies the domains of all of the component programs. It is the polymorphic type δ solving the system of simultaneous type equations $\delta = \delta_i$ ($1 \le i \le n$).



transforms its input stream to its output stream by applying program p_l, retaining the label on the output stream.

Whereas projectors destroy structured streams, labellers create labelled streams. The labeller $\uparrow^l$ places the label l on its input stream. In general, we permit l to be any program that computes a label value from its input. So for example if b is a program that produces either the label 1 or the label 2, then the F shell program

$$p_1 \circ \uparrow^b \circ \langle p_2, p_3 \rangle$$

corresponds to the outlawed conditional C shell program

p_1 | if b then p_2 else p_3 endif

mentioned in the introduction.

4.4. Powers

Repetition in the F shell is expressed in terms of *powers* of the three binary operators $\circ$, $\otimes$, and $\oplus$. Let ϑ stand for any one of these three. Define

$$f \langle p^{\vartheta 0} \equiv \bot$$

(the everywhere undefined program),

$$f \langle p^{\vartheta 1} \equiv f \langle p, \text{ and}$$

$$f \langle p^{\vartheta i} \equiv f \langle p \vartheta p^{\vartheta i-1} \text{ for } i > 1.$$

So, for example, $p^{\otimes n} \equiv p \otimes \cdots \otimes p$, where there are n occurrences of p on the right.

Given the above, the *iterate* of ϑ, $p^{\vartheta *}$, is defined as follows.

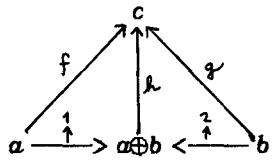
$$p^{\vartheta *} \equiv \bigsqcup_{i \geq 0} p^{\vartheta i}.$$

while loops are used in the C shell for iterated application of a command sequence. The iterate of composition provides the analogous service in the F shell. For example, the C shell program **while** b p **end** corresponds to the F shell program $(\uparrow^b \circ \langle p, \mathbf{id} \rangle)^{\circ *}$.

Other iterates are used to build programs that apply a program to all components of a structured stream, in the case of products, or that apply a program to the data part of a labelled stream, preserving the label, in the case of sums. For example, the C shell program fragment **foreach** *file* (*$*) p $file* **end** could be rendered in the F shell as $p^{\otimes *}$

4.5. * Remarks on the semantics of F shell operators

The F shell algebra treats the structurers and selectors as though they were categorial *duals*, and this demands some explanation. Although the functor $\otimes$ is the categorial product in the category of retracts, the category is not closed under the corresponding coproduct, which produces spaces that look like separated sums shorn of their bottom (and top) elements. The problem with separated sums is that there are as many ways to choose a function h to make the diagram below commute as there are ways to choose a value for $\bot_{a \oplus b}$.



Coalesced sums work fine as long as we restrict ourselves to strict functions, as in FP [1], but if f and g are allowed to be nonstrict then there is *no* function h that will make the diagram commute, because if f is strict but g is not, then

$$\bot_c = f(\bot_a) = h(\uparrow^1(\bot_a)) = h(\bot_{a \oplus b})$$
$$= h(\uparrow^2(\bot_b)) = g(\bot_b) \neq \bot_c,$$

which is a contradiction.

Our problem is that we want both the algebraic tidiness of having dual operations *and* nonstrict functions. This dilemma is solved by ensuring that the *only* way to construct an F shell program $h : a \oplus b \to c$ that makes the diagram commute is to combine programs $f : a \to c$ and $g : b \to c$ with the case tuple operator to form the program $\langle f, g \rangle$. Thus $\oplus$ *is* the categorial dual of $\otimes$ in the category of F shell programs, and a single set of operators and a single algebra can be used to construct and reason about both types and programs.

5. Definitions

In UNIX, any single command line can be given a name, called an *alias*. A list of active aliases is maintained in the environment of each shell. Some combinations of UNIX commands require multiple lines, however, and in that case they must be placed in a *shell script*. Unfortunately, the name space control rules for aliases and scripts are not uniform, which is sometimes awkward.

By design, the operators of the F shell are well integrated so that *any* F shell program can be composed as a single expression. Consequently, a single mechanism for binding identifiers to F shell expressions suffices for all purposes. In its simplest form, an F shell *definition*

Def *name* ≡ *F shell expression*

binds the *name* on the left-hand side to the program denoted by the *F shell expression* on the right-hand side.

