The FL Project: The Design of a Functional Language

Alexander Aiken
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
email: aiken@cs.berkeley.edu

John H. Williams
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: williams@almaden.ibm.com

Edward L. Wimmers
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: wimmers@almaden.ibm.com

Abstract

FL is the result of an effort to design a practical functional programming language based on Backus' FP. This paper provides an introduction to and critique of the FL language. The language effort is analyzed from several points of view: language design, implementation, and user experiences. Emphasis is placed on the unusual aspects of FL, its strengths and weaknesses, and how FL compares with other functional programming languages.

1 Introduction

In his Turing Award paper, John Backus introduced a simple notation for functional programming called FP [Bac78]. The initial design of FP underwent considerable evolution in subsequent years [Wil82b, Wil82a, Bac85, HWWW85, HWW86, BWW86, WW88, BWW90, HWW90], culminating in the definition of FL [BWW+89]. Since the FL definition in 1989 a substantial implementation effort has been underway at the IBM Almaden Research Center to build an optimizing compiler for and to test the viability of programming in FL.

This paper is about the design of, implementation of, and experience with the FL language. The design of FL has been heavily influenced by FP and the philosophy set forth in [Bac78]. The core of this philosophy is expressed in the opening critique of conventional languages [Bac78]:

> Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more. ... Each new language claims new and fashionable features ... but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Thus the major goal of the FL design is to have a simple, yet flexible and useful, programming language. Simplicity means avoiding any language features beyond the bare minimum needed to write realistic programs. The motivation for such spareness is more than the usual desire to avoid duplication in the language. In addition, there is the belief that a language's semantics ought to be a useful everyday tool for the programmer and that a semantic description is useful to programmers only if it is very simple.

The design of FL has been guided primarily by the desire for a simple and useful semantics. A principle that has been adhered to throughout the project is that at each step of the design process the language should have a full, formal denotational semantics. In fact the relative simplicity of various versions of the language has been judged primarily by the relative simplicity of the denotational semantics.

With FP as a starting point, following these principles has led to a language with many unconventional aspects. For example, FL has no static type system, because such a system would add a layer of complexity to the language definition and make it harder to learn. This point contrasts with other functional languages currently in use, most of which include static type systems as part of the language. Another unusual feature of FL is first-class exceptions. FL has exceptions because it was considered necessary to specify formally what happens in the event of a run-time error, such as division by zero. Most other functional languages do not provide run-time exceptions, choosing instead to leave the behavior of run-time errors unspecified and implementation-dependent. Standard ML [HMT89], however, has an exception mechanism similar in spirit to FL's. Like ML, but unlike Haskell [HWA+88], FL is a strict language.

One desirable feature that is not a design goal of FL is efficiency of compiled code. Not surprisingly, as a result some features of FL are difficult to compile well. The goal of the implementation effort has been to see to what extent the simple, clean semantics of FL can be used as the basis for an optimizing compiler that produces respectable code.

This paper gives an overview of three related topics. The first topic is a description of FL. Section 2 gives a brief, informal description of FL together with numerous examples. This section is intended to acquaint the reader with the syntax and style of FL programs without going into details. Section 3 presents the formal syntax and semantics of FL. Because FL is a small language, this section fairly presents almost everything there is to know about FL. For brevity, however, a few minor features are not discussed; the interested reader is referred to [BWW+89].

The second topic is a critique of FL on the subjects of syntax, types, input/output, exceptions and order of evaluation. FL is very different in each of these respects from most other functional languages. The purpose of Section 4 is to highlight these differences, explain why FL is different, and to assess the impact of these differences.

The third and final topic is an overall assessment of what it is like to program in FL. Many FL applications—including some large applications—have been written and it is now possible to make some judgements about what works and what doesn't work in the language and the implementation. Section 5 relates the lessons learned from FL programming.

2 A Brief Description of FL

This section provides an informal description of FL with examples. In presenting the examples, language constructs are used without giving complete definitions in the hope that seeing a few sample programs will serve to give the flavor of the language without requiring lots of text. Each example consists of a program, an example of its use, and some brief comments if necessary. A formal definition of FL is given in Section 3.

2.1 Function Level Programming

FL is designed to be a programming language in which it is easy to write clear and concise programs. The language design is based on the tenet that clarity is achieved when programs are written at the function level—that is, by putting together existing programs to form new ones, rather than by manipulating objects and then abstracting from those objects to produce programs. This premise is embodied in the name of the language: Function Level. The emphasis on programming at the function level results in programs that have a rich mathematical structure and that may be transformed and optimized according to an underlying algebra of programs.

For example, consider how a program `SumAndProd` , which computes the sum and product of two numbers, is defined in a language based on the lambda calculus. One begins with the objects `x` and `y`, constructs the object $\langle \mathtt{x}+\mathtt{y},\ \mathtt{x}*\mathtt{y}\rangle$ (note that sequences are written $\langle \mathtt{x}_1, ..., \mathtt{x}_n\rangle$), and abstracts with respect to `x` and `y` giving a program:

$$\mathtt{def\ SumAndProd} \equiv \lambda(\mathtt{x},\mathtt{y}).\langle \mathtt{x}+\mathtt{y}, \mathtt{x}*\mathtt{y}\rangle$$

In contrast, in FL one begins with the functions + and $*$ and applies the combining form *construction* (written [ ]). Construction is defined so that for any functions $\mathtt{f}_1, ..., \mathtt{f}_n$, the function $[\mathtt{f}_1, ..., \mathtt{f}_n]$ applied to `x` produces the `n` element sequence whose ith element is $\mathtt{f}_i$ applied to `x`.

$$[\mathtt{f}_1, ..., \mathtt{f}_n]{:}\mathtt{x} = \langle \mathtt{f}_1{:}\mathtt{x}, ..., \mathtt{f}_n{:}\mathtt{x}\rangle$$

(Note that the application of `f` to `x` is written `f:x`.) This results in the function level definition

$$\mathtt{def\ SumAndProd} \equiv [+, *] \tag{1}$$

A simple example of the use of `SumAndProd` is given below. Throughout this section, the $\rightsquigarrow$ relation indicates one or more steps in the reduction of a combinatorial expression to its value.

$$\begin{aligned} & \mathtt{SumAndProd}{:}\langle 2,3\rangle \\ \rightsquigarrow\ & [+,*]{:}\langle 2,3\rangle \\ \rightsquigarrow\ & \langle +{:}\langle 2,3\rangle, *{:}\langle 2,3\rangle\rangle \\ \rightsquigarrow\ & \langle 5,6\rangle \end{aligned}$$

The next example illustrates how composition (written $\circ$) can be used to create new functions from more primitive ones. (In general `si` is the primitive function that selects the ith element of a sequence, and `tl` is the primitive function that returns all but the first element of a sequence.) In the following `second` is defined to be the same as the primitive `s2`.

$$\texttt{def second} \equiv \texttt{s1} \circ \texttt{tl} \tag{2}$$

$$\begin{aligned} & \texttt{second:}\langle 2,3,4\rangle \\ \rightsquigarrow\ & (\texttt{s1} \circ \texttt{tl})\texttt{:}\langle 2,3,4\rangle \\ \rightsquigarrow\ & (\circ\texttt{:}\langle \texttt{s1}, \texttt{tl}\rangle)\texttt{:}\langle 2,3,4\rangle \\ \rightsquigarrow\ & \texttt{s1:}(\texttt{tl:}\langle 2,3,4\rangle) \\ \rightsquigarrow\ & \texttt{s1:}\langle 3,4\rangle \\ \rightsquigarrow\ & 3 \end{aligned}$$

Example (2) shows a simple use of infix notation in expressions. The functional $\circ$ (which is defined so that $(\circ\texttt{:}\langle \texttt{f}, \texttt{g}\rangle)\texttt{:x} = \texttt{f:}(\texttt{g:x})$) is applied to the two arguments `s1` and `tl` . In general, `f op g` means $\texttt{op:}\langle \texttt{f}, \texttt{g}\rangle$.

The following factorial program introduces some new combining forms as well as recursion.

$$\texttt{def fact} \equiv \texttt{id} = \texttt{\~{}0} \rightarrow \texttt{\~{}1;id} * (\texttt{fact} \circ \texttt{sub1}) \tag{3}$$

$$\texttt{fact:5} \rightsquigarrow 120$$

The function `fact` is built by applying the combining forms *composition* ($\circ$), *constant* (˜) and *conditional* to the functions $=, \texttt{id}, *$ and the (recursive) function name `fact` . As noted above, composition is defined so that $(\texttt{f} \circ \texttt{g})\texttt{:x} = \texttt{f:}(\texttt{g:x})$. Constant is defined so that $\texttt{\~{}x:y} = \texttt{x}$ (provided `y` terminates and is not an exception—see Section 3.4), and conditional is defined so that $(\texttt{p} \rightarrow \texttt{f;g})\texttt{:x} = \texttt{f:x}$ if $\texttt{p:x}$ is true, and $\texttt{g:x}$ if $\texttt{p:x}$ is false. Thus, `fact:5` reduces to $5 * (\texttt{fact:4})$ (since $5 = 0$ is false), `fact:4` reduces to $4 * (\texttt{fact:3})$, etc.

One unusual aspect of the "FL style" of programming is that explicit recursion is avoided. Closed-form definitions using primitive recursive functions are preferred, such as:

$$\texttt{def fact} \equiv * \circ \texttt{intsto} \tag{4}$$

$$\begin{aligned} & (* \circ \texttt{intsto})\texttt{:5} \\ \rightsquigarrow\ & *\texttt{:(intsto:5)} \\ \rightsquigarrow\ & *\texttt{:}\langle 1,2,3,4,5\rangle \\ \rightsquigarrow\ & 120 \end{aligned}$$

The function `intsto` (for "integers up to") is a primitive such that $\texttt{intsto:n} = \langle 1, ..., \texttt{n}\rangle$ for a non-negative integer `n`. Note that $*$ maps a sequence of numbers into the product of those numbers (for the empty sequence multiplication is defined so that $*\texttt{:}\langle\rangle = 1$).

There are two main advantages to such a programming style. First, termination proofs become very simple. The primitive recursive combining forms preserve termination; that is, they are guaranteed to terminate if their function arguments are themselves terminating. Second, programs are arguably easier to understand since the inductive structure of the computation is in general simpler and more explicit. The former advantage is useful for an optimizing compiler, because some interesting optimizations of FL programs are valid only if functions are known to terminate for all arguments.

Another example of closed-form programming is the following program to compute the length of a sequence.

$$+ \circ \alpha{:}\tilde{}1 \tag{5}$$

$$\begin{aligned}
& (+ \circ \alpha{:}\tilde{}1){:}\langle a, b, c\rangle \\
\rightsquigarrow\ & +{:}(\alpha{:}\tilde{}1{:}\langle a, b, c\rangle) \\
\rightsquigarrow\ & +{:}\langle \tilde{}1{:}a, \tilde{}1{:}b, \tilde{}1{:}c\rangle \\
\rightsquigarrow\ & +{:}\langle 1, 1, 1\rangle \\
\rightsquigarrow\ & 3
\end{aligned}$$

This program works by simply replacing each element of the sequence by a 1 and then adding them up. The functional α (called *map* in many other languages) is defined so that $(\alpha{:}f){:}\langle x_1, ..., x_n\rangle = \langle f{:}x_1, ..., f{:}x_n\rangle$. Note that ":" associates to the left (i.e., $x{:}y{:}z = (x{:}y){:}z$) so that $\alpha{:}\tilde{}1{:}a$ means $(\alpha{:}\tilde{}1){:}a$.

A fancier example of closed-form programming is the following $\mathcal{O}(n \log n)$ sorting program.

$$\mathtt{def\ sort} \equiv \mathtt{tree}{:}\langle \mathtt{merge}, \langle\rangle\rangle \circ \alpha{:}[\mathtt{id}] \tag{6}$$

$$\begin{aligned}
& \mathtt{sort}{:}\langle 8, 4, 2, 3\rangle \\
\rightsquigarrow\ & (\mathtt{tree}{:}\langle \mathtt{merge}, \langle\rangle\rangle \circ \alpha{:}[\mathtt{id}]){:}\langle 8, 4, 2, 3\rangle \\
\rightsquigarrow\ & \mathtt{tree}{:}\langle \mathtt{merge}, \langle\rangle\rangle{:}(\alpha{:}[\mathtt{id}]{:}\langle 8, 4, 2, 3\rangle) \\
\rightsquigarrow\ & \mathtt{tree}{:}\langle \mathtt{merge}, \langle\rangle\rangle{:}\langle\langle 8\rangle, \langle 4\rangle, \langle 2\rangle, \langle 3\rangle\rangle \\
\rightsquigarrow\ & \mathtt{merge}{:}\langle \mathtt{merge}{:}\langle\langle 8\rangle, \langle 4\rangle\rangle, \mathtt{merge}{:}\langle\langle 2\rangle, \langle 3\rangle\rangle\rangle \\
\rightsquigarrow\ & \mathtt{merge}{:}\langle\langle 4, 8\rangle, \langle 2, 3\rangle\rangle \\
\rightsquigarrow\ & \langle 2, 3, 4, 8\rangle
\end{aligned}$$

The function tree takes a function and an object and produces a function such that $\mathtt{tree}{:}\langle f, z\rangle{:}\langle x_1, ..., x_n\rangle$ rewrites to z if $n = 0$, x_1 if $n = 1$, and $f{:}\langle \mathtt{tree}{:}\langle f, z\rangle{:}\langle x_1, ..., x_m\rangle, \mathtt{tree}{:}\langle f, z\rangle{:}\langle x_{m+1}, ..., x_n\rangle\rangle$ if $n > 1$ and $m = \lceil n/2 \rceil$. The function merge merges two sorted sequences of numbers into a single sorted sequence; e.g., $\mathtt{merge}{:}\langle\langle 3, 6\rangle, \langle 1, 3, 5\rangle\rangle = \langle 1, 3, 3, 5, 6\rangle$. Notice that $\alpha{:}[\mathtt{id}]$ maps an n-element sequence into a sequence of n singleton sequences. This ensures that merge is applied to a pair of sequences in the base case.

2.2 Higher-Order Functions

FL programs may be higher-order. In addition to the primitive combining forms mentioned so far, the language includes mechanisms for defining new higher-order functions.

There is a primitive function `lift` for raising functions to functionals. The definition of `lift` is

$$\texttt{lift:f:}\langle g_1, \ldots, g_n\rangle = \texttt{f} \circ [g_1, \ldots, g_n]$$

`lift` is useful for writing closed-form, combinatorial definitions of higher-order functions. In fact, `lift` is so useful that it has a special notation $\texttt{f}' = \texttt{lift:f}$.

