

FL Language Manual

Introduction

This manual consists of two parts: first, an informal description of the language with some short examples of its use, and second, a formal description of its syntax and denotational semantics. A separate paper [ref] discusses the design choices that led to FL and some of their alternatives. This paper also discusses the mathematical properties of the language.

FL is a function level language; this means that it is based on the use of combining forms, operations for building new programs from existing ones. These program combining operations have mathematical properties that generate many algebraic identities for FL programs. These comprise laws and theorems that can be used to reason about and optimize programs. One purpose of the FL project is to test two interrelated hypotheses: (a) that FL programs can be compiled into programs that run as fast as the best Fortran and Lisp programs (as a consequence of the mathematical properties of FL programs) and (b) that the function level style of programming is a powerful and flexible one, and that it gives programs a useful structure that it is helpful in understanding and reasoning about them.

Overview of the FL language.

In this section we give a brief informal explanation of the main features of the FL language, with examples.

Expressions and values

An FL program is an *expression*. Every expression has a *value*, which may be either a normal or an abnormal value. A normal value may be an atom, that is, a *number*, a *char* (a character), or a *truth value* (*true* or *false*). A normal value that is not an atom may be a *function*, a *sequence* of values, or an element of an abstract data type (whose structure may be hidden). There are two kinds of abnormal values: *exceptions* and $\perp$, *bottom*. Exceptions result

when functions are applied to inappropriate arguments ; $\perp$ denotes the result of a non-terminating computation. When an expression produces an abnormal value we say that it *bombs*, otherwise we say it is *defined*. Abnormal values are persistent and cannot be elements of composite values (e.g., for any abnormal value v and any function f , f of v is v ; a sequence containing v has v as its value). In the following informal discussion we shall assume that no expression bombs (the details of evaluation will be treated fully later).

Application

The simplest expressions are atoms and names of functions. A composite expression may be built by *application*; if e_1 and e_2 are expressions, then the expression $e_1:e_2$ denotes the result of applying the function denoted by e_1 to the value of e_2 . For example, the value of the expression

```
tl:<1,2,3>
```

is the result of applying the primitive function *tail* (denoted by the function name *tl*) to a sequence of three numbers, giving the value $\langle 2,3 \rangle$.

Sequences

A sequence of expressions is also an expression. For example, the expression

```
< tl:<1>, +:<2,3,4> >
```

has the value

```
< <>, 9 >
```

obtained by evaluating the two applications in the original expression. If one of the expressions in a sequence bombs, then the whole sequence bombs (has the value of the first exception in it).

Strings A sequence of chars is called a *string*, thus all operations on sequences can be applied to strings. For example, $\langle \text{'a'}, \text{'b'}, \text{'c'} \rangle$ is a string comprising the three chars 'a' , 'b' and 'c' (chars begin with ' to distinguish them from function names; the char 'a' denotes the character "a"). There is an alternative way to write strings. For example, "abc" is a string that is exactly equivalent to $\langle \text{'a'}, \text{'b'}, \text{'c'} \rangle$; thus $s1:\text{"abc"} = s1:\langle \text{'a'}, \text{'b'}, \text{'c'} \rangle = \text{'a'}$.

Where clauses

An expression may be modified by a **where** clause that defines various function names used in it. For example, the expression

```
f:<1,2,3> where Def f ≡ s1◦t1 end
```

defines f in the expression $f:<1,2,3>$ to be the composition ($\circ$) of the primitives $s1$ (selects the first element of a sequence) and $t1$, with the result that $f:<1,2,3> = (s1 \circ t1):<1,2,3> = s1:(t1:<1,2,3>) = s1:<2,3> = 2$. The **where** clause may enclose a list of definitions (separated by spaces).

The language provides a variety of means for controlling the scope of definitions and for hiding and exporting definitions from lists of definitions within **where** clauses. These are discussed in a later section.