As an example the program *distl* takes two

streams and pairs the first up with each component of the second. It can be defined as a combination of the simpler programs **null**, which tests for equality to [] (the structured stream having no components), and **append**, which takes in the structured stream $|x,|y_1,\cdots,y_n||$ and outputs the stream $|x,y_1,\cdots,y_n|$ as follows.

Def *distl* $\equiv$ $\overset{2^{nd}\circ\mathbf{null}}{\uparrow}\circ$

$$\lhd[\,],\ [\mathbf{id}\otimes 1^{st},\ (\mathbf{id}\otimes\mathbf{rest})\circ distl\,]\circ\mathbf{append}\rhd$$

Note that [] is a left unit for **append**.

As in this example, F shell definitions may be recursive. Of course, programs defined in this way are likely to be relatively inefficient, but they can always be recoded and added to the repertoire of primitive programs if their frequency of use justifies the effort.

It is also possible to have parameterized definitions in the F shell, with the syntax

Def (*parameter list*)*name* $\equiv$ *F shell expression*

where the *F shell expression* on the right-hand side may include instances of the identifiers in the parameter list. Parameterized definitions are effectively "generic scripts" - *scenarios* for program combination. For example, programs to apply a binary operator to reduce a structured stream having an arbitrary number of components to a single result stream are most conveniently written using the scenario "/" ("insert").

Def $(f,u)/\ \equiv\ \overset{\mathbf{null}}{\uparrow}\circ\lhd u,\ [1^{st},\mathbf{rest}\circ(f,u)/\,]\circ f\ \rhd$

Some examples of its use are:

Def $\forall\ \equiv (\&,'1')/$

Def $\sum \equiv (+,'0')/$

Def *Doall* $\equiv (\circ,\mathbf{id})/$

The definition of "doall" makes sense because it is permitted to have program streams as well as data streams in the F shell. If p is a defined program, then $p<$ is a program stream source, carrying the denotation of p. In ordinary UNIX shells, a distinction is made between executable and nonexecutable files. In the F shell, however, there are no executable files. The only executable entities are programs, which are either primitive, defined, or F shell expressions. Moreover, the only operations over programs are the F shell program combining operations $\circ$, [ ], $\otimes$, etc. It is crucial for the extensionality of the semantics of the F shell that none of these operators gives one the ability to manipulate the instension of a program. In other words, there is no quotation mechanism in the F shell. Rather, everything is done with higher-order functions.

Although parameterized definitions can be used to define arbitrary programs, they should be reserved for constructing new program forming operations such as "/" in order to keep the number and complexity of the program structuring mechanisms intellectually manageable. Another useful derived pfo is α ("apply-to-all" or "map"):

Def $(f)\alpha \equiv f^{\otimes *}$

The orthogonality of the F shell operators, the uniformity of name space control provided by definitions, the clear distinction between programs and data, and the expressiveness afforded by the use of higher-order functions and scenarios, all contribute in providing a congenial environment to support the development of software in a hierarchical fashion, as advocated for example by Kernighan and Plauger [8].

6. Examples

The examples in this section show how to define a number of common utilities in the F shell.

6.1. Copy

We have already seen the program to copy a file, but here it is as a defined command.

Def (*source*,*target*)*cp* $\equiv$ *source* <> *target*

More interesting programming examples require primitive operations on simple streams. The F shell includes at least the following.

match: **pattern** → (**simple_stream** → **simple_stream** ⊕ **simple_stream**)

concat: **simple_stream** ⊗ **simple_stream** → **simple_stream**

split: **pattern** → (**simple_stream** → **simple_stream** ⊗ **simple_stream**)

replace: **pattern** ⊗ **simple_stream** → (**simple_stream** → **simple_stream**)

The primitive domain of **patterns** is essentially that of SNOBOL4 [7], and the pattern notation used in these examples is based on that of SNOBOL.

6.2. Expand and Print

The following shell script prints a list of files, first expanding tabs to blanks. The default is to expand tabs to 3 blanks.

```
#
if ($1 =~ -[0-9]) then
        set tab = $1
        shift
    else
        set tab = '-3'
    endif
foreach file ($*)
        expand $tab $file | pr
    end
```

The F shell expression corresponding to this

script has an almost identical structure, making it apparent how a sequence of structured commands is converted into a similarly structured composition of expressions.

$1^{st} \circ ('-' ('0..9')any)match \circ label \uparrow \circ \langle \mathbf{id}, ['-3', \mathbf{id}] \rangle \circ$

$distl \circ (expand \circ pr)\alpha$

Notes:

The primitive function *label* is used to convert any function whose range is a sum type into the corresponding predicate. *'string'* denotes a source constant, i.e. it creates a simple stream consisting of the characters in *string*.