Lifting can be used to convert a first-order function into a combining form. For example, `cat` is a function that concatenates sequences together. Thus, `cat'` is a functional that combines sequence-valued functions into a new sequence-valued function. An FL *string* `"xyz"` stands for a sequence of characters $\langle \texttt{`x}, \texttt{`y}, \texttt{`z}\rangle$. Consider the following program that appends the string "Hello " to the front of an argument sequence:

$$\texttt{~"Hello " cat' id} \tag{7}$$

$$\begin{aligned}
& (\texttt{~"Hello " cat' id}):\texttt{"World"} \\
\rightsquigarrow\ & \texttt{cat':}\langle \texttt{~"Hello ", id}\rangle:\texttt{"World"} \\
\rightsquigarrow\ & (\texttt{cat} \circ [\texttt{~"Hello ", id}]):\texttt{"World"} \\
\rightsquigarrow\ & \texttt{cat:}\langle \texttt{~"Hello ":"World", id:"World"}\rangle \\
\rightsquigarrow\ & \texttt{cat:}\langle \texttt{"Hello ","World"}\rangle \\
\rightsquigarrow\ & \texttt{"Hello World"}
\end{aligned}$$

Another important way of defining higher-order functions in FL is by *currying*. FL has a primitive currying combinator `C` defined so that

$$\texttt{C:f:x:y} \rightsquigarrow \texttt{f:}\langle \texttt{x, y}\rangle \tag{8}$$

For example, the function `C:+` maps a number `x` to a function that takes a number `y` and returns `x + y`. An increment function can be defined as follows:

$$\texttt{def inc} \equiv \texttt{C:+:1} \tag{9}$$

$$\begin{aligned}
& \texttt{inc:2} \\
\rightsquigarrow\ & \texttt{C:+:1:2} \\
\rightsquigarrow\ & \texttt{+:}\langle 1, 2\rangle \\
\rightsquigarrow\ & 3
\end{aligned}$$

2.3 Scope

FL is a statically scoped language. An FL program consists of an expression together with a list of definitions to be used in evaluating that expression. A definition list entry may itself be a nested definition list that exports only some of its definitions. This provides a convenient hiding mechanism which, together with the facility for programmer defined data types, provides a technique for making encapsulated type definitions (see Section 2.7).

The following program uses a subsidiary definition to compute the average of a sequence of numbers.

$$\begin{array}{l} \texttt{def ave} \equiv \quad + \texttt{/ length where} \\ \qquad\qquad\quad \{\texttt{def length} \equiv + \circ \alpha\texttt{:}\tilde{}\texttt{1}\} \end{array} \tag{10}$$

$$\begin{array}{rl} & \texttt{ave:}\langle \texttt{3,4,8} \rangle \\ \rightsquigarrow & (+ \texttt{/ length})\texttt{:}\langle \texttt{3,4,8} \rangle \\ \rightsquigarrow & /\texttt{:}\langle +\texttt{:}\langle \texttt{3,4,8} \rangle, \texttt{length:}\langle \texttt{3,4,8} \rangle \rangle \\ \rightsquigarrow & /\texttt{:}\langle \texttt{15,3} \rangle \\ \rightsquigarrow & \texttt{5} \end{array}$$

Any expression (in this case the right-hand side of the definition of `ave`) can have an associated environment of function definitions. In this expression, the associated environment is the singleton environment defining `length`. Such definitions are *local*—their scope of definition is limited to the expression to which they are attached.

2.4 Exceptions

Exceptions, including run-time errors, occur in practice in any programming language, so some facility for reporting them is required. For certain applications, such as operating systems that control the execution of other processes, error handling is essential. Errors and error handling are an integral part of FL. FL provides error handling in a purely functional manner.

The domain of FL values is subdivided into the *normal* and the *exception* values. Semantically, exception values are treated differently than normal values. All functions are strict with respect to exceptions, so that $\texttt{f:x} = \texttt{x}$ for all functions `f` and exceptions `x`. Sequences are also strict with respect to exceptions; a sequence collapses to the leftmost exception it contains, so that $\langle \texttt{x,y} \rangle = \texttt{x}$ if `x` is an exception. This behavior is justified by the intended use of errors in FL: errors represent a situation in which something extraordinary has happened and therefore an error should persist until caught or until it escapes from (and becomes the result of) the program.

Exceptions can be raised by primitive functions. For example, the meaning of division by zero is an exception:

$$/\texttt{:}\langle \texttt{1,0} \rangle = \mathit{Exc}(\langle \texttt{"/"}, \texttt{"arg1"}, \langle \texttt{1,0} \rangle \rangle)$$

Recall that a string `"xyz"` stands for the sequence of characters $\langle \texttt{`x}, \texttt{`y}, \texttt{`z} \rangle$. In general, if a primitive function `f` is applied to an inappropriate argument `x`, then `f:x` evaluates to an exception of the form $\mathit{Exc}(\langle \texttt{"f"}, \texttt{"arg1"}, \texttt{x} \rangle)$.

Most programming languages make a distinction between "run-time errors" (such as division by zero) and "compile-time errors" (such as type mismatches). Exclusive of syntax errors, all errors are treated uniformly in FL as exceptions. For example, adding a character and a number is not sensible. In FL, the primitive function + is defined so that it works as expected on sequences of numbers and returns an

addition exception for all other arguments. Thus, using the program in example (10),

$$\begin{aligned}
& \texttt{ave:}\langle 3, \texttt{`a}\rangle \\
\rightsquigarrow\ & (+/\texttt{length})\texttt{:}\langle 3, \texttt{`a}\rangle \\
\rightsquigarrow\ & (/\texttt{:}\langle +\texttt{:}\langle 3, \texttt{`a}\rangle, \texttt{length:}\langle 3, \texttt{`a}\rangle\rangle \\
\rightsquigarrow\ & /\texttt{:}\langle \mathit{Exc}(\langle \texttt{" + "}, \texttt{"arg1"}, \langle 3, \texttt{`a}\rangle\rangle), \texttt{length:}\langle 3, \texttt{`a}\rangle\rangle \\
\rightsquigarrow\ & /\texttt{:}\,\mathit{Exc}(\langle \texttt{" + "}, \texttt{"arg1"}, \langle 3, \texttt{`a}\rangle\rangle) \\
\rightsquigarrow\ & \mathit{Exc}(\langle \texttt{" + "}, \texttt{"arg1"}, \langle 3, \texttt{`a}\rangle\rangle)
\end{aligned}$$

All FL primitives raise exceptions if applied to inappropriate arguments. This makes FL a *secure* language [App93]. An FL program never "dumps core" or otherwise terminates without returning a valid FL value.

There is a special primitive `signal` such that $\texttt{signal:x} = \mathit{Exc}(\texttt{x})$ for any normal value `x`. This provides the programmer with a way to raise exceptions directly. In addition, there is a higher-order primitive function `catch` for catching and handling exceptions. The function `catch` takes as arguments two functions, a *body* and a *handler*. Informally, $\texttt{catch:}\langle \texttt{body, handler}\rangle\texttt{:x}$ evaluates `body:x`. If `body:x` is not an exception, the result is `body:x`. If `body:x` is an exception $\mathit{Exc}(\texttt{y})$, then the result is the `handler` applied to a pair consisting of `x` and the contents `y` of the exception. Note that this description of `catch` is purely functional. The functionality of `catch` is easily understood with an example. A safe division function that returns 0 if division fails can be written as follows.

$$\texttt{def safediv} \equiv \texttt{catch:}\langle /, \texttt{~0}\rangle \tag{11}$$

$$\begin{aligned}
& \texttt{safediv:}\langle 2, 1\rangle \\
\rightsquigarrow\ & \texttt{catch:}\langle /, \texttt{~0}\rangle\texttt{:}\langle 2, 1\rangle \\
\rightsquigarrow\ & 2 \qquad\qquad \text{since } /\texttt{:}\langle 2, 1\rangle \rightsquigarrow 2
\end{aligned}$$

Thus, `safediv` is exactly like / in the normal case. However,

$$\begin{aligned}
& \texttt{safediv:}\langle 2, 0\rangle \\
\rightsquigarrow\ & \texttt{catch:}\langle /, \texttt{~0}\rangle\texttt{:}\langle 2, 0\rangle \\
\rightsquigarrow\ & \texttt{~0:}\langle\langle 2, 0\rangle, \langle \texttt{"/"}, \texttt{"arg1"}, \langle 2, 0\rangle\rangle\rangle \quad \text{since } /\texttt{:}\langle 2, 0\rangle \rightsquigarrow \mathit{Exc}(\langle \texttt{"/"}, \texttt{"arg1"}, \langle 2, 0\rangle\rangle) \\
\rightsquigarrow\ & 0
\end{aligned}$$

A formal definition of `catch` is given in Section 3.4.

2.5 Input/Output

FL is an interactive language. In order to accommodate interactive input and output and to facilitate the writing of programs that require persistent storage, all FL programs are required to map pairs into pairs. The first component of the pair can be any FL value, but the second component is always a *history*: an FL sequence that contains the current value of the file system and the current status of all input and output devices. Thus, the true functionality of any FL program `f` is $\texttt{f:}\langle \texttt{val, history}\rangle \rightarrow \langle \texttt{val}', \texttt{history}'\rangle$. In the examples (1)-(11) the history component has been omitted.

Most primitive functions are completely independent of the history component and leave it unchanged. For example, $\texttt{tl}:\langle\langle 1,2,3\rangle,\texttt{history}\rangle \rightsquigarrow \langle\langle 2,3\rangle,\texttt{history}\rangle$ no matter what the contents of `history` may be. A few of the primitives depend on the contents of `history` but do not change the file system or the status of any device; e.g., `get` retrieves the most recent file system from the `history` . Finally, some primitives both depend on the contents of the history and alter it; e.g., `in` maps $\langle\texttt{dev},\texttt{history}\rangle$ to $\langle\texttt{next\_inp},\texttt{history}'\rangle$, where `next_inp` is the next value from device `dev` and the history is changed to reflect the fact that `dev` has been read.

It is important to note that without the history as part of its argument, a primitive such as `in` could not be a function, since `in:reader` could return different values depending on when it is called. Including the history permits a functional treatment of input/output and file system fetches. Of course, this implies that *all* functions must map pairs to pairs, even programs such as `tl` that don't depend on the history. However, if the history component were to be added explicitly, it would then be possible to treat it like any other object and thus to make multiple copies of it and to apply different operations to the various copies. Such manipulations would not be consistent with the intended use of the history component as representing *the* current status of the I/O devices and the file system.

Therefore, some methodology is required whereby exactly one history is in existence at any given time. This is accomplished in FL by making the history component *implicit*. There is no possibility of FL programs treating the history in a way inconsistent with the methodology, since the history component may be accessed or changed only by built in primitives such as `in` and `out`, and these primitives are guaranteed to preserve the consistency of the history component.

Whenever possible in this paper the history component is not mentioned explicitly. For example, `tl` is described as though it simply mapped values to values. This suppression of the history component except where necessary for understanding a program is the result of an attempt to reintroduce the notion of an underlying state into an otherwise basically functional language in a disciplined way. Since 1986 when implicit I/O was developed, the approach has been generalized by the use of monadic I/O; see Section 4.3.

The following interactive program prompts for two numbers and prints their sum.

```
def adder ≡ answer ∘ + ∘ getnums
    where {
        def answer ≡ out_screen ∘ (~"sum  =  " cat' int2string)
        def getnums ≡ [prompt ∘ ~"Enter a", prompt ∘ ~"Enter b"]
        def prompt ≡ in ∘ ~"keyboard" ∘ out_screen
        def out_screen ≡ out ∘ [~"screen", id]
    }
```

$$
\begin{array}{lll}
& \texttt{adder:"x"} & \\
\rightsquigarrow & (\texttt{answer} \circ + \circ \texttt{getnums}):\texttt{"x"} & \\
\rightsquigarrow & \texttt{answer:}(+:(\texttt{getnums:"x"})) & \\
\rightsquigarrow & \texttt{answer:}(+:\langle \texttt{prompt:"Enter a"}, \texttt{prompt:"Enter b"} \rangle) & \\
\rightsquigarrow & \texttt{answer:}(+:\langle 1, \texttt{prompt:"Enter b"} \rangle) & \texttt{Enter a} \text{ appears on the screen and the user types 1} \\
\rightsquigarrow & \texttt{answer:}(+:\langle 1, 2 \rangle) & \texttt{Enter b} \text{ appears on the screen and the user types 2} \\
\rightsquigarrow & \texttt{answer:}3 & \\
\rightsquigarrow & (\texttt{out\_screen} \circ (\tilde{}\texttt{"sum = "} \texttt{cat'int2string})):3 & \\
\rightsquigarrow & \texttt{out\_screen:}(\texttt{cat:}\langle \tilde{}\texttt{"sum = ":3}, \texttt{int2string:}3 \rangle) & \\
\rightsquigarrow & \texttt{out\_screen:}(\texttt{cat:}\langle \texttt{"sum = "}, \texttt{"3"} \rangle) & \\
\rightsquigarrow & \texttt{out\_screen:"sum = 3"} & \\
\rightsquigarrow & (\texttt{out} \circ [\tilde{}\texttt{"screen"}, \texttt{id}]):\texttt{"sum = 3"} & \\
\rightsquigarrow & \texttt{out:}\langle \texttt{"screen"}, \texttt{"sum = 3"} \rangle & \\
\rightsquigarrow & \langle \texttt{"screen"}, \texttt{"sum = 3"} \rangle & \texttt{sum = 3} \text{ appears on the screen}
\end{array}
$$

2.6 Patterns

The following program is similar to the "safe" divide in example (11). The difference is that if the argument is not appropriate then an informative exception is returned.

$$
\begin{array}{l}
\texttt{def newdiv}_1 \equiv \texttt{ispair} \rightarrow \\
\qquad (((\texttt{isnum} \circ \texttt{s1}) \wedge (\texttt{isnum} \wedge (\texttt{not} \circ \texttt{iszero})) \circ \texttt{s2}) \rightarrow \\
\qquad\qquad \texttt{s1/s2;} \\
\qquad\qquad \texttt{diverr)}; \\
\qquad \texttt{diverr}
\end{array}
$$

$$
\texttt{def diverr} \equiv \texttt{signal} \circ [\tilde{}\texttt{"newdiv"}, \tilde{}\texttt{"arg1"}, \texttt{id}]
$$

This program first checks that its argument is a pair, then that the first element of the pair is a number and the second element of the pair is a non-zero number; if so, it divides the first by the second, otherwise it returns an exception saying that `newdiv` aborted when applied to its first argument.

$$
\begin{array}{rcll}
\texttt{newdiv}_1\texttt{:}\langle 8,4 \rangle & \rightsquigarrow & 2 & \\
\texttt{newdiv}_1\texttt{:}\langle 8,4,2 \rangle & \rightsquigarrow & \mathit{Exc}(\langle \texttt{"newdiv"}, \texttt{"arg1"}, \langle 8,4,2 \rangle \rangle) & \text{since } \langle 8,4,2 \rangle \text{ is not a pair} \\
\texttt{newdiv}_1\texttt{:}\langle 8,0 \rangle & \rightsquigarrow & \mathit{Exc}(\langle \texttt{"newdiv"}, \texttt{"arg1"}, 0 \rangle) & \text{since } \texttt{s2:}\langle 8,0 \rangle \rightsquigarrow 0
\end{array}
$$