Definitions

A definition assigns a function to a function name within the scope of the definition. Definitions have the form

```
Def leftside ≡ expression
```

where *leftside* can be a function name or various other forms to be described later. For example,

```
Def length ≡ + ◦ α:~1
```

defines *length* to be the function that computes the length of a sequence. Its rightside is built from $+$ and the constant-valued function ~ 1 using the combining forms *composition* and *apply to all* (α). Applying $\alpha:\sim 1$ to a sequence results in ~ 1 being applied to each element (where $\sim 1:x = 1$), thus turning it into a sequence of all 1's; adding these 1's gives the length of the original.

If *DL* is a definition list and **Def name ≡ expression** is one of the definitions in it, then in the expression

```
e where DL end
```

names in *e* can see their definitions in *DL*; for example, if *name* occurs in *e*, it will get its definition from **Def** *name* $\equiv$ *expression* in *DL*. If a name in *e* is not defined in *DL*, then it must be a primitive or get its definition from a larger context -- another **where** clause -- modifying this whole expression or one containing it. All the names occurring in the right sides of definitions in *DL* get their definitions first from definitions in *DL* itself, or if they are not defined there, from the surrounding context.

Combining forms

FL emphasizes the function level approach to defining functions, that of building a new function by applying a *combining form* to existing functions. The four principal combining forms and one predicate combining form are described below.

Composition Using composition ($\circ$) to combine the functions *f* and *g* results in the new function *f* $\circ$ *g*, where, for any argument *x*,

$$(f \circ g):x = f:(g:x)$$

For example, (*s1* $\circ$ *tl*):*x* selects the second element of *x* since *tl* produces the tail of *x* and the primitive *s1* then selects the first element of the tail (*x* must be a sequence with two or more elements or an exception will result).

Constant The constant-valued function $\sim x$ is built from the value *x* and the combining form $\sim$; when applied to any *y* it yields *x*. For example, (~ 2):<1,5> = 2.

Condition The function *p* $\rightarrow$ *f*; *g* is built from any three functions *p*, *f*, and *g* by the combining form condition (indicated by $\rightarrow$ and $;$); the resulting function works as follows:

$$(p \rightarrow f; g) : x = \begin{array}{ll} f:x & \text{if } p:x \neq \text{false, an exception, or } \perp \\ g:x & \text{if } p:x = \text{false} \\ \text{exception or } \perp & \text{otherwise} \end{array}$$

For example, (*isseq* $\rightarrow$ *len*; ~ 0):*x* computes the length of *x* (*len*:*x*) if *x* is

a sequence; otherwise it yields 0 ($\sim 0:x$). The above condition will bomb only if p does or if p is true and f bombs, or if p is false and g bombs.

The one-arm condition $p \rightarrow f$ works similarly, except that it bombs if the predicate p is false:

$$(p \rightarrow f):x = f:x \quad \text{if } p:x \neq \text{false, an exception, or } \perp$$

$$\text{an exception or } \perp \quad \text{otherwise}$$

Construction Construction is another basic combining form; the construction $[...]$ of the functions $f_1, \dots, f_n$ is the function $[f_1, \dots, f_n]$, which when applied to x gives the sequence of all the results of each f_i :

$$[f_1, \dots, f_n]:x = \langle f_1:x, \dots, f_n:x \rangle$$

For example, $[+,-]:\langle 3,2 \rangle = \langle 5,1 \rangle$ and $[s1,t1]:\langle 1,2,3 \rangle = \langle 1,\langle 2,3 \rangle \rangle$. A construction bombs if any of its functions bomb.

The principal combining forms without special syntax. Construction is an example of a combining form that has special syntax $[...]$ to make it more readable. However, it and all the other combining forms can be written as the application of a higher order function to another function or sequence of functions. For example, the function $[f_1, \dots, f_n]$ can be built using the primitive function `cons` as follows:

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

Similarly,

$$f \circ g = \text{comp}:\langle f, g \rangle$$

$$\sim x = K:x$$

$$p \rightarrow f;g = \text{cond}:\langle p, f, g \rangle$$

These applications are in normal form, that is, there is no rule for reducing them to a simpler form as there is, for example, for $+: \langle 3,4 \rangle$, which reduces to 7.