6.3. Breaking a simple stream into lines

Filtering is a useful concept in program design, and is used in many UNIX utilities. Filter programs are those that read a portion of their input, perform a local transformation on it, and pass the result to the output stream. In the F shell, this paradigm is adopted in a somewhat modified form, the difference being that the output stream is a structured stream whose components are the transformed fragments of the input stream. As an example, the following program uses filtering to break a simple stream into individual lines.

Def *lines* $\equiv [\mathbf{id}, (('<CR>')break)split \circ$

$((((('<CR>')notany)break)split \circ 2^{nd}) \otimes \mathbf{id})] \circ$

$filter$

The *filter* program takes a simple stream and a command and applies the command to the stream until the stream is empty. The result of filtering is a structured stream whose components are the results of the successive applications of the command to the input stream.

Def $filter \equiv 1^{st} \circ \mathbf{eof} \uparrow \circ$

$\langle [], [\mathbf{apply}, 2^{nd}] \circ$

$[1^{st} \circ 1^{st}, (2^{nd} \otimes \mathbf{id}) \circ filter] \circ \mathbf{append}$

$\rangle$

eof tests for equality to the empty stream $\varphi <$ (φ is shorthand for */dev/null*). The filtered stream can easily be reassembled into a simple stream using the *concat-all* program below.

Def $concat\_all \equiv (concat, \varphi <)/$

Where / is the "insert" operator defined in §5.

6.4. Quicksort

In this section we develop a sorting utility, based on quicksort. The input to the sort program is assumed to be a structured stream whose components are the records to be sorted. A data file can be broken up into records using a variant of the *lines* program.

The sorting problem then breaks down neatly into two phases. The first phase arranges the records in order in a nested structured stream, and the second phase removes the superfluous structure.

Def $sort \equiv arrange \circ flatten$

The *flatten* program simply extracts the elements from a nested structure without altering their order of appearance, and is generally useful for "traversing" tree-structured data.

Def $flatten \equiv (is\_simple \uparrow \circ \langle \mathbf{id}, flatten \rangle)\alpha$

The *arrange* program is an instance of the classic divide-and-conquor paradigm.

Def $arrange \equiv [\#components, '1'] \circ \leq \uparrow \circ$

$\langle \mathbf{id},$

$partition \circ (arrange \otimes \mathbf{id} \otimes arrange)$

$\rangle$

The *partition* program splits its input into three substreams consisting of those records whose keys are less than, equal to, or greater than a given pivot record. It does this by first pairing each record up with the pivot, then comparing the keys of each pair, and finally splitting the records up into the three "bins" according to the results of the comparison.

Def *partition* $\equiv$

$pivot\_pair \circ compare\_all \circ split\_up$

pivot_pair is easily written using *distl*.

Def $pivot\_pair \equiv [1^{st}, \mathbf{id}] \circ distl$

compare_all extracts the keys from each pair and compares them, and then selects the actual data record to pass on along with its appropriate label.

Def $compare\_all \equiv (key \otimes key) \circ compare \uparrow \circ (2^{nd})^{\oplus 3})\alpha$

split_up first creates the three bins and then cycles through the records, placing each in the appropriate bin.

Def $split\_up \equiv [\mathbf{id}, [\,], [\,], [\,]] \circ$

$(1^{st} \circ null \uparrow \circ \langle 2^{nd}, [1^{st} \circ rest, route\_first] \rangle)^{\circ *}$

Finally, *route_first* deposits a record in the correct bin by passing it through a "decision matrix".

Def $route\_first \equiv (1^{st} \otimes \mathbf{id}) \circ$

$[\langle \mathbf{id}, [], [] \rangle \otimes 1^{st},$

$\langle [], \mathbf{id}, [] \rangle \otimes 2^{nd},$

$\langle [], [], \mathbf{id} \rangle \otimes 3^{rd}] \circ (\mathbf{append})\alpha$

7. System Verification

F shell expressions obey certain algebraic laws. For example, the composition of two products is semantically equivalent to the product of the composition of the components; i.e.,

$$(p_1 \otimes \cdots \otimes p_n) \circ (q_1 \otimes \cdots \otimes q_n) \equiv$$

$$(p_1 \circ q_1) \otimes \cdots \otimes (p_n \circ q_n)$$

A list of such laws is given as an appendix. These laws, in conjunction with laws relating the primitive programs and the fixed point induction rule, are used to reason about F shell programs.