Such nested conditionals arise so frequently in practice that FL provides a primitive function `pcons` (predicate construction) with special syntax $[\![...]\!]$. $[\![\texttt{p}_1, ..., \texttt{p}_\texttt{n}]\!] : \langle \texttt{x}_1, ..., \texttt{x}_\texttt{m} \rangle$ is true if $\texttt{n} = \texttt{m}$ and $\texttt{p}_\texttt{i}$:

$\mathtt{x_i}\,\mathtt{istrueforall1} \leq \mathtt{i} \leq \mathtt{n}$, and false otherwise. Thus $\mathtt{newdiv_1}$ can be written more succinctly,

$$\mathtt{def\ newdiv_1} \equiv [\![\mathtt{isnum}, (\mathtt{isnum} \wedge \mathtt{not} \circ \mathtt{iszero})]\!] \rightarrow \mathtt{s1/s2; diverr} \tag{12}$$

Modern functional programming languages permit the naming of arguments to enable more mnemonic expressions than `s1/s2`. The following program is equivalent to $\mathtt{newdiv_1}$ but uses a *pattern* to create function names for the arguments.

```
def newdiv2 ≡ [[n.,m.]] →
                (((isnum ∘ n) ∧ (isnum ∧ not ∘ iszero) ∘ m) →
                        n/m;
                        diverr);
                diverr
```

$\mathtt{newdiv_2}{:}\langle 8,4\rangle \rightsquigarrow 2$

$\mathtt{newdiv_2}{:}\langle 8,4,2\rangle \rightsquigarrow \mathit{Exc}(\langle \text{"newdiv"}, \text{"arg1"}, \langle 8,4,2\rangle\rangle)$ since $\langle 8,4,2\rangle$ fails to match the pattern $[\![\mathtt{n.,m.}]\!]$

$\mathtt{newdiv_2}{:}\langle 8,0\rangle \rightsquigarrow \mathit{Exc}(\langle \text{"newdiv"}, \text{"arg1"}, 0\rangle)$ since $(\mathtt{not} \circ \mathtt{iszero} \circ \mathtt{m}){:}\langle 8,0\rangle \rightsquigarrow \mathtt{false}$

The pattern mechanism is defined so that $\mathtt{newdiv_2}$ is just a notational shorthand for $\mathtt{newdiv_1}$. That is, the pattern $[\![\mathtt{n.,m.}]\!]$ produces a predicate that tests for the structure of the pattern (`ispair` in this case) and also defines the function names indicated by "." (`n` and `m` in this case) to be selector functions that correspond to their places in the structure (in this case `s1` and `s2` respectively). These function definitions are local; their scope is limited to the arms of the conditional.

Notice that patterns are treated as an extension of predicate construction, consistent with the FL design philosophy of adding as few new constructs as possible. The pattern $[\![\mathtt{n.,m.}]\!]$ is actually shorthand for $[\![\mathtt{n.tt, m.tt}]\!]$, where `tt` is the everywhere true predicate; see Section 4.1.3 for a critique of this design. Using the full power of patterns results in the following equivalent version of `newdiv`.

$$\mathtt{def\ newdiv_3} \equiv [\![\mathtt{n.isnum, m.(isnum} \wedge \mathtt{not} \circ \mathtt{iszero})]\!] \rightarrow \mathtt{n/m; diverr} \tag{13}$$

$\mathtt{newdiv_3}{:}\langle 8,4\rangle \rightsquigarrow 2$

$\mathtt{newdiv_3}{:}\langle 8,0\rangle \rightsquigarrow \mathit{Exc}(\langle \text{"newdiv"}, \text{"arg1"}, 0\rangle)$ the pattern fails since $(\mathtt{not} \circ \mathtt{iszero}){:}0 \rightsquigarrow \mathtt{false}$

Again, the pattern mechanism is defined so that $\mathtt{newdiv_3}$ is exactly equivalent to $\mathtt{newdiv_1}$. To match a pattern with embedded predicates, not only must the argument have the proper structure, but its (sub)components must pass the corresponding predicates as well. This ability to embed predicates in patterns greatly increases the usefulness of the pattern matching mechanism for writing terse, easy to read function definitions. Note that the precedence of the infix operators . , $\wedge$, and $\circ$ is such that $\mathtt{m.isnum} \wedge \mathtt{not} \circ \mathtt{iszero}$ means $\mathtt{m.(isnum} \wedge (\mathtt{not} \circ \mathtt{iszero}))$.

The final version of `newdiv` uses a special notation for defining a pattern on the argument of a user-defined function.

$$\mathtt{def\ newdiv_4\ [\![n.isnum, m.(isnum} \wedge \mathtt{not} \circ \mathtt{iszero})]\!] \equiv \mathtt{n/m} \tag{14}$$

$$\begin{aligned}
\texttt{newdiv}_4{:}\langle 8,4\rangle &\rightsquigarrow 2\\
\texttt{newdiv}_4{:}\langle 8,0\rangle &\rightsquigarrow \mathit{Exc}(\langle\texttt{"newdiv"},\texttt{"arg1"},0\rangle)
\end{aligned}$$

The shorthand notation used in $\texttt{newdiv}_4$ is exactly equivalent to $\texttt{newdiv}_3$. In general, in a definition $\texttt{def f pat} \equiv \texttt{e}$, if an argument x fails to match pat then an exception of the form $\mathit{Exc}(\langle\texttt{"f"},\texttt{"arg1"},\texttt{x}\rangle)$ is generated. This notation provides a convenient way for programmers to specify where informative exceptions should be produced.

2.7 Data Types

FL has user-defined data types. New data types are implemented as tagged objects of an underlying representation type. The language is designed so that all such tagged objects are initially in the domain of FL values, thus giving the programmer a convenient, concrete way of thinking about and manipulating data types. Moreover, these tagging operations can be hidden using the scoping mechanism, thereby providing a capability for abstract data type definitions.

In keeping with the function-level programming paradigm, data types are specified by their characteristic functions. The programmer supplies a predicate that is true for values that represent elements of the type and is false for all other values. For example, consider an employee record with two fields, one for an employee's name and one for an employee's department. An FL data type definition for such a record is

$$\texttt{type emp} \equiv [\![\texttt{isstring},\texttt{isstring}]\!] \tag{15}$$

This type definition defines three functions: a constructor mkemp that maps pairs of strings to elements of emp, a destructor unemp that maps elements of emp to pairs of strings, and a predicate isemp that is true for emp values and false for all other values. An element of data type emp with contents x is denoted $\prec \texttt{emp},\texttt{x} \succ$.

$$\begin{aligned}
\texttt{mkemp}{:}\langle\texttt{"Williams"},\texttt{"K53"}\rangle &\rightsquigarrow \prec \texttt{emp},\langle\texttt{"Williams"},\texttt{"K53"}\rangle \succ\\
\texttt{isemp}{:}\langle\texttt{"Williams"},\texttt{"K53"}\rangle &\rightsquigarrow \texttt{false}\\
(\texttt{isemp}\circ\texttt{mkemp}){:}\langle\texttt{"Williams"},\texttt{"K53"}\rangle &\rightsquigarrow \texttt{true}\\
(\texttt{unemp}\circ\texttt{mkemp}){:}\langle\texttt{"Williams"},\texttt{"K53"}\rangle &\rightsquigarrow \langle\texttt{"Williams"},\texttt{"K53"}\rangle\\
\texttt{mkemp}{:}\langle 1,2\rangle &\rightsquigarrow \mathit{Exc}(\langle\texttt{"mkemp"},\texttt{"arg1"},\langle 1,2\rangle\rangle) \quad \text{since } \langle 1,2\rangle \text{ is not a pair of strings}
\end{aligned}$$

To manipulate data types the FL programmer writes functions that work on the representation. For example, selectors for the name and department fields can be defined as follows.

$$\begin{aligned}
&\texttt{def name isemp} \equiv \texttt{s1}\circ\texttt{unemp}\\
&\texttt{def dept isemp} \equiv \texttt{s2}\circ\texttt{unemp}
\end{aligned}$$

$$\begin{aligned}
\texttt{name}{:}\prec \texttt{emp},\langle\texttt{"Williams"},\texttt{"K53"}\rangle \succ &\rightsquigarrow \texttt{"Williams"}\\
\texttt{dept}{:}\prec \texttt{emp},\langle\texttt{"Williams"},\texttt{"K53"}\rangle \succ &\rightsquigarrow \texttt{"K53"}\\
\texttt{dept}{:}\langle 1,2\rangle &\rightsquigarrow \mathit{Exc}(\langle\texttt{"dept"},\texttt{"arg1"},\langle 1,2\rangle\rangle) \quad \text{since } \texttt{isemp}{:}\langle 1,2\rangle \rightsquigarrow \texttt{false}
\end{aligned}$$

Note that in the definitions of `name` and `dept` the predicate `isemp` is used as a guard on the left-hand side. This notation has the same meaning as when a pattern is used on the left-hand side (Section 2.6) except that no function definitions are made.

An alternative definition of the data type `emp` uses a pattern in the type predicate. When a pattern is used to define a data type, the functions defined by the pattern become selector functions for the data type.

$$\texttt{type emp} \equiv [\![\texttt{name.isstring}, \texttt{dept.isstring}]\!] \tag{16}$$

This `type` definition defines `mkemp`, `unemp`, `isemp`, `name`, and `dept` exactly as above.

3 A Formal Definition of FL

This section provides a formal definition of the syntax and semantics of FL. Section 3.1 gives a grammar for the subset of the language used in this paper and provides a brief discussion of the syntax. Section 3.2 describes the FL semantic domain. Finally, Section 3.3 gives a meaning function that assigns elements of the semantic domain to FL programs. Readers less interested in the formal definition of FL may wish to skip ahead to Section 4, which contains a commentary and comparison with other languages, returning to this section only as needed to clarify the subsequent discussion.

3.1 Syntax

An FL grammar is given in Figure 1. The grammar is written using a mostly-conventional BNF syntax; the conventional notations used in Figure 1 are:

nonterminal	nonterminals are in italic font
terminal	terminals are underlined in bold font
::=	definition symbol
|	alternatives

In addition there are non-standard notations convenient for specifying sequences, FL's prime notation, optional syntax, and grouping:

x^*_p	zero or more elements of type x separated by p
x^+_p	one or more elements of type x separated by p
$[^p$, $\rightarrow^p$, $:^p$, $\tilde{}^p$	[, →, :, ~ with zero or more primes
$\lceil \ldots \rceil$	optional item
$\{\ldots\}$	grouping
construct *small italics*	*small italics* is a comment describing "construct"

Uses of $\{\ldots\}$ are potentially ambiguous because curly braces are used both in FL environments and as meta-symbols of the grammar. To emphasize the distinction, uses as FL syntax are written $\underline{\{}\ldots\underline{\}}$, which is consistent with the notation for terminal symbols.

The grammar in Figure 1 is only a subset of the the full FL language. The omissions are three additional productions for more general forms of patterns, one environment operator, one additional form of function definition, a form of lambda abstraction, and approximately a dozen productions for defining *assertions*, a kind of formal comment [BWW+89]. The more general forms of patterns are used in practice, but infrequently. The other omitted features are rarely used.

To complete the definition of FL syntax it is necessary to describe its lexical structure as well as precedence rules for infix notation. FL is unremarkable in both respects, so a brief, informal description should suffice. The lexical units of FL not defined in Figure 1 are numbers, identifiers, and operators. Identifiers are strings over letters, numbers, and some special characters (#, $, /, _). Identifiers begin with an *identifier letter*, which is any of the usual Roman letters plus the special characters. Numbers begin with a number or a prefix + or −. FL has a fixed set of operator names such as $+, -, *, =, \circ, \wedge$, and $\vee$. Except for unary + and − in numbers, operators always stand for themselves and cannot be included in numbers or identifiers.

There are 15 levels of precedence for infix and unary operators in FL. Precedence is fixed and cannot be changed by the programmer. A consequence of this is that all programmer-defined function names have the same precedence when used in infix expressions.

Although simple, the lexical structure and precedence rules of FL have some annoying problems. These problems are discussed as part of a general critique of FL syntax in Section 4.1.

3.2 The Domain

The FL semantic domain is defined in a standard way as the solution of a system of recursive equations. The proof that these equations have a solution is straightforward using standard techniques [Sto77].

Before presenting the domain equations a few definitions are required. The FL domain, denoted $\mathcal{D}_{\mathrm{FL}}$, contains atoms, functions, sequences (written $\langle x_1, \ldots, x_n \rangle$), tagged pairs (written $\prec i, x \succ$ where i is an integer tag), and exceptions (written $Exc(\mathtt{x})$). There is a partial order $\leq$ on the elements of $\mathcal{D}_{\mathrm{FL}}$ such that the following all hold:

$$
\begin{array}{rcll}
\bot & \leq & x & \text{for all } x \\
\langle x_1, \ldots, x_n \rangle & \leq & \langle y_1, \ldots, y_n \rangle \quad \Leftrightarrow \quad x_i \leq y_i & \text{for all } 1 \leq i \leq n \\
Exc(x) & \leq & Exc(y) \quad \Leftrightarrow \quad x \leq y & \\
\prec i, x \succ & \leq & \prec i, y \succ \quad \Leftrightarrow \quad x \leq y & \\
f & \leq & g \quad \Leftrightarrow \quad f : x \leq g : x & \text{for any functions } f \text{ and } g \text{ and for all } x
\end{array}
$$

Let $X \subseteq \mathcal{D}_{\mathrm{FL}}$. An *upper bound* of X is an element y such that $x \leq y$ for all $x \in X$. The *least upper bound* of X, written $\sqcup X$, is the least value y such that $x \leq y$ for all $x \in X$. The set X is *directed* if every finite subset of X has an upper bound in X. A function f is *continuous* if $f : (\sqcup X) = \sqcup_{x \in X} f : x$ for every directed set X.