Predicate combining forms

Predicate combining forms specialize in building (total) predicates from (total) predicates. Many predicate combining forms are also pattern combining forms that can be used to build patterns (see below); these include `[]..[]`, `=>` and `^`.

Predicate construction. *Predicate construction* is one such form; the expression

`[p1, ..., pn]:x`

applies the predicate construction (`[]...[]`) of $p_1, \dots, p_n$ to x ; it will yield true only if x is a sequence $\langle x_1, \dots, x_n \rangle$ of exactly n elements and for each i x_i satisfies the predicate p_i ; otherwise, if x is not a sequence, or not of length n , or any element fails to satisfy the corresponding predicate, then the value of the expression is false.

For example, `[isnum, isseq]:x` is true only if x has the form $\langle \text{number}, \text{sequence} \rangle$; otherwise it is false.

Some other predicate combining forms.

<u>Combining form</u>	<u>Use of form</u>	<u>True only for</u>
Seqof	seqof:p	sequences all of whose elements satisfy p.
$\wedge$	$p \wedge q$	values for which both p and q are true.
$\sim$	$\sim:p$	values for which p is false.
$\Rightarrow$	$p \Rightarrow q$	non-empty sequences whose first element satisfies p and whose tail satisfies q.

Patterns

Patterns are constructs that denote predicates but also make definitions. The purpose of a pattern is to describe a structure (that satisfies its predicate) and to define function names so that, if a name occupies a position in the pattern, its function will select an element from that same position in the structure. For example, the pattern

$[x., y.]$

corresponds to the predicate that is true for all pairs $\langle X, Y \rangle$ and it defines x to select X (**Def** $x \equiv s1$) and y to select Y (**Def** $y \equiv s2$). We shall often refer to a pattern as if it were the predicate it denotes; thus we might say that "the pattern $[x., y.]$ is true for pairs" or " $\langle x, y \rangle$ satisfies $[x., y.]$ ", etc.

Elementary patterns The simplest patterns have the form *name.predicate*, with the "." after *name* indicating that *name* is being defined; such a pattern has the predicate *predicate* and defines **Def** *name* $\equiv$ *id* (*id* is the identity function). Omitting the predicate is equivalent to using the predicate **TT** that is always true. For example, the pattern $x.$ has the predicate **TT** and defines x to be *id*.

Composite patterns A more complicated pattern may be built from simpler ones by a pattern combining form (pattern combining forms are a subset of the predicate combining forms, plus composition); the resulting pattern denotes the given combination of the predicates of its constituent patterns and it modifies the definitions they make to reflect their "position" in the new pattern. For example, the pattern

`[x.isnum, y.isseq]`

has the predicate `[isnum, isseq]` (which is true for pairs of the form `<number, sequence>`) and modifies the original definition of `x=id` (from `x.isnum`) to be `x = id.s1` (since `x.isnum` is in the `s1` position in the pattern) and of `y=id` to be `y = id.s2` (since `s2` selects the second element of a sequence and `y.isseq` is in the `s2` position). Similarly,

`[[x.isnum, y.isseq], z.]`

is true for values of the form `<<number, sequence>, value>` and it defines `x`, `y` and `z` to select their counterparts from values of the form `<<X,Y>,Z>` (those that satisfy the pattern). Thus it defines `x = id.s1.s1`, `y = id.s2.s1` and `z = s2` by composing, for example, the definition of `y = id.s2` in the inner pattern with `s1`, the position of the inner pattern in the whole.

Use of patterns

As we have seen, each function that a pattern defines will select a part of its argument corresponding to the position of the function name in the pattern. Furthermore, each such function will be meaningful when applied to any argument that satisfies the pattern, since then the argument must have the same shape as the pattern.