One possible use of the laws is in system optimization. Each law states an equivalence of program schemes. Further schematic equivalences can be derived as theorems, possibly with conditions restricting the instances of the schemes to which an equivalence theorem applies. The equivalences can be used as rewrite rules by giving a preferential weighting to one of the terms, and the rewriting rules used to transform programs for improved performance. (Note: although the programmer can access only the denotation of a program, the command interpreter will certainly perform differently with various expressions of the same function.)

As an example, consider the following F shell program to compute Stirling numbers.

Def *Stirling* $\equiv$ $\overset{[=,('1' \otimes \mathbf{id}) \circ =] \circ \mathbf{or}}{\uparrow}$ $\circ$

$\lhd$ '1',

$[(\mathbf{pred} \otimes \mathbf{pred}) \circ \mathit{Stirling},$

$[(\mathbf{pred} \otimes \mathbf{id}) \circ \mathit{Stirling}, 2^{nd}] \circ \times] \circ +$

$\rhd$

This program is straightforward, given the definition of Stirling numbers, but its time complexity is $\binom{n}{k}$ where n and k are the two arguments. The first step toward a faster algorithm is to recognize that the Stirling program is an instance of the following scheme.

Def $f \equiv \overset{p}{\uparrow} \circ \lhd g, [j_1 \circ f, [j_2 \circ f, k] \circ i] \circ h \rhd$

A simple inductive proof in the F shell algebra (see [16, 9]) establishes that whenever j_1 and j_2 commute and h is strict on its first argument a semantically equivalent program can be produced by making the appropriate substitutions into the scheme below.

Def $f' \equiv [\mathbf{id}, []] \circ w \circ 1^{st}$

where

Def $w \equiv \overset{1^{st} \circ p}{\uparrow} \circ$

$\lhd [1^{st} \circ g],$

$\overset{2^{nd} \circ \mathbf{null}}{\uparrow} \circ \lhd [1^{st}, (j_1 \otimes []) \circ w], \mathbf{id} \rhd \circ$

$[1^{st} \circ k, 2^{nd} \circ 1^{st}, (j_2 \otimes \mathbf{rest}) \circ w] \circ$

$[[[1^{st}, 3^{rd} \circ 1^{st}] \circ i, 2^{nd}] \circ h,$

$3^{rd} \circ \mathbf{rest}] \circ \mathbf{append}$

$\rhd$

When the appropriate substitutions are made, the resulting "optimized" program to compute Stirling numbers has a time complexity of only $O(n + k)$.

Another use for the algebra is in verifying properties of systems. For example, an equational specification for an abstract data type can be construed as a system of simultaneous equations in n program unknowns. A collection of F shell programs purporting to implement the operations of the abstract type can be verified simply by checking the proposed solution algebraically.

For example, the F shell specification of the operations on a stack includes the following equation.

$$[top, pop] \circ push \equiv \overset{stack\_empty}{\uparrow} \circ \lhd 'error', \mathbf{id} \rhd$$

A simple algebraic manipulation shows that this equation is satisfied by the six functions shown below.

Def *create* $\equiv []$

Def *stack_empty* $\equiv [\mathbf{id}, create] \circ =$

Def *stack_error* $\equiv [\mathbf{id}, 'error'] \circ =$

Def *top* $\equiv \overset{[stack\_empty, stack\_error] \circ \mathbf{or}}{\uparrow} \circ$

$\lhd$ *'error'*, 1^{st} $\rhd$

Def *pop* $\equiv \overset{[stack\_empty, stack\_error] \circ \mathbf{or}}{\uparrow} \circ$

$\lhd$ *'error'*, **rest** $\rhd$

Def *push* $\equiv \overset{2^{nd} \circ stack\_error}{\uparrow} \circ \lhd 2^{nd}$, **append** $\rhd$

Often the enabling conditions for transformations can be similarly expressed. For example, the transformation shown above for the Stirling numbers program requires that two equations hold, viz.