FL functions map a (*value*, *history*) pair to another (*value*, *history*) pair. A *history* is a potentially infinite sequence of elements of $\mathcal{D}_{\mathrm{FL}}$. The set of *finite histories* consisting of finite sequences of elements

$$
\begin{array}{rcl}
expr & ::= & atom \mid name \mid seq \mid \underset{application}{expr :^{p} expr} \mid (expr) \mid \underset{primed\ expression}{expr^{p}} \mid cond \mid \\
 & & \underset{constant}{{}^{\sim p} expr} \mid \underset{construction}{[^{p} expr^{*}_{,}]} \mid \underset{predicate\ constr}{[\![^{p} expr^{*}_{,}]\!]} \mid expr \underset{infix\ expr}{expr} expr \mid expr\ \underline{\mathbf{where}}\ expr \\
atom & ::= & \text{`}character \mid number \mid \underset{truth\ values}{\underline{\mathbf{true}} \mid \underline{\mathbf{false}}} \\
seq & ::= & \langle expr^{*}_{,} \rangle \mid string \\
string & ::= & \text{"}character^{*}\text{"} \\
cond & ::= & expr \rightarrow^{p} expr[; expr] \mid pat \rightarrow expr[; expr] \\
patlist & ::= & \{pat\_expr_{,}\}^{*}\ \ pat\ \ \{, pat\_expr\}^{*} \\
pat & ::= & \underset{elementary\ pattern}{name.[pat\_expr]} \mid \underset{pat\ construction}{[\![patlist]\!]} \\
pat\_expr & ::= & pat \mid expr \\
env & ::= & \underline{\{}defn^{+}\underline{\}} \mid \underline{\mathbf{export}}(name^{+}_{,})\ \ env \mid \underline{\mathbf{hide}}(name^{+}_{,})\ \ env \mid \ \underline{\mathbf{lib}}(string) \mid \ \underline{\mathbf{PF}}\ \ \mid \\
 & & env\ \ \underline{\mathbf{uses}}\ \ env \mid env\ \ \underline{\mathbf{where}}\ \ env \mid env\ \ \underline{\mathbf{union}}\ \ env \mid \underline{\{}env\underline{\}} \\
defn & ::= & \underline{\mathbf{def}}\ name\ [pat\_expr] \equiv expr \mid \underline{\mathbf{nrdef}}\ name\ [pat\_expr] \equiv expr \mid \underline{\mathbf{type}}\ identifier \equiv pat \\
name & ::= & identifier \mid operator
\end{array}
$$

Figure 1: An FL grammar.

of X is written $X^{<\omega}$. The set of *infinite histories* consisting of infinite sequences of elements of X is written X^{ω}. The ordering on $\mathcal{D}_{\mathrm{FL}}$ is extended to value-history pairs as follows: $(x,h) \leq (x',h')$ if (1) $x = \perp$ and h is a prefix of h', or (2) $x \leq x'$, $|h| = |h'|$, and $h_i \leq h'_i$ for corresponding elements h_i, h'_i of the two histories.

The FL domain includes sequences, tagged pairs (to represent data types), and exceptions. Notations for these three kinds of values are given by the following definitions:

$$\begin{aligned} Seqs(X) &= \{\langle x_1, \ldots, x_n\rangle | x_i \in X \wedge n \geq 0\} \\ Tag(X) &= \{\prec i, x \succ | i \in I \wedge x \in X\} \\ Exc(X) &= \{Exc(x) | x \in X\} \end{aligned}$$

Definition 3.1 The domain $\mathcal{D}_{\mathrm{FL}}$ is defined to be the solution of the following system of equations:

$$\begin{aligned} \mathcal{D}_{\mathrm{FL}} &= \mathcal{D}^{+}_{\mathrm{FL}} \cup \mathcal{E}_{\mathrm{FL}} \\ \mathcal{D}^{+}_{\mathrm{FL}} &= Atoms \cup Seqs(\mathcal{D}^{+}_{\mathrm{FL}}) \cup Tag(\mathcal{D}^{+}_{\mathrm{FL}}) \cup (\mathcal{DH} \rightarrow_{SH} \mathcal{DH}) \\ \mathcal{DH} &= \mathcal{D}_{\mathrm{FL}} \times (\mathcal{D}^{+}_{\mathrm{FL}})^{<\omega} \ \cup \ \{\perp\} \times (\mathcal{D}^{+}_{\mathrm{FL}})^{\omega} \\ \mathcal{E}_{\mathrm{FL}} &= Exc(\mathcal{D}^{+}_{\mathrm{FL}}) \cup \{\perp\} \\ Atoms &= Integers \cup Floats \cup Characters \cup \{true, false\} \end{aligned}$$

The function space $X \rightarrow_{SH} Y$ is the set of continuous, *strict* and *honest* functions from X to Y. A function f is *strict* if $f : (e,h) = (e,h)$ for all $e \in \mathcal{E}_{\mathrm{FL}}$. A function f is *honest* if whenever $f : (x,h) = (x',h')$ then h is a prefix of h'.

The restriction of $\mathcal{D}_{\mathrm{FL}}$ to strict and honest functions is motivated by the following considerations. FL incorporates I/O through an explicit history component that is manipulated by certain primitive functions. This requires that the order-of-evaluation of FL expressions be specified—otherwise, the order of changes to the history would not be well-defined. If program semantics must depend on the order of evaluation, then a strict semantics is preferable to a lazy one, because it is much easier for a programmer to understand and predict the order of events. There are other reasons that FL is strict besides I/O; these are discussed in Section 4.4.

The restriction to honest functions stems from the desire to admit only functions that treat the history component in a manner consistent with the view that the history is a record of all external events. Thus, a history can be extended, but an event, once it occurs, cannot be modified or removed from the history. The honest functions only extend histories.

Finally, $\mathcal{DH}$ defines the set of valid (*value*,*history*) pairs. The structure of $\mathcal{DH}$ is determined by the role of the history component: only an infinite computation (i.e., one whose meaning is $\perp$) can produce an infinite history, and any finite computation can produce only a finite history.

3.3 Semantics

Let the set of all functions names be $\mathcal{N}$. An *assignment* is a mapping $\mathcal{N} \rightarrow \mathcal{D}_{\mathrm{FL}} \cup \{\#\}$. The value # is used to denote a function name that has no binding. The meaning of an FL expression is given by two

functions

$$\begin{aligned}\mu &: \text{Expressions} \times \text{Assignments} \times (\mathcal{D}^+_{\text{FL}})^{<\omega} \to \mathcal{DH}\\ \rho &: \text{Expressions} \times \text{Assignments} \to \mathcal{N} \to \text{Assignments} \cup \{\#\}\end{aligned}$$

Informally, μ computes the meaning denoted by an expression while ρ computes the assignment denoted by an expression; ρ is used to give meaning to environments of FL functions. The functions μ and ρ are meaningful only in the case that the assignment gives a non-# meaning to every free function name in the expression.

3.3.1 The Function μ

The meaning function μ for expressions is given in Figure 2 . New notation used in Figure 2 is described below. A point of potential confusion is that the same notation is used both for FL expressions and semantics in applications $f : x$ and sequences $\langle x_1, \ldots, x_n \rangle$. Since expressions are written with a bold font (i.e., $\mathtt{x}_1\mathtt{:x}_2$) and values with an italic font (i.e. $x_1 : x_2$) the meaning is always clear from context. There is no FL syntax for tagged pairs and exceptions—respectively $\prec i, x \succ$ and $Exc(x)$—so these notations always denote values.

In Figure 2, the notation $\overline{\mathtt{e}}$ is shorthand for `e` **where** `PF`. This notation is used in cases where FL syntax can be "desugared" into applications of primitive functions. To make clear where primitive functions are meant one can write `e` **where** `PF`, which has the effect of binding free function names in `e` to their primitive definitions (the environment `PF` stands for "primitive functions"). This degree of care in the resolution of primitive function names is required because, in contrast with many other languages, FL permits primitive function names to be redefined.

The following proposition gives the definition of FL function application.

Proposition 3.2 There is a function $: \in \mathcal{D}_{\text{FL}} \to \mathcal{DH} \to \mathcal{D}_{\text{FL}}$ satisfying:

$$x : (y,h) = \begin{cases} (x,h) & \text{if } x \in \mathcal{E}_{\text{FL}} \\ (y,h) & \text{if } x \in \mathcal{D}^+_{\text{FL}} \wedge y \in \mathcal{E}_{\text{FL}} \\ (Exc(\langle \texttt{"apply"}, \texttt{"arg1"}, \langle \mathtt{x}, \mathtt{y} \rangle \rangle), h) & \text{if } x \in \mathcal{D}^+_{\text{FL}} \wedge y \in \mathcal{D}^+_{\text{FL}} \wedge x \text{ is not a function} \\ x(y,h) & \text{otherwise} \end{cases}$$

The meaning of : is the same as the meaning of the primitive function `apply`.

One auxiliary function is used in the definition of μ. The function F turns an arbitrary expression $\mathbf{x}$ into a function that applies the meaning of $\mathbf{x}$ to an argument:

$$F(\mathbf{x}, V) = (\lambda(y,h).x : (y,h')) \text{ where } \mu(\mathbf{x}, V, h) = (x, h')$$

Note that F has the effect of delaying the evaluation of `x` until a second argument is supplied. The operation $\oplus$ combines environments of function definitions and is defined in Section 3.3.2.

$$
\begin{array}{rcl}
\mu(\mathbf{a}, V, h) & = & (a, h) \quad \text{if } \mathbf{a} \text{ is an atom} \\
\mu(\mathbf{f}, V, h) & = & (V(\mathbf{f}), h) \quad \text{if } \mathbf{f} \text{ is a function name} \\
\mu(\langle \mathbf{x_1}, \ldots, \mathbf{x_n} \rangle, V, h) & = & \left\{ \begin{array}{ll} (\langle y_1, \ldots, y_n \rangle, h_n) & \text{if } \{y_1, \ldots, y_n\} \cap \mathcal{E}_{\mathrm{FL}} = \emptyset \\ (y_j, h_j) & \text{if } \{y_1, \ldots, y_{j-1}\} \cap \mathcal{E}_{\mathrm{FL}} = \emptyset \text{ and } y_j \in \mathcal{E}_{\mathrm{FL}} \text{ and } j \leq n \end{array} \right. \\
 & & \text{where } \left\{ \begin{array}{rcl} h_0 & = & h \\ (y_i, h_i) & = & \mu(\mathbf{x_i}, V, h_{i-1}) \quad \text{for } i = 1, \ldots, n \end{array} \right. \\
\mu(\text{"}\mathbf{c_1} \ldots \mathbf{c_n}\text{"}, V, h) & = & (\langle `c_1, \ldots, `c_n \rangle, h) \\
\mu((\mathbf{x}), V, h) & = & \mu(\mathbf{x}, V, h) \\
\mu(\mathbf{x_1} :^p \mathbf{x_2}, V, h) & = & \mu(\overline{\mathbf{apply}}^p : \langle \mathbf{x_1}, \mathbf{x_2} \rangle, V, h) \\
\mu(\tilde{}^p \mathbf{x}, V, h) & = & \mu(\overline{\mathbf{K}}^p : \mathbf{x}, V, h) \\
\mu([^p \mathbf{x_1}, \ldots, \mathbf{x_n}], V, h) & = & \mu(\overline{\mathbf{cons}}^p : \langle \mathbf{x_1}, \ldots, \mathbf{x_n} \rangle, V, h) \\
\mu([\![^p \mathbf{x_1}, \ldots, \mathbf{x_n}]\!], V, h) & = & \mu(\overline{\mathbf{pcons}}^p : \langle \mathbf{x_1}, \ldots, \mathbf{x_n} \rangle, V, h) \\
\mu(\mathbf{x}', V, h) & = & \mu(\overline{\mathbf{lift}} : \mathbf{x}, V, h) \\
\mu(\mathbf{p} \rightarrow \mathbf{x_1}; \mathbf{x_2}, V, h) & = & \left\{ \begin{array}{ll} (cond : \langle y_0, y_1, y_2 \rangle, h_2) & \text{if } \{y_0, y_1, y_2\} \cap \mathcal{E}_{\mathrm{FL}} = \emptyset \\ (y_j, h_j) & \text{if } \{y_0, \ldots, y_{j-1}\} \cap \mathcal{E}_{\mathrm{FL}} = \emptyset \text{ and } y_j \in \mathcal{E}_{\mathrm{FL}} \text{ and } j \leq 2 \end{array} \right. \\
 & & \text{where } \left\{ \begin{array}{rcl} (y_0, h_0) & = & \mu(\mathbf{p}, V, h) \\ (y_1, h_1) & = & \mu(\mathbf{x_1}, \rho(\mathbf{p}, V) \oplus V, h_0) \\ (y_2, h_2) & = & \mu(\mathbf{x_2}, \rho(\mathbf{p}, V) \oplus V, h_1) \end{array} \right. \\
\mu(\mathbf{p} \rightarrow^p \mathbf{x}, V, h) & = & \mu(\mathbf{p} \rightarrow^p \mathbf{x}; \overline{\mathbf{signal} \circ [\tilde{}\text{"}\mathbf{cond\ arm}\text{"}, \mathbf{id}]}, V, h) \\
\mu(\mathbf{p} \rightarrow^p \mathbf{x_1}; \mathbf{x_2}, V, h) & = & \mu(\overline{\mathbf{cond}}^p : \langle \mathbf{p}, \mathbf{x_1}, \mathbf{x_2} \rangle, V, h) \\
\mu(\mathbf{x_1} : \mathbf{x_2}, V, h) & = & \left\{ \begin{array}{ll} y_1 : \mu(\mathbf{x_2}, V, h_1) & \text{if } y_1 \notin \mathcal{E}_{\mathrm{FL}} \\ (y_1, h_1) & \text{if } y_1 \in \mathcal{E}_{\mathrm{FL}} \end{array} \right. \\
 & & \text{where } (y_1, h_1) = \mu(\mathbf{x_1}, V, h) \\
\mu(\mathbf{f}.\mathbf{x}, V, h) & = & \mu(\mathbf{x}, V, h) \\
\mu(\mathbf{f}., V, h) & = & (\overline{\mathbf{tt}}, h) \\
\mu(\mathbf{x_1}\ \mathbf{x_2}\ \mathbf{x_3}, V, h) & = & \mu(\mathbf{x_2} : \langle \mathbf{x_1}, \mathbf{x_3} \rangle, V, h) \\
\mu(\mathbf{x}\ \underline{\mathbf{where}}\ \mathbf{E}, V, h) & = & \mu(\mathbf{x}, \rho(\mathbf{E}, V) \oplus V, h) \\
\mu(\mathbf{e}, V, h) & = & \# \quad \text{if none of the above apply}
\end{array}
$$

Figure 2: The function μ.

The rest of this section provides commentary on the rather terse formal definition of μ in Figure 2. The purpose of this commentary is both to explain the formal definition and to highlight some unusual aspects of FL's semantics. Judgements about the merits of the various features are postponed to Section 4.

The function μ gives FL a leftmost-innermost order of evaluation. For example, if a sequence contains an element of $\mathcal{E}_{\mathrm{FL}}$, the sequence collapses to the leftmost element of $\mathcal{E}_{\mathrm{FL}}$ in the sequence. The semantic operator : is defined to be compatible with leftmost-innermost evaluation; thus, $Exc(x) : Exc(y) = Exc(x)$.

FL provides special notation for the combining forms construction $[\ldots]$, predicate construction $[\![\ldots]\!]$, conditional $\rightarrow$, lifting $'$, constant $\tilde{}$, and application : . These notations correspond to the primitive functions `cons`, `pcons`, `cond`, `lift`, `K` and `apply` respectively. The special forms for `cons`, `pcons`, `cond` and `K` are inherited from FP. The new notations for `lift` and `apply` are introduced because these functions are used pervasively in higher-order FL programming.

The introduction of the special syntax for certain function names creates a problem with the prime notation for lifting. Any function name can be lifted using a prime; for example, $\texttt{cons}'\!:\langle \texttt{f}, \texttt{g}\rangle$ applies lifted `cons` to `f` and `g`. However, the prime notation doesn't extend automatically to the special forms. Thus, FL has additional syntax for lifting each of the special forms: $[\ldots]$ becomes $['\ldots]$, $[\![\ldots]\!]$ becomes $[\!['\ldots]\!]$, etc. The semantic function translates this notation into applications of the appropriate primitive functions.

In a conditional $\texttt{p} \rightarrow \texttt{f}; \texttt{g}$, the predicate `p` may be a pattern. The semantics is such that any definitions made by `p` are local to the expressions `f` and `g` (see Figure 2 and Figure 3). Note that `p` may not be a pattern in the expression $\texttt{cond}\!:\langle \texttt{p}, \texttt{f}, \texttt{g}\rangle$. Extending FL to include this case would require giving meaning to sequences containing patterns. This would introduce a new semantic category into the language, since a sequence containing a pattern is neither a simple sequence nor a pattern.

The syntax of FL also forbids any combination of lifting and patterns (see Figure 1). Since the translation of $\texttt{x} \rightarrow^{\texttt{p}} \texttt{y}; \texttt{z}$ should be $\texttt{cond}^{\texttt{p}}\!:\langle \texttt{x}, \texttt{y}, \texttt{z}\rangle$ for consistency, allowing priming and patterns would also require admitting patterns in sequences. Beyond this problem it is not even clear what a pattern in a lifted conditional should mean. (For example, do the defined selectors apply to the first curried argument or the second?)

The definitions specified by patterns are described by the function ρ (see Section 3.3.2). The predicate specified by a pattern is described by μ. For an elementary pattern `f.x`, the predicate is just the function `x`. The predicate after the dot may be omitted (i.e., `f.`) in which case the predicate is the constant true function `tt`. In a pattern construction $[\![\texttt{p}_1, \ldots, \texttt{p}_\texttt{n}]\!]$, the predicate is $\texttt{pcons}\!:\langle \texttt{p}_1, \ldots, \texttt{p}_\texttt{n}\rangle$. The primitive `pcons` is defined so that the $\texttt{pcons}\!:\langle \texttt{p}_1, \ldots, \texttt{p}_\texttt{n}\rangle\!:\langle \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}\rangle$ is true if $\texttt{p}_\texttt{i}\!:\texttt{x}_\texttt{i}$ is true for all `i`.

Finally, note that FL has a general infix notation. Any function name can be used infix without special syntax.

3.3.2 The Function ρ

The meaning of FL environments is given by the function ρ in Figure 3. The function ρ also serves to give meaning to the function definitions made by patterns. Assignments are denoted $V, W, \ldots$. The following auxiliary functions on assignments are used in the semantics:

$$
\begin{aligned}
\mathrm{dom}(V) &= \{\mathbf{f} \mid V(\mathbf{f}) \neq \#\} \\
(V_1 \oplus V_2)(\mathbf{f}) &= \begin{cases} V_1(\mathbf{f}) & \text{if } V_1(\mathbf{f}) \neq \# \\ V_2(\mathbf{f}) & \text{otherwise} \end{cases} \\
V_1 \mid V_2 &= \begin{cases} V_1 \oplus V_2 & \text{if } \mathrm{dom}(V_1) \cap \mathrm{dom}(V_2) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases} \\
(V_1 \downarrow X)(\mathbf{f}) &= \begin{cases} \# & \text{if } \mathbf{f} \notin X \\ V_1(\mathbf{f}) & \text{otherwise} \end{cases} \\
(\mathbf{f} \leftarrow x)(\mathbf{g}) &= \begin{cases} x & \text{if } \mathbf{f} = \mathbf{g} \\ \# & \text{otherwise} \end{cases}
\end{aligned}
$$

The assignment $V_1 \oplus V_2$ is the union of V_1 and V_2 with clashes resolved in favor of V_1. The assignment $V_1 | V_2$ is a symmetric union of two assignments with disjoint domains. The assignment $V_1 \downarrow X$ is V_1 with domain restricted to X. Finally, $\mathbf{f} \leftarrow x$ is an assignment defined at the single point $\mathbf{f}$.

The rest of this section elaborates on the definition of ρ in Figure 3. The first several clauses explain the meaning of the various environment operators. The difference between `e1` **where** `e2` and `e1` **uses** `e2` is that the former exports only the definitions of `e1` whereas the latter exports definitions from both `e1` and `e2`, with clashes resolved in favor of exporting the definition in `e1`. `PF` is the environment of primitive functions; `PF` is discussed further in Section 3.4. The environment `lib("libname")` is the FL mechanism for importing functions from a pre-compiled library of function definitions.

The semantics of function definitions is straightforward except for the use of patterns and the function F on the right-hand side of definitions. Patterns are translated into a conditional where an appropriate exception is generated in the case where the predicate fails. The function F is applied to guarantee that the right-hand side of a function definition is in fact a function. For example, the definition `def foo ≡ 1` is a valid FL function definition. The meaning of this function is $F(1) = \lambda(y,h).1 : (y,h)$. Thus, `foo` is a function that returns $Exc(\langle$`"apply","arg1",`$\langle$`1,x`$\rangle\rangle)$ when applied to `x`. The function F is the identity on functions, so if the right-hand side of a function definition is already a function then F does not change it.

This mechanism for converting non-functions into functions may seem unnecessarily elaborate. An obvious alternative is to try to forbid function definitions where the right-hand side is not a function and thus avoid the use of F. This approach does not work for two reasons. First, because FL is a strict language, a lazy operation like F is needed somewhere to give recursive equations non-trivial least solutions. Second, there is a difficulty with I/O. To test whether the right-hand side of a definition is a function or not, in general it is necessary to evaluate it. This cannot be done at compile-time if the

$$
\begin{array}{rcl}
\rho(\mathbf{e_1}\ \underline{\mathbf{where}}\ \mathbf{e_2}, V) & = & \rho(\mathbf{e_1}, \rho(\mathbf{e_2}, V) \oplus V) \\
\rho(\mathbf{e_1}\ \underline{\mathbf{uses}}\ \mathbf{e_2}, V) & = & \rho(\mathbf{e_1}, \rho(\mathbf{e_2}, V) \oplus V) \oplus \rho(\mathbf{e_2}, V) \\
\rho(\mathbf{e_1}\ \underline{\mathbf{union}}\ \mathbf{e_2}, V) & = & \rho(\mathbf{e_1}, V) \mid \rho(\mathbf{e_2}, V) \\
\rho(\underline{\mathbf{export}}(\mathbf{f_1}, \ldots, \mathbf{f_n})\,\mathbf{e}, V) & = & \rho(\mathbf{e}, V) \downarrow \{\mathbf{f_1}, \ldots, \mathbf{f_n}\} \\
\rho(\underline{\mathbf{hide}}(\mathbf{f_1}, \ldots, \mathbf{f_n})\,\mathbf{e}, V) & = & \rho(\mathbf{e}, V) \downarrow (\mathcal{N} - \{\mathbf{f_1}, \ldots, \mathbf{f_n}\}) \\
\rho(\mathbf{PF}) & = & PF \\
\rho(\{\mathbf{defn_1}, \ldots, \mathbf{defn_n}\}, V) & = & \text{least } V' \text{ s.t. } \left\{ \begin{array}{rcl} V' & = & (\rho(\mathbf{defn_1}, V_1) \mid \ldots \mid \rho(\mathbf{defn_n}, V_n)) \oplus V \\ V_i & = & \left\{ \begin{array}{ll} V & \text{if } \mathbf{defn_i} = \mathbf{nrdef} \ldots \\ V' & \text{otherwise} \end{array} \right. \end{array} \right. \\
\rho(\mathbf{def\ f} \equiv \mathbf{e}, V) & = & \mathbf{f} \leftarrow F(\mathbf{e}, V) \\
\rho(\mathbf{nrdef\ f} \equiv \mathbf{e}, V) & = & \mathbf{f} \leftarrow F(\mathbf{e}, V) \\
\rho(\mathbf{def\ f\ p} \equiv \mathbf{e}, V) & = & \mathbf{f} \leftarrow F(\mathbf{p} \rightarrow \mathbf{e}; \overline{\mathbf{signal} \circ [\tilde{}\mathbf{"f"}, \tilde{}\mathbf{"arg1"}, \mathbf{id}]}, V) \\
\rho(\mathbf{nrdef\ f\ p} \equiv \mathbf{e}, V) & = & \mathbf{f} \leftarrow F(\mathbf{p} \rightarrow \mathbf{e}; \overline{\mathbf{signal} \circ [\tilde{}\mathbf{"f"}, \tilde{}\mathbf{"arg1"}, \mathbf{id}]}, V) \\
\rho(\mathbf{type\,x} \equiv \mathbf{e}, V) = W \text{ s.t. } \left\{ \begin{array}{rcl} W(\mathbf{isx}) & = & cond : \langle hastag : i_x, tt, ff \rangle \\ W(\mathbf{mkx}) & = & cond : \langle \gamma : F(\mathbf{e}, V), tag : i_x, \lambda(y,h).(Exc(\langle \text{"}mkx\text{"}, \text{"}arg1\text{"}, y \rangle), h) \rangle \\ W(\mathbf{unx}) & = & cond : \langle W(\mathbf{isx}), untag : i_x, \lambda(y,h).(Exc(\langle \text{"}unx\text{"}, \text{"}arg1\text{"}, y \rangle), h) \rangle \\ W(\mathbf{f}) & = & \left\{ \begin{array}{ll} (\rho(\mathbf{e}, V)(\mathbf{f})) \circ W(\mathbf{unx}) & \text{if } \mathbf{f} \in \mathrm{dom}(\rho(\mathbf{e}, V)) \\ \# & \text{otherwise} \end{array} \right. \end{array} \right. \\
\rho(\mathbf{f.x}, V) & = & \mathbf{f} \leftarrow id \\
\rho([\![\mathbf{f_1}, \ldots, \mathbf{f_n}]\!], V) & = & (\rho_1(\mathbf{f_1}, V) \mid \ldots \mid \rho_n(\mathbf{f_n}, V)) \oplus V \\
 & & \text{where } (\rho_i(\mathbf{f_i}, V))(\mathbf{f}) = \left\{ \begin{array}{ll} (\rho(\mathbf{f_i}, V)(\mathbf{f})) \circ si & \text{if } \mathbf{f} \in \mathrm{dom}(\rho(\mathbf{f_i}, V)) \\ \# & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure 3: The function ρ.

evaluation involves I/O. For example, consider the definition `def foo` $\equiv$ `s1:`$\langle$`1, adder:0`$\rangle$, where `adder` is the function defined in example (2.5). The use of F in the semantics makes clear that the I/O happens each time that `foo` is applied.

Type declarations simply add new functions to the environment. The type declaration `type x` $\equiv$ `e` defines the functions `mkx`, `unx`, and `isx` as well as any selectors defined by `e` if `e` is a pattern. The tag $\mathtt{i_x}$ is selected by the implementation and is guaranteed to be unique even if there is another declaration `type x` $\equiv$ `e`$'$ in another scope. The function γ is used to prevent I/O from taking place in the type predicate (see the definition of γ in Section 3.4). The motivation for this design is that types are intended to be fixed sets of values; that is, the set of values that are elements of a type should not depend on external state. For this reason, the function γ is used to ensure that the definition of a type cannot depend on the history component of the semantics.

Finally, the functions defined by a pattern are given by the last three clauses. An elementary pattern `f.` or `f.x` defines `f` to be the identity. If `f` is embedded in a pattern construction $[\![\ldots]\!]$, then the definition of `f` is composed with the appropriate selector function for `f`$'$s position in the pattern. Note that if a pattern is given for a type definition that any selectors defined by the pattern are composed with the destructor for that type.

3.4 Primitive Functions

The primitive functions of FL are exported by the environment PF. There are a lot of primitives—about 75 in all. Definitions of the more important primitives used in this paper are given in Figure 4. In this figure, it is assumed that x, y, x_i and y_i are elements of $\mathcal{D}^+_{\mathrm{FL}}$, that f, g, and f_i are functions, and that i is an integer. These definitions omit some cases. The missing cases are defined uniformly for all primitive functions as follows. In an application `f:x`, if `f` or `x` is an exception or $\perp$, then the meaning of the application is an exception or $\perp$ by strictness. Otherwise, if `f:x` does not match the pattern in Figure 4 for primitive `f`, then the meaning is $Exc(\langle$`"f", "arg1", x`$\rangle)$. For a second-order function `f:x:y` the rules are the same, except that exceptions generated by the primitive have the form $Exc(\langle$`"f", "arg2",` $\langle$`x, y`$\rangle\rangle)$.

The functions `cond`, `catch`, and `gamma` merit further discussion. It is well-known that a strict conditional operator is problematic, since strictness requires that `if p then q else r` evaluate all of `p, q`, and `r` . The desired behavior is, of course, that `q` should be evaluated only if `p` is true and `r` should be evaluated only if `p` is false.

The FL function `cond` is strict (as are all FL functions) but the desired semantics is still achieved because `cond` is a higher-order function. That is `cond:`$\langle$`p, q, r`$\rangle$ evaluates all of `p, q` and `r`, but in this case `p, q`, and `r` are functions that are used to define the conditional. When an additional argument is supplied, `cond:`$\langle$`p, q, r`$\rangle$`:x`, evaluates to `q:x` if `p:x` is true and `r:x` if `p:x` is false. (Note that truth is denoted by any non-false, non-error value in FL, a convention adopted from Lisp.) Thus, a strict, functional version of conditional can be had by making conditional a second-order function. The fact that `cond` is a second order function in FL has a large impact on programming style. The function `cond` is the only

way to write a conditional in FL. Because conditionals are used heavily in programming, this essentially forces FL programmers to write at the function level.

The function `catch` provides the only mechanism for catching and handling exceptions in FL. All FL functions are strict with respect to exceptions, so that $\texttt{catch}{:}Exc(\texttt{x}) = Exc(\texttt{x})$. Thus, it is not possible to define `catch` so that it simply can be composed with another function; i.e., $\texttt{catch} \circ \texttt{f}$ cannot handle exceptions generated by `f` because `catch` is strict. The trick to defining a strict function that can handle exceptions is to make it higher-order. Thus, `catch` is a function that takes two functions `f` and `g` and produces a function that returns $\texttt{g}{:}\langle \texttt{x}, \texttt{y} \rangle$ if $\texttt{f}{:}\texttt{x} = Exc(\texttt{y})$ (recall Example (11) in Section 2.4).