$j_1 \circ j_2 \equiv j_2 \circ j_1$

$[\perp, r] \circ h \equiv \perp$

In the case of the Stirling program, the proofs are easy, since + is a primitive function which we know to be strict on both of its arguments, and we can readily establish that

$$(\mathbf{pred} \otimes \mathbf{pred}) \circ (\mathbf{pred} \otimes \mathbf{id}) \equiv (\mathbf{pred} \circ \mathbf{pred}) \otimes (\mathbf{pred} \circ \mathbf{id})$$

$$\equiv (\mathbf{pred} \circ \mathbf{pred}) \otimes (\mathbf{id} \circ \mathbf{pred})$$

$$\equiv (\mathbf{pred} \otimes \mathbf{id}) \circ (\mathbf{pred} \otimes \mathbf{pred})$$

The benefit of the algebra is that it provides a relatively simple way to reason about those aspects of a system that depend only on the functionality of its components. I must point out, however, that there is a limitation to these algebraic program analysis and verification methods. Specifically, there is no way to distinguish among differing intensions of a given function within the algebra. Thus excluded from algebraic analysis are those aspects of a program that depend on its implementation.

8. Remarks

The basic functional style of the F shell was inspired by Backus's FP systems [1]. The primary motivation for using the functional style in the F shell is its support for algebraic reasoning about systems, using the terms of the language directly. The laws of these program algebras are essentially like the "characteristic equivalences" of Landin's ISWIM [10].

The idea of extending the UNIX command language to include other functionals is similar to Raoult and Sethi's work on metalanguages for compiler generation [14]. They added certain *permutators* [3] to express delayed applications, thereby enabling a "direct" style of expression for continuation semantics.

The F shell notation itself is based on the algebraic metalanguage SSL [16]. SSL shares much of its underlying semantic theory and inspiration with Mosses' Abstract Semantic Algebras [13]; in particular, both originate in the work of the ADJ group on algebraic semantics (see, e.g., [5, 17]). The type system of the F shell language, including polymorphic operators and scenarios (generic abstract types), is based on the work of Milner [12] as clarified and generalized by MacQueen and Sethi [11].

A prototype implementation of a pared down version of the F shell has been written by Beverly Rollins in Franz Lisp [4]. Large parts of the F shell language can also be translated into C shell scripts, including all of the iterators. Structured data streams are implemented by aggregates of simple files having the same root name. Components are given numbered filename extensions, and labels are also coded in filename extensions. Data streams are passed through the temporary file system. The components of an output stream are given the final extension .o, and composition simply renames all .o files to have final extensions of .i, which is where the next program looks for its inputs. Files that are the targets of sinks have all permissions stripped between their creation and the end of the current program, to enforce the side-effect freedom of the F shell. A serious limitation to this method of implementation is the maximum length of a filename under UNIX.

Some aspects of the F shell language, such as recursion, scenarios, and type checking, are more difficult to implement in the C shell, which has neither higher-order functions nor an adequate quotation mechanism which could be used to simulate higher-order functions in the manner of LISP. A complete prototype, implementing the full command language described in this report, along with automation of the F shell algebra, is being developed in Edinburgh LCF [6]. Future plans include development of a complete F shell system for a personal workstation.

One final point about implementation is that much of the processing of structured data streams can be done in parallel. In particular, the component programs of a product can all be run as independent processes. A planned local network in the Computer Science Department at CU Boulder will enable us to conduct experiments with distributed processing of F shell programs and optimizations dealing with processor/communications/speed tradeoffs.

The F shell language as it stands suffers from the notational curse known as *functio illegibilis*. Encoding the operator symbols in a limited character set such as ASCII only makes matters worse. The problem is that the language operators describe the flow of data, which is better presented using graphs than strings of characters. A system for the display, editing, and transformation of F shell programs using color graphics is well underway. The drawings in this paper are crude approximations of the display format being developed.

Other future research directions include data type representation optimizations, verification and analysis tools, and the use of the F shell for the specification, rapid prototyping and incremental development of large software systems.

Acknowledgements

I would like to thank Eugene Rollins for a number of critical remarks that have led to improvements in the F shell language. The development of the F shell is being funded in part by a University of Colorado Early Career Development Award.