Both `cond` and `catch` are instances of a general technique for achieving desired functionality in a strict language; the lesson is that functionality often can be hidden in the internals of a higher-order function definition. A final example is γ, which suppresses I/O. Intuitively, γ converts any function `f` into a function that cannot perform I/O. If `f` does no I/O, then $\gamma{:}\texttt{f}{:}\texttt{x} = \texttt{f}{:}\texttt{x}$. If `f` does attempt to perform I/O, then the result is a `"gamma arg2"` exception and the history is unmodified. Note that the definition of γ in Figure 4 is purely functional. The function γ is used primarily in the language semantics to guarantee that no I/O takes place as part of a user-defined type constructor. It is also used by programmers to make explicit portions of programs that do not perform I/O.

4 Commentary and Comparison With Other Languages

This section provides a wide-ranging comparison of FL with other languages as well as opinions on what is good and bad about FL. The discussion is organized around four topics: syntax, types, I/O, and exceptions; included in the discussion of exceptions are remarks on the order of evaluation in FL. These topics are the main points of contrast between FL and other functional languages. Several specific items are discussed under each general topic. The choice of items is *ad hoc*, but every item illustrates either something especially unusual about FL, a particularly strong or weak point of the language design, or an interesting relationship to other languages.

4.1 Syntax

FL programs are written at the function level—programs are formed from other programs using higher-order combinators. For example, recall the program SumAndProd from Section 2 in which one begins with an object $\langle \texttt{x+y}, \texttt{x*y} \rangle$ and abstracts over the variables of the object to obtain the program $\lambda(\texttt{x}, \texttt{y}).\langle \texttt{x+y}, \texttt{x} * \texttt{y} \rangle$.

One advantage of the lambda-based approach is that the function "looks like" the answer; by inspecting the form of the body of the lambda expression one can get an idea of the answer— i.e., the expression $\lambda(\texttt{x}, \texttt{y}).\langle \texttt{x+y}, \texttt{x*y} \rangle$ clearly returns a pair, the first component of which is a sum and the second component of which is a product. On the other hand there are two apparent disadvantages to the object-level style in this case: the size of the resulting program, and the need to name and reference the bound variables.

$$
\begin{array}{rcl}
K : x : (y,h) & = & (x,h) \\
apply : (\langle f,x\rangle,h) & = & f : (x,h) \\
comp : \langle f_1,\ldots,f_n\rangle : (x,h) & = & f_1 : (\ldots(f_n : (x,h))\ldots) \\
cons : \langle f_1,\ldots,f_n\rangle : (x,h_0) & = & (\langle y_1,\ldots,y_n\rangle,h_n) \quad \text{where } (y_i,h_i) = f_i : (x,h_{i-1}) \\
cond : \langle f_1,f_2,f_3\rangle : (x,h) & = & \left\{ \begin{array}{ll} (y_1,h_1) & \text{if } \ y_1 \in \mathcal{E}_{\mathrm{FL}} \\ f_3 : (x,h_1) & \text{if } \ y_1 = \mathit{false} \\ f_2 : (x,h_1) & \text{otherwise} \\ \multicolumn{2}{r}{\text{where } \ (y_1,h_1) = f_1 : (x,h)} \end{array} \right. \\
pcons : \langle f_1,\ldots,f_n\rangle : (x,h_0) & = & \left\{ \begin{array}{ll} (\mathit{false},h_0) & \text{if } x \neq \langle x_1,\ldots,x_n\rangle \\ (y_i,h_i) & \text{if } \{y_1,\ldots,y_{i-1}\} \cap (\mathcal{E}_{\mathrm{FL}} \cup \{\mathit{false}\}) = \emptyset \text{ and } y_i \in \mathcal{E}_{\mathrm{FL}} \\ (\mathit{false},h_i) & \text{if } \exists 1 \leq i \leq n . y_i = \mathit{false} \\ (\mathit{true},h_n) & \text{otherwise} \\ \multicolumn{2}{r}{\text{where } \ (y_i,h_i) = f_i : (x_i,h_{i-1})} \end{array} \right. \\
id : (x,h) & = & (x,h) \\
tt & = & K : \mathit{true} \\
ff & = & K : \mathit{false} \\
\gamma : f : (x,h) & = & \left\{ \begin{array}{ll} (y,h) & \text{if } \ f : (x,h) = (y,h) \\ (Exc(\langle \text{"}gamma\text{"}, \text{"}arg2\text{"}, x\rangle),h) & \text{if } \ f : (x,h) = (y,h'),\ h \neq h' \end{array} \right. \\
\alpha : f : (\langle x_1,\ldots,x_n\rangle,h_0) & = & (\langle y_1,\ldots,y_n\rangle,h_n) \quad \text{where } (y_i,h_i) = f : (x_i,h_{i-1}) \\
catch : \langle f,g\rangle : (x,h) & = & \left\{ \begin{array}{ll} (y,h_1) & \text{if } \ f : (x,h) = (y,h_1) \wedge y \in \mathcal{D}^{+}_{\mathrm{FL}} \cup \{\bot\} \\ g : (\langle x,y\rangle,h_1) & \text{if } f : (x,h) = (Exc(y),h_1) \end{array} \right. \\
signal : (x,h) & = & (Exc(x),h) \\
intsto : (i,h) & = & (\langle 1,\ldots,i\rangle,h) \quad \text{if } \ i \geq 0 \\
hastag : i : (x,h) & = & \left\{ \begin{array}{ll} (\mathit{true},h) & \text{if } \ x = \prec i, y \succ \\ (\mathit{false},h) & \text{otherwise} \end{array} \right. \\
tag : i : (x,h) & = & (\prec i, x \succ, h) \\
untag : i : (\prec i, x \succ, h) & = & (x,h) \\
si : (\langle x_1,\ldots,x_n\rangle,h) & = & (x_i,h) \quad \text{if } \ n \geq i
\end{array}
$$

Figure 4: Some elements of PF.

The FL version of this program $[+, *]$ accomplishes the same thing at the function level by providing syntax for the function `cons` that parallels the syntax for sequences $\langle \ldots \rangle$, resulting in a shorter expression without "junk references." For a more complete discussion of the advantages of programming with a carefully selected collection of combining forms with rich algebraic properties, see [Bac78, Wil82b].

These advantages notwithstanding, FL syntax is peculiar, particularly to programmers who are familiar with other functional languages. Some of the peculiarity is superficial; people who use FL for the first time often are surprised at how easy it is to write FL programs. Still, FL's syntax appears to have more problems than merits. A few of each are discussed in this section.

4.1.1 Prime Notation

While FL's combining forms facilitate the writing of first-order functions, writing higher-order functions is less convenient. FL's prime notation is an attempt to make higher-order combinator expressions "look like" the values they compute.

Consider a function $[\mathtt{f}, \mathtt{g}]$. The lambda calculus version of this function is $\lambda \mathtt{x}.\langle \mathtt{f\ x}, \mathtt{g\ x} \rangle$ (function application of `f` to `x` in lambda calculus is written as the juxtaposition `f x`.) To abstract over `f` in the lambda expression one writes $\lambda \mathtt{f}.\lambda \mathtt{x}.\langle \mathtt{f\ x}, \mathtt{g\ x} \rangle$. To do this in FL, it is necessary to write a combinator that, when applied to `f` , evaluates to $[\mathtt{f}, \mathtt{g}]$. One way to do this is to write:

$$\mathtt{cons} \circ [\mathtt{id}, \tilde{}\mathtt{g}]$$

To see that this works, apply it to `f`:

$$\begin{array}{rl} & (\mathtt{cons} \circ [\mathtt{id}, \tilde{}\mathtt{g}]):\mathtt{f} \\ \rightsquigarrow & \mathtt{cons}:\langle \mathtt{id}:\mathtt{f}, \tilde{}\mathtt{g}:\mathtt{f} \rangle \\ \rightsquigarrow & \mathtt{cons}:\langle \mathtt{f}, \mathtt{g} \rangle \\ = & [\mathtt{f}, \mathtt{g}] \end{array}$$

This program isn't awful, but it isn't easy to read. A better version uses lifted construction:

$$['\mathtt{id}, \tilde{}\mathtt{g}]$$

Now it is clear what the result of the function will be: a lifted construction returns a construction, the argument is substituted for `id` , and `~g` is replace by `g`. This definition is equivalent to the previous one since

$$['\mathtt{id}, \tilde{}\mathtt{g}] = \mathtt{lift}:\mathtt{cons}:\langle \mathtt{id}, \tilde{}\mathtt{g} \rangle = \mathtt{cons} \circ [\mathtt{id}, \tilde{}\mathtt{g}]$$

The next step is to abstract the program again with respect to `g` (i.e., $\lambda \mathtt{g}.\lambda \mathtt{f}.\lambda \mathtt{x}.\langle \mathtt{f\ x}, \mathtt{g\ x} \rangle$). One solution is:

$$\circ \circ [\tilde{}\mathtt{cons}, \mathtt{cons} \circ [\tilde{}\mathtt{id}, \mathtt{id}]]$$

This program is awful—it is not at all clear at a glance what this program does. With prime notation, one can write

$$[''\tilde{}\mathtt{id}, \mathtt{K}]$$

The double prime says that the result is a construction after two arguments are supplied; `~id` throws away the first argument and keeps the second, `K` throws away the second argument and keeps the first. It is straightforward to show that this program is equivalent to the unreadable one above.

To new FL programmers this kind of programming is usually mysterious, but experienced users become surprisingly adept at reading and writing functions using prime notation. In short, the prime notation provides a concise and fairly readable notation for writing higher-order combinators. It is not as readable as the lambda calculus, but it is vastly superior to an unsugared combinator language. An open question: are FL's conveniences at the function level worth the inconveniences of priming at higher levels?

4.1.2 Infix Notation and Precedence

Most contemporary functional languages support infix notation. The favored strategy (adopted in ML and Haskell) is to provide a special form for declaring certain names to be "infixable" along with a precedence level, which is usually an integer in the range 0-9. Some primitive functions are automatically infix operators with pre-specified precedences.

In contrast, FL has a general infix notation and no precedence declarations. Any FL expression `x1 x2 x3` means `x2:`$\langle$`x1,x3`$\rangle$. This is quite convenient, since a programmer may use an operator infix, prefix, or any combination of the two, even in the same expression. Other advantages are that the rules are simple and easy to remember and no new kind of declaration is introduced into the language.

A problem with general infix comes up in printing expressions, because the printer must make an arbitrary decision whether to display a function of two arguments prefix or infix. A reasonable solution is to print all primitive operations that are normally infix (e.g., $\circ, +, *$, etc.) infix and all user-defined functions prefix.

Unfortunately, general infix notation does not come for free. General infix uses up a language's most valuable lexical unit, the blank. In most functional languages, a blank denotes function application, e.g., `f x y` is `f` applied to `x` and `y`. In FL, `f x y` is infix application of `x`. Thus, the cost of general infix is that another notation is required for application; this is why ":" is used to denote function application in FL. General infix notation works reasonably well for first-order expressions, where the most common infix application is function composition. In higher order FL expressions, however, plain function application is used at least as often and usually much more often than infix. The experience with FL programming is that it is often difficult to write FL programs that are not cluttered with ":" symbols. In retrospect, although general infix is a nice idea, it doesn't pay its weight in practice.

A general problem with infix notation that FL shares with other functional languages is precedence. FL has 15 levels of precedence for the unary and infix portions of the grammar. This has proven far too much to expect FL programmers to remember, let alone use. The following example shows a typical FL programming error involving precedence. Assume one wants to write `f` $\circ$ `g`, where `g` = `x` $\rightarrow$ `y;z` is a conditional. Then `f` $\circ$ `x` $\rightarrow$ `y;z` does not parse as desired, since $\rightarrow$ binds more loosely than $\circ$. (A correct

version is f ∘ (x → y; z).) The same problem exists for z and compositions trailing the conditional. Since FL has no static type checking, the first hint of a problem is normally a run-time exception (in the absence of program analysis—see Section 4.2.1). Usually the exception helps the programmer to find the error almost immediately, but occasionally substantial time is lost searching for precedence mistakes.

Although FL has more levels of precedence than other functional languages, FL is probably no worse in practice. This is because FL programmers have difficulty remembering and applying more than a small handful of the precedence rules anyway, so 10 levels would not be better than 15. The solution most programmers adopt is to keep function definitions small by making many subsidiary definitions. This style helps to make programs readable (if informative function names are chosen) and reduces the opportunities for a precedence mistake.

4.1.3 Special Notation

FL has a lot of special notation adopted from FP. FP is a first-order language, with user-definable first-order functions and a fixed set of second-order functionals (the combining forms). Because of the special role of the combining forms, it is reasonable that FP has a special notation for each. This design doesn't work so well in FL. FL is a higher-order language in which programmers can define new second-order (and higher order) functionals. With first-class higher-order functions, giving special notation to a few functions seems arbitrary. It also complicates parsing unnecessarily.

Another problem with FL's special notation is that some of it is too economical—i.e., a one character change can result in a valid, but completely different, program from the one intended. The hardest FL programming error to find is an example of this phenomenon. Recall that an elementary pattern has the form f.p, where f is a function name and p is a predicate. This pattern defines a predicate that matches values for which p is true. If p is just tt (that is, the pattern should match any value) then FL provides a convenient shorthand f. that means the same thing. For example [[f., g.]] is a pattern matching all pairs; f is a selector s1 for the first component and g is a selector s2 for the second component. As discussed in Section 2.6, allowing embedded predicates in patterns is a powerful feature and is very useful in eliminating nested conditionals.

However, consider what happens if a "." is omitted. The pattern [[f, g.]] is still syntactically well-formed. But now f is just a free function name and resolves to whatever definition is visible in the current scope. If one is lucky, f is not defined and a syntax error results. Unfortunately, because programmers tend to reuse the same names for selectors in patterns, the chances are quite good that f is defined in an outer scope; perhaps it is the selector s2. It is very easy to overlook the fact that a "." has been omitted, even when the programmer knows there is a mistake in a pattern. A better language design would be to eliminate the shorthand f. and force programmers to write f.tt, thereby making the difference between the right and wrong versions much more obvious.

4.2 Types

FL is dynamically typed—in principle, all type checking is done at run-time. In this respect, FL is more closely related to Lisp and Scheme than it is to ML or Haskell.

Many people consider static polymorphic type checking to be one of the central features of functional languages. The advantages of static type checking are great: programming is both more secure (since bugs that arise from type errors cannot be introduced into a program) and programs run faster (since no dynamic type checking is required). On the other hand, there are costs. First, because no static type system can be both sound and complete (i.e., no type system can recognize exactly the set of programs that make run-time errors) some programs must be rejected that are otherwise meaningful. Second, the language description becomes more complex, and programmers must understand polymorphic type checking to write working programs. The former point is the source of an endless debate between adherents of dynamic and static typing about whether any of the programs rejected by static type systems are really useful. The latter point is indisputable, but experienced programmers have little if any trouble dealing with static typing. Within the functional programming community, an overwhelming majority supports static typing.

FL has taken a different direction, exploring to what extent an implementation of a simple dynamically typed language can achieve the security and efficiency of a statically typed language. In FL, type errors result in run-time exceptions. For example, the application `s1:0` would be a static type error in most languages (since `s1` is meant to be applied to lists) but results in an `s1` exception in FL. This design imposes a substantial burden on the implementation, because applications of primitives of `s1` must always test to ensure that arguments are of the proper type. One of the goals of the FL implementation has been to investigate to what degree the effects of static type checking can be recovered through program optimization: replacing primitives such as `s1` by versions that perform no type-checking in contexts where program semantics are not changed. For example, in the program $\texttt{s1} \circ [\texttt{\~{}1}, \texttt{\~{}2}]$ the function `s1` is guaranteed to be applied to a non-empty sequence. Therefore, in this program `s1` can be replaced by a different function `Safe_s1` that performs no run-time type check.[1] To determine where run-time type checking can be omitted the FL implementation uses a type inference algorithm.

4.2.1 FL Type Inference

Although the FL language is dynamically typed, the FL implementation has a powerful type inference system. The type system is not part of the language definition. In this respect it functions as a traditional program analysis, which may be used at the discretion of the programmer or compiler, but is not required. This section very briefly describes the FL type system and presents two short examples. The interested reader is referred to [AW93, AWL94] for details.

Functional languages such as ML and Haskell have type systems based on the well-known Hindley/Milner type system [DM82]. Hindley/Milner types include type constructors such as `Int`, `Bool`, and

[1] In fact, $\texttt{s1} \circ [\texttt{\~{}1}, \texttt{\~{}2}]$ can be optimized to `~1`, as described in the following section.

`List(t)` where `t` is a type, function types $\mathtt{t}_1 \rightarrow \mathtt{t}_2$, type variables $\alpha, \beta, \ldots$, and quantified types $\forall\alpha.\mathtt{t}_1$. Types denote sets of values. Type constructors denote sets of primitive and user-defined values such as integers, booleans, and lists. A function type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ denotes the set of functions mapping elements of $\mathtt{t}_1$ to elements of $\mathtt{t}_2$. A variable stands for an unknown type, and quantified types denote sets of polymorphic values. For example, the function `s1` that selects the first element of a sequence has the Hindley/Milner type

$$s1 \ :: \ \forall\alpha.List(\alpha) \rightarrow \alpha$$

where $List(\alpha)$ stands for all FL sequences with elements drawn from the type α and the notation "::" is read "has type".

The FL type system is an extension of the Hindley/Milner system. In addition to the Hindley/Milner types, FL has restricted intersection $t_1 \cap t_2$, union $t_1 \cup t_2$, and complement $\neg t_1$ types, as well as recursive types $\alpha = E(\alpha)$. There is also a least type 0 and a universal type 1. The FL type system (abbreviated FLT) has several nice properties.

- If a program has a Hindley/Milner type, then the Hindley/Milner type is derivable in FLT.
- Every program is typable in FLT.
- Every program has a *minimal* FLT type.
- The minimal FLT type is computable.

FLT's minimal types are the analog of *principal* types in the Hindley/Milner system. The minimal type of a program e is the smallest type derivable for e within the type system. The notion of a minimal type is somewhat different from the notion of a principal type. Principal types are defined syntactically, whereas minimal types are defined semantically. Principal and minimal types coincide in the Hindley/Milner system; FLT has minimal types but not principal types [AWL94].

For brevity, a formal development of the FL type system is not presented here. Two examples should suffice to illustrate how the FL type system differs from the Hindley/Milner system. The first example is the program $\mathtt{s1} \circ [\tilde{}\mathtt{1}, \tilde{}\mathtt{2}]$. This program is a constant function that returns 1. The general Hindley/Milner type for a primitive such as `s1` is

$$\mathtt{s1} \ :: \ \forall\alpha.\mathtt{List}(\alpha) \rightarrow \alpha$$

The type $\mathtt{List}(\alpha)$ is defined by the recursive equation

$$\mathtt{List}(\alpha) = \mathtt{Cons}(\alpha, \mathtt{List}(\alpha)) \cup \mathtt{Nil}.$$

Using this type for `s1`, the best type that can be assigned to the subexpression `s1` in $\mathtt{s1} \circ [\tilde{}\mathtt{1}, \tilde{}\mathtt{2}]$ by the Hindley/Milner system is

$$\mathtt{s1} \ :: \ \mathtt{List(Int)} \rightarrow \mathtt{Int}$$

This typing proves that `s1` always is applied to a list, so `s1` does not need to check its argument at run-time to ensure that it is a list. In addition, since type inference succeeds on this program, the programmer can be confident that `s1` is not applied to, say, a floating point number.

For this program, the FL type system can improve on the efficiency and security of the Hindley/Milner algorithm. Because `Nil` is a possible value of the `List` type, under the Hindley/Milner typing `s1` must still perform a run-time check to ensure that its argument is not `Nil` . To improve the accuracy of type inference, FL type inference does not use the `List` type but rather includes `Cons` and `Nil` as distinct type constructors. In addition, there is a type constructor `Exc(t)` to make exceptions explicit in the type. The general type of the function `s1` in the FL type system is

$$\texttt{s1} :: \forall\alpha,\beta.\texttt{Cons}(\alpha,\texttt{Seq}) \cup (\beta \cap \neg\texttt{Cons}(1,\texttt{Seq})) \rightarrow \alpha \cup \texttt{Exc}(\texttt{Triple}(\texttt{String},\texttt{String},\beta \cap (\neg\texttt{Cons}(1,\texttt{Seq}))))$$

where $\texttt{Seq} = \texttt{Cons}(1,\texttt{Seq}) \cup \texttt{Nil}$ is the set of all sequences, $\neg\texttt{Cons}(1,\texttt{Seq})$ is the set of all values that are not sequences of length at least one, $\texttt{Triple}(\texttt{X},\texttt{Y},\texttt{Z}) = \texttt{Cons}(\texttt{X},\texttt{Cons}(\texttt{Y},\texttt{Cons}(\texttt{Z},\texttt{Nil})))$, and $\texttt{String} = \texttt{Cons}(\texttt{Char},\texttt{String}) \cup \texttt{Nil}$. This example is typical of the types of FL primitive functions. The domain consists of two parts, the "good" arguments (in this case $\texttt{Cons}(\alpha,\texttt{Seq})$) and the "bad" arguments (in this case $\beta \cap \neg\texttt{Cons}(1,\texttt{Seq})$). In the range, there are the normal results (in this case α) and the error results. The type $\texttt{Exc}(\texttt{Triple}(\texttt{String},\texttt{String},\beta \cap (\neg\texttt{Cons}(1,\texttt{Seq}))))$ is the type of an `s1` exception $\langle$`"s1"`, `"arg1"`, `x`$\rangle$ where `x` is the argument to `s1`.

Note that the type signature is polymorphic in both the "good" and the "bad" portions of the domain. For the good arguments, $\texttt{Cons}(\alpha,\texttt{Seq})$ is polymorphic in the type of the head of the list. For the bad arguments, $\beta \cap (\neg\texttt{Cons}(1,\texttt{Seq}))$ can be refined to any subset of the exception producing domain by instantiating the type variable β. Polymorphism for the exception producing domain is useful for two reasons. First, exceptions are intended to be used liberally in FL programming and sacrificing the accuracy of type inference for exceptions would make exceptions more expensive than necessary (since fewer optimizations could be applied) and therefore discourage their use. Second, accurate type information for exceptions is useful for debugging, as it can tell the programmer exactly what types of values may produce an exception.

The type assigned by the FL type system to the instance of `s1` in this program is

$$\texttt{s1} :: \texttt{Cons}(\texttt{Int},\texttt{Seq}) \rightarrow \texttt{Int}$$

It is easy to check that this is an instance of the quantified type for `s1` by substituting `Int` for α and `0` for β. (The minimal type 0 has the property that $0 \cap T = 0$ and $0 \cup T = T$ for all types T.) This typing proves that `s1` is always applied to a non-empty sequence, so no run-time check is required for `Nil`, and it is immediate that `s1` can never generate an exception.

The second example uses a function that computes the last element of a sequence:

$\texttt{last:}\langle 1, 1, \texttt{`a}\rangle$ `where`

$\{$ `def last isseq` $\wedge$ $(\texttt{not} \circ \texttt{isnull}) \equiv \texttt{isnull} \circ \texttt{tl} \rightarrow$ `s1;last` $\circ$ `tl` $\}$

The function `last` is defined using the predicate `isseq` $\wedge$ (`not` $\circ$ `isnull`) on the left-hand side of the $\equiv$ sign. Recall (Section 2.6) that this predicate is applied to `last`'s argument, and if it fails, an exception is returned. In this case, an exception is returned if `last` is applied to a non-sequence or $\langle\rangle$.

The meaning of this expression is '`a`. The type assigned by the FL type system is `Char`. The most interesting aspect of the analysis is the type assigned to the use of `last` in the top-level expression:

$$\texttt{last} :: \texttt{X} \rightarrow \texttt{Char where X} = \texttt{Cons}(\texttt{Int}, \texttt{X}) \cup \texttt{Cons}(\texttt{Char}, \texttt{Nil})$$

This type shows that `last` takes a non-empty sequence where the last element is a `Char` and returns a `Char`. The type also says that sequence elements other than the last are integers. The FL type shows that no run-time exceptions are possible, so this program can run without run-time type checking. This program is not typable in the Hindley/Milner system, because the argument to `last` is a heterogeneous sequence.

4.2.2 Discussion

FL's lack of a static type system has motivated considerable research on new and powerful type systems that are appropriate for dynamically typed languages; the FL type system is the result of this work. The FL type system is a proper extension of the Hindley/Milner system that is able to type more programs more accurately than the Hindley/Milner system. This is accomplished in a language with no static type constraints, so the programmer and compiler are free to use the type system or not. So what, if anything, is the catch? This section covers four catches: the speed of type inference, understanding types, run-time type tags, and module interfaces.

The FL type system does more than the Hindley/Milner system, so it is not surprising that it is slower. The asymptotic complexity of the FL type inference algorithm is double exponential time, while the Hindley/Milner type inference algorithm runs in (only!) single exponential time. In practice, the FL type system is at least one order of magnitude slower than the Hindley/Milner system. The FL type system is still fast enough to be usable, but it would be nice if it were a lot faster; it currently takes up to 30 seconds to analyze a 300 line program.

FL types are more complex than Hindley/Milner types, so an FL programmer who wishes to make use of type information must deal with types more complex than an ML programmer. FL types are harder to read than Hindley/Milner types, primarily because there is not a unique representation of a given type (i.e., many type expressions are equivalent). The implementation of the FL type system makes use of identities between types to simplify the representation of types. In practice, such simplification greatly improves the readability of type expressions.

The FL system provides type inference as a program analysis tool—it is not a required step of compilation. Thus, an FL run-time system must carry type tags on all data, because compiled code does in general perform run-time type checking. All existing implementations of functional languages carry at least some run-time tags on data because existing garbage collection algorithms require them. There

is, however, an emerging technology of tagless garbage collection, where compile-time types are used at run-time by the garbage collector to reconstruct tags [App89]. If and when tagless garbage collection algorithms become practical, implementations of statically typed functional languages will be able to use them; this option cannot be exploited in an FL implementation.

The last and most serious cost of the FL approach to types is the lack of any module system (in FL parlance, a module is a *library*). This is not an inherent property of the type system; FL type inference could accommodate a mechanism for specifying module interfaces. Rather, the absence of a module system is the result of a design philosophy that avoids language complexity.

FL's module problem can best be explained with another perspective on the relationship between FL type inference and standard Hindley/Milner type inference. Compare again the Hindley/Milner and FL types for `s1`.

$$\texttt{s1} \;::\; \forall\alpha.\texttt{List}(\alpha) \rightarrow \alpha$$

$$\texttt{s1} \;::\; \forall\alpha,\beta.\texttt{Cons}(\alpha,\texttt{Seq}) \cup \beta \cap (\neg\texttt{Cons}(1,\texttt{Seq})) \rightarrow \alpha \cup \texttt{Exc}(\texttt{Triple}(\texttt{String},\texttt{String},\beta \cap (\neg\texttt{Cons}(1,\texttt{Seq}))))$$

Among other things, the Hindley/Milner type says that `s1` must be used in a context where it is applied to a list; if it is not, the compiler rejects the program. The FL type has no such restriction—the domain of `s1` includes all values and so the function can be used in any context. If `s1` is in fact applied to a list, then the FL type of `s1` is simplified for that instance (see the examples in Section 4.2.1). Thus, FL type inference relies very heavily on information about the context in which a function is used in determining the functions that do not require run-time type checking.

Unfortunately, this use of context information does not work well with separate compilation and program optimization. Consider an FL library A that uses functions defined in library B. Library A can be compiled using the types inferred for functions in B, but library B cannot be compiled knowing that functions in B are used in the context given by A, because library B may be used at some future point by a new library C that uses B's functions in a different context. If functions in library B were specialized for library A, then B would have to be recompiled for C; in fact, B would have to be recompiled even if A were modified. In short, libraries could not be compiled separately. The current FL solution is to compile libraries assuming that exported functions can be used in any context. This allows separate compilation at the cost of performing less program optimization.

A method for specifying module interfaces would fix this problem. The programmer could then describe the expected contexts in which library functions would be used; the compiler would both optimize functions using that information and verify that uses of a function were consistent with its specification. To do this, however, would require that module specifications and types be included in the FL language definition.

4.3 Input/Output

Functional programming does not lend itself readily to interaction with agents independent of a program. The canonical problem is I/O and, specifically, the problem of writing interactive functional programs.

For many years there were only two solutions to this problem. The "convenient solution" was not to have a functional language at all. ML, Lisp, and Scheme are in this category. Although all are functional in the sense that an applicative style of programming is encouraged in practice, all have state and primitive operations that modify the state by side-effect. The "pure solution" was to exploit lazy evaluation by treating I/O routines as stream-valued stream functions. The convenient approach has the advantage of allowing an unrestricted style of I/O programming at the cost of a more complicated underlying semantics. On the other hand, the difficulty with the explicitly functional approach is that the increased complexity of a program's functionality becomes a notational nuisance, which masks the primary purpose of the program. For example, a lazy, interactive program to correct misspelled words in a file using an (updatable) dictionary has the primary purpose of mapping an `in_file` to a `corrected_file` but has the proper functionality

$$\langle \texttt{in\_file}, \texttt{dictionary}, \texttt{keyboard} \rangle \rightarrow \langle \texttt{corrected\_file}, \texttt{dictionary}', \texttt{screen}, \texttt{rest\_of\_keyboard} \rangle$$

Note that the program must be given `keyboard` as input to receive responses to interactive queries it puts on `screen` , and it must return the "rest" of the keyboard input for use by subsequent programs.