References

1. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Comm. ACM* **21**(8) pp. 613-641 (Aug. 1978).
2. R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "HOPE: An Experimental Applicative Language," *Proc. 1980 LISP Conference*, pp. 136-143 (August 1980).
3. H. B. Curry and R. Feys, *Combinatory Logic*, North-Holland, Amsterdam (1968).
4. J. K. Foderaro, "FRANZ LISP Manual, Version 1," Berkeley UNIX manual, v.2 ().
5. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial Algebra Semantics and Continuous Algebras," *JACM* **24**(1) pp. 68-95 (Jan. 1977).
6. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, "Edinburgh LCF," in *Lecture Notes in Computer Science, no. 78*, Springer-Verlag, Berlin (1979).
7. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language (second edition)*, Prentice-Hall, Englewood Cliffs, NJ (1971).
8. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley (1976).
9. R. B. Kieburtz and J. Shultis, "Transformations of FP Program Schemes," in *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ed. Arvind, J. Dennis,, Portsmouth, NH (Oct. 1981).
10. P. J. Landin, "The Next 700 Programming Languages," *Comm. ACM* **9**(3) pp. 157-166 (March 1966).
11. D. B. MacQueen and R. Sethi, "A Semantic Model of Types for Applicative Languages," mss. for LISP '82 conf. ().
12. R. Milner, "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci.* **17** pp. 348-375 (1978).
13. P. D. Mosses, "Abstract Semantic Algebras!," DAIMI PB-145, Aarhus Univ. (July 1982).
14. J-C. Raoult and R. Sethi, "On Metalanguages for a Compiler Generator: Properties of a Notation for Combining Functions," Mss. (July 1981).
15. D. Scott, "Data Types as Lattices," *SIAM J Comput.* **5**(3) pp. 522-587 (Sept. 1976).
16. J. Shultis, "Hierarchical Semantics, Reasoning, and Translation," Ph.D. Dissertation, Dept. of Computer Science, State Univ. of New York at Stony Brook (Aug. 1982).
17. J. W. Thatcher, E. G. Wagner, and J. B. Wright, "More on Advice on Structuring Compilers and Proving them Correct," RC 7588 (#32847), IBM T. J. Watson Research Center (April 1979).

Appendix: Laws of F Shell Algebra

I) $f \circ (g \circ h) \equiv (f \circ g) \circ h$

II) $[f_1, \cdots, f_n] \circ i^{th} \equiv f_i \; (1 \le i \le n)$

III) $f_1 \otimes \cdots \otimes f_n \equiv [1^{st} \circ f_1, \cdots, n^{th} \circ f_n]$

IV) $\overset{k}{\uparrow} \circ \langle g_1, \cdots, g_n \rangle \equiv g_k$

V) $f_1 \oplus \cdots \oplus f_n \equiv \langle f_1 \circ \overset{1}{\uparrow}, \cdots, f_n \circ \overset{n}{\uparrow} \rangle$

VI) $\mathbf{append} \circ \overset{\mathbf{null}}{\uparrow} \equiv \mathbf{append} \circ \overset{2}{\uparrow}$

VII) $\mathbf{append} \circ [1^{st}, \mathbf{rest}] \equiv \mathbf{id}$

VIII) $\overset{\mathbf{null}}{\uparrow} \circ \langle \mathbf{id}, [1^{st}, \mathbf{rest}] \circ \mathbf{append} \rangle \equiv \mathbf{id}$

IX) $\overset{p}{\uparrow} \circ \langle f_1, \cdots, \overset{p}{\uparrow} \circ f_k, \cdots, f_n \rangle \equiv$

$\overset{p}{\uparrow} \circ \langle f_1, \cdots, \overset{k}{\uparrow} \circ f_k, \cdots, f_n \rangle$

X) $(f_1 \otimes \cdots \otimes f_n) \circ (g_1 \otimes \cdots \otimes g_n) \equiv$

$(f_1 \circ g_1) \otimes \cdots \otimes (f_n \circ g_n)$

XI) $(f_1 \oplus \cdots \oplus f_n) \circ (g_1 \oplus \cdots \oplus g_n) \equiv$

$(f_1 \circ g_1) \oplus \cdots \oplus (f_n \circ g_n)$

XII) $\langle f_1 \circ g, \cdots, f_n \circ g \rangle \equiv \langle f_1, \cdots, f_n \rangle \circ g$

XIII) if π is a permutation of $\{1, \cdots, n\}$

$(\pi \otimes g) \circ \langle f_1, \cdots, f_n \rangle \equiv$

$\langle g \circ f_{\pi(1)}, \cdots, g \circ f_{\pi(n)} \rangle$

XIV) $\mathbf{id} \circ f \equiv f \circ \mathbf{id} \equiv f$

XV) $f \circ [g_1, \cdots, g_n] \equiv [f \circ g_1, \cdots, f \circ g_n]$

XVI) $[f_1 \circ g_1, \cdots, f_n \circ g_n] \equiv$

$[f_1, \cdots, f_n] \circ (g_1 \otimes \cdots \otimes g_n)$