The treatment of I/O in FL is an attempt to combine the advantages (and avoid the problems) of these two approaches by (semantically) attaching an explicit history component to the functionality of all programs and (syntactically) suppressing consideration of that component. The success of this approach depends on the extent to which programmers can use the convenience of treating the history component implicitly, thus keeping their program notations uncluttered, and still be able to write clear programs.

Overall, FL's I/O mechanism has worked out quite well in the FL implementation. Only a small portion of the typical FL program uses I/O, so any reasonable scheme for I/O would be workable. The FL design is especially convenient in two respects. First, because every function has an implicit history component, it is very easy to add or remove I/O from programs without rewriting large sections of code. If the history component were explicit, then adding I/O to one function would require changing the functionality (i.e., rewriting) every function that depended, directly or indirectly, on that function. Second, because the history component is abstract and does not attach any particular semantics to I/O, it provides a convenient device for making FL programs work in other ways with the outside world in addition to I/O. For example, in the FL compiler primitive functions that may make arbitrary C function calls are modelled as functions that modify the history. Because the FL compiler is guaranteed to preserve program semantics (including history operations) this has proven to be a simple way to integrate existing C library routines into FL programs.

A valid argument against the FL design is that the pervasive, implicit I/O component of the semantics could seriously handicap program optimization, since program transformations must take account of the potential for I/O anywhere in an FL program. In practice, this potential problem is a minor consideration. Because most primitive functions do not depend on the history component of the semantics and I/O is not used pervasively in most programs, very simple program analysis can identify almost all expressions that do not perform I/O; within these sections of code more general program transformations apply.

Furthermore, because a machine's I/O operations are usually very slow relative to the speed of a machine's processor, large-scale optimization on portions of programs that do a great deal of I/O frequently does not improve program performance significantly. This last argument must be qualified by the type of I/O that is being performed. It certainly holds for interactive programs; it is less true of programs with stringent real-time I/O constraints.

Since the development of FL's I/O system another, more general, approach to I/O in functional languages has been discovered. This approach, known as *monadic* I/O, uses higher-order combinators to hide and control how the history component is threaded through a computation [Wad90, PJW93]. Monadic I/O uses a particular *monad*; variations on the same higher-order combinators can implement other language features such as exceptions and continuations as monads. Essentially, FL I/O is itself a particular monad built into the language's denotational semantics. The monadic approach takes I/O out of the semantics and makes it available to the programmer, which gives monadic I/O several advantages over the FL design. First, it is more programmable—the history component is not wired into the language in a particular way and, within limits, the wiring can be rearranged by the programmer. Second, monads can explicitly delimit the scope of computations that use I/O, making program analysis to discover where I/O cannot be performed unnecessary. The only disadvantage of monadic I/O with respect to FL is that programs must be structured using a monad to take advantage of it. Thus, adding I/O to a program that is not already structured using the I/O monad may involve rewriting a large portion of the program. Because every FL program has I/O built in, there is no such cost in FL programming.

4.4 Exceptions and Order of Evaluation

This section discusses the experience with exceptions in FL. Because exceptions are inextricably tied with order of evaluation, that topic is addressed here as well. Exceptions are one of the successes of FL. On the semantic level, it is worth stressing that FL's denotational treatment of exceptions is referentially transparent—FL exceptions are functional. On the pragmatic level, exceptions have proven very useful in programming and reasonably efficient to implement.

Exceptions help FL programmers in at least two ways. First, there is a "throw and catch" style of programming that is easy with exceptions but difficult to simulate without exceptions. The common scenario is that one wishes to exit early from a computation in certain circumstances. For example, in computing the conjunction of a sequence of truth values there may be no reason to continue after the first `false` value is encountered. Signalling an exception ends the computation; all that is needed is a surrounding `catch` that handles the exception.

The second advantage of exceptions in FL is the security that a program cannot terminate without returning a meaningful value. There is nothing implementation-dependent about FL exceptions. Program optimization or porting code to a different implementation cannot change the exception produced. This property is important because exceptions play a very large role in helping FL programmers debug programs (recall that all kinds of errors—including type errors—are exceptions in FL). The standard

system exceptions are often informative enough to isolate program bugs quickly. An extension that has been considered but not implemented in the FL system is to include line number information in exceptions that are the meaning of an entire program. The argument for this extension is that it makes finding where exceptions arise even easier than just having the name of the function; the argument against it is a minor loss of referential transparency, since a program's meaning can now depend on the textual layout of the program.

A useful point of comparison is the situation with run-time exceptions and Lisp compilers. When running Lisp programs, it is not uncommon to receive one error message when the program is compiled and a different error message when the program is interpreted. Frequently, the error message from the compiled program is on a topic unrelated to the actual problem. This sort of behavior arises because optimizing Lisp compilers freely rearrange operations without regard to preserving errors. It also makes debugging Lisp programs much more difficult than necessary.

Another point of comparison is ML, which incorporates exceptions in a way quite similar to FL. Both the motivation for and experience with exceptions in ML are similar to that of FL [App93]. The major difference is that exceptions are used uniformly in FL, while in ML certain errors are treated as static type errors and others as run-time exceptions.

There are semantic and pragmatic arguments for not using exceptions. The pragmatic argument is twofold: first, implementations of exceptions and exception handling are expensive, and second, making exceptions part of program semantics greatly inhibits program optimization. These problems are serious in FL, because exceptions are used pervasively. The solution to these problems in the FL implementation is itself twofold. The FL type system is able to prove at compile-time that most functions cannot produce exceptions. In addition, a robust theory of program transformation in the presence of exception producing functions has been developed and implemented in the FL compiler [AWW90]. Together, these two tools reduce the cost of FL's pervasive use of exceptions to a tolerable level.

The semantic argument against exceptions is that adding exceptions forces too much to be specified about the order of evaluation. For example, using exceptions a programmer can observe the order in which a function evaluates its arguments. Allowing this kind of programming does not interact well with lazy evaluation; this is one of the reasons why lazy functional languages do not have built-in exceptions.

At one time, this semantic argument carried significant weight for FL—the language that evolved into FL was lazy for many years [HWW86]. Originally, FL became strict because it was necessary to specify an order of evaluation to guarantee that the single-threaded history component for I/O is handled correctly. Exceptions were added to the language later [BWW+89]. Today, much of the original motivation for strictness has been removed by the discovery of monadic I/O (see Section 4.3). FL's I/O system could be replaced by monadic I/O; I/O programming would become somewhat more inconvenient and the language semantics would be simplified. Overall, the tradeoff may well be worthwhile, since I/O is a small component of most programs. Exceptions, too, can be expressed using a monad, so one might suppose that exceptions could also be taken out of the semantics, provided as a monad, thus further simplifying FL. While this argument has some validity, it does not account for the pragmatic requirements

of FL programming. A large part of the security of FL programming comes from the fact that exceptions are built-in. Monads require some extra effort to use, so FL programmers would constantly pay a price for the privilege of occasionally programming without exceptions. Because of the lack of static typing as a default, programmers who chose not to have exceptions would be seriously handicapped in debugging FL programs. There is of course another alternative: drop exceptions and add static typing. This alternative is the design taken in Haskell [HWA+88].

After much experience with writing functional programs in FL, another rationale has emerged for having a strict language. It is often necessary to rewrite a functional program to make it run faster; there are always things even an aggressive optimizing compiler cannot do. The task of rewriting programs for performance is aided greatly by a clear understanding of the order of evaluation. Using FL's simple, conventional leftmost-innermost evaluation order, it is easy to reason about the time and space complexity of programs. In lazy languages very little is known about how to do such reasoning in general, and programmers writing in lazy languages today have to use considerable ingenuity to write efficient programs. It should be said that not much attention has been given to the problem of writing efficient lazy functional programs as yet, although it is becoming an active area of research [Lau93].

5 Experience and Lessons Learned

FL is designed to be a simple language to learn and to use. These desiderata are reflected both in the syntax and semantics of FL. FL's syntax is very uniform at all levels of programming; the only syntax a programmer must learn is the function definition. In contrast, most other languages (functional or otherwise) often have a separate type language, module interface language, and perhaps a compiler pragma language included as part of the programming language. Semantically FL manages to be simple while still providing support for practical programming features including I/O, exceptions, and user-defined types.

Among the people who have tried programming in FL, the ones who like FL most are those who have little previous programming experience. Frequently, these people have been frustrated by the level of sophistication required to program in conventional imperative languages such as C. FL appeals to naive programmers because it provides a simple, flexible programming model that can be learned very quickly. For this group of people, it is considered an advantage that FL has no static type system or module interface system that must be learned.

At the other end of the spectrum, users who have substantial programming experience in other functional languages are less enthusiastic about FL. While these people find FL usable for programming, FL lacks the basic tools they are accustomed to, especially static typing and a module system. Ironically, these omissions are part of what makes FL attractive to inexperienced programmers. In fact, both groups have a point. FL's very simplicity makes it easy for someone to start programming without being burdened with the daunting task of learning a large language. On the other hand, FL is too simple for sophisticated programmers who want more precise control of the programming process.

Details of FL's syntax are annoying to inexperienced and experienced users alike. The most common problems arise from having two syntactic forms that differ by only a single character, which makes it too easy to accidentally enter a valid program that means something different from what was intended. The lexical structure of FL is too concise—there should be more redundancy to help prevent this kind of programming error.

On the other hand, exceptions and I/O both work very well in FL. Exceptions, in particular, turned out to be much more important than originally imagined. In a programming language with exceptions but without static typing, exceptions assume the role of the primary debugging assistance provided by the language. This design works reasonably well in practice, because in most cases the exception returned by a primitive function is sufficient to isolate a type error quickly.

FL's I/O mechanism provides an unobtrusive mechanism for interacting with agents external to an FL program. The important lesson to be drawn from FL's I/O mechanism is the value of designing I/O into a programming language from the outset. The FL definition has a very general I/O interface that is well-integrated into the language semantics. In the FL implementation, the I/O interface was eventually used in ways that were not considered during the design; for example, the I/O mechanism provided a convenient, semantically safe way to call foreign functions. The careful, general initial design of FL paid off handsomely in this case; without the general I/O mechanism, it would have been necessary to design something *ad hoc* part way through the language implementation and it is unlikely that it would have worked out as nicely.

In many respects the design and quality of a language implementation is as important as the design of the programming language itself. For the FL compiler developed at IBM Almaden, considerable emphasis has been placed on generating code that is completely faithful to the language semantics. This compiler has been in use internally for small and large projects for several years and at this point there is a high degree of confidence in the correctness of the code produced. The target language of the compiler is C. Many other compiler projects have used C as a target because it is reasonably well-suited to the role of a portable assembly language. However, portability turned out not to be the greatest benefit of using C. The single greatest benefit to using C as a target is that it is very easy to write programs that interoperate with C routines; in addition, one immediately has access to the vast array of C library routines already in existence.

Somewhat surprisingly, compatibility with C is the key that has made it feasible for users to write in FL. Writing a large application from scratch is hard work in any language; it is particularly uninviting when significant portions of the application are already implemented in another language. Making it possible for people to use FL without giving up their favorite C libraries has made giving FL a try a much more attractive and viable prospect.

It was recognized from the outset of the FL compiler effort that program optimization would play a critical role in generating good code from FL programs. This has indeed proven to be the case. The FL compiler performs a great deal of program analysis and transformation, including many standard optimizations as well as some unique to FL. Among the latter, the most important is the combination

of type inference and program transformation aimed at reducing the overhead of run-time type checking and exception handling (see Section 4.2).

The most serious flaw in the FL compiler is a direct result of the heavy emphasis on program analysis and optimization. The FL compiler is slow, with most of the time spent in the backend phases. In this respect FL is in good company with other functional language implementations, some (but certainly not all) of which are slower than the FL compiler. Unfortunately, compile time is a scarce resource and must be treated as such. In the FL compiler, it has been necessary to insert a "fast path" for compilation that bypasses all of the analysis and optimization and enables programs to be compiled in a few seconds. Almost all FL development is done using the non-optimizing version of the compiler; optimization is only used in the final stages of software development. As functional programming becomes more popular and "mainstream", it will be necessary to place more emphasis on writing compilers for functional languages with good interactive performance.

6 Acknowledgements

The authors are grateful to all of the people who have worked on the design and implementation of FL. John Backus supervised and supported all of the design and much of the implementation. Peter Lucas also contributed a great deal to the FL definition and compiler. In roughly chronological order, Tim Winkler, Bill Griswold, Thom Linden, Paul Tucker, Brian Murphy, Brennan Gaunce, David Evans, and TK Lakshman all wrote substantial portions of the FL compiler and runtime system. Finally, the authors are indebted to Ray Strong and Danny Dolev for illustrating by example how to program large systems in FL.

References

[App89] A. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[App93] A. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4), 1993. to appear.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[AWW90] A. Aiken, J. H. Williams, and E. L. Wimmers. Program transformation in the presence of errors. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 210–217, January 1990.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bac85] J. Backus. From function level semantics to program transformation and optimization. Technical Report RJ 4567 (49035), IBM, 1985.

[BWW86] J. Backus, J. H. Williams, and E. L. Wimmers. The FL language manual. Technical Report RJ 5339 (54809), IBM, 1986.

[BWW+89] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, and A. Aiken. The FL language manual parts 1 and 2. Technical Report RJ 7100 (67163), IBM, 1989.

[BWW90] J. Backus, J. H. Williams, and E. L. Wimmers. An introduction to the programming language FL. In D. Turner, editor, *Research Topics in Functional Programming*, pages 219–247. Addison-Wesley, June 1990.

[DM82] L. Damas and R. Milner. Principle type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[HMT89] R. Harper, R. Milner, and M. Tofte. The definition of standard ML—version 3. Technical Report ECFS-LFCS-89-81, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[HWA+88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. P. Jones, R. Nikhil, M. Reeve, D. Wise, and J. Young. Report on the functional programming language Haskell. Technical Report DCS/RR-666, Yale University, December 1988.

[HWW86] J. Halpern, J. Williams, and E. Wimmers. Good rewrite strategies for FP. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 149–162, January 1986.

[HWW90] J. Halpern, J. Williams, and E. Wimmers. Completeness of rewrite rules and rewrite strategies for FP. *Journal of the ACM*, 37(1):86–143, January 1990.

[HWWW85] J. Halpern, J. Williams, E. Wimmers, and T. Winkler. Denotational semantics and rewrite rules for FP. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 108–120, January 1985.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, January 1993.

[PJW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, January 1993.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, MA, 1977.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, June 1990.

[Wil82a] J. Williams. Notes on the FP style of functional programming. In *Functional Programming and its Applications*. Cambridge University Press, January 1982.

[Wil82b] J. Williams. On the development of the algebra of functional programs. *ACM Transactions on Programming Languages and Systems*, 4(4):733–757, October 1982.

[WW88] J. H. Williams and E. L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 169–179, January 1988.