Patterns in conditions. Because a pattern serves as a guard for all the functions it defines, patterns can appear in the predicate position of a condition and the scope of its definitions extends just over the arm or arms of the condition. For example, consider the one-armed condition

`[x.,y.] → [y,x]`

it denotes a function that is defined (i.e., yields a normal value) only on pairs; when applied to $\langle X, Y \rangle$ it produces $\langle Y, X \rangle$:

$$(\llbracket x., y. \rrbracket \rightarrow [y, x]) : \langle X, Y \rangle = [y, x] : \langle X, Y \rangle = \langle y : \langle X, Y \rangle, x : \langle X, Y \rangle \rangle = \langle Y, X \rangle$$

since the pattern is true for pairs and defines $x = s1$ and $y = s2$.

Patterns on the left side of definitions. Patterns can be used on the left side of definitions; this makes clear what the domain of the defined function is and names the key parts of the argument (actually, the functions for accessing them). For example, the definition

```
Def sum_prod ← [x.isnum, y.isnum, z.isnum] ≡ + ◦ [x◦[x,z], x◦[y,z]]
```

indicates that `sum_prod` takes an argument of three numbers x , y and z in the form $\langle X, Y, Z \rangle$ and returns the result $X \times Z + Y \times Z$. This definition is equivalent (when no exception occurs) to the following one with the pattern on the right:

```
Def sum_prod ≡ [x.isnum, y.isnum, z.isnum] → +◦[x◦[x,z], x◦[y,z]]
```

This equivalence holds in general; the following two definitions define equivalent functions f_1 and f_2 when both are defined:

```
Def f1 ← pattern ≡ expression
```

```
Def f2 ≡ pattern → expression
```

That is, for any x for which either $f_1 : x$ or $f_2 : x$ does not bomb, then $f_1 : x = f_2 : x$. If $f_1 : x$ bombs because the pattern is false for x , it will indicate an error from trying to apply " f_1 " to x . But if $f_2 : x$ bombs for the same reason, it will indicate an error from trying to apply a condition (the right side of the definition) to x (the function name f_2 will not occur in the exception).

Lambda expressions

The above function level style of definition (using patterns and condition)

is often hard to use for defining higher order functions. For example, suppose we wish to define the function `curry` such that for any function `f` and values `x` and `y`:

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

(application associates to the left). There is no obvious simple way to write a definition in the above style. For defining such higher order functions the language provides a special form of *lambda expressions* that use patterns in the lambda part. Here is the definition of `curry` using a lambda expression:

$$\text{Def } \text{curry} \equiv \lambda(f.)\lambda(x.)\lambda(y.) f:\langle x,y \rangle$$

In general, a lambda expression has the form:

$$\lambda(\text{pattern}) \text{ expression}$$

The result of applying the expression $\lambda(P)E$ to `x` is `E` with each occurrence of a name `f` defined by `P` replaced by `f:x`, the part of `x` that `f` selects. For example, here is the way `curry:g` works when applied first to 3 and then to 4:

$$\begin{aligned} ((\lambda(f.)\lambda(x.)\lambda(y.) f:\langle x,y \rangle):g):3:4 &\equiv (\lambda(x.)\lambda(y.) g:\langle x,y \rangle):3:4 \\ &\equiv (\lambda(y.) g:\langle 3,y \rangle):4 \equiv g:\langle 3,4 \rangle \end{aligned}$$

since the patterns in the lambda expressions define `f = x = y = id`, thus `f` is replaced in $\lambda(x.)\lambda(y.)f:\langle x,y \rangle$ by `f:g = id:g = g`, etc.

For another example, suppose we wish to define `map2` such that `map2:<f,g,h> = [f∘g, f∘h]`. Here is its definition:

$$\text{Def } \text{map2} \equiv \lambda([f.,g.,h.])[f\circ g, f\circ h]$$

Left sides of definitions for lambda expressions. In a style similar to the use of patterns on the left side of definitions, there is an alternative form for the above definition of `map2` which puts the lambda information on the left side (without the λ):

Def map2([f.,g.,h.]) $\equiv$ [f.g, f.h]

And this is the alternative form for the definition of curry:

Def curry(f.)(x.)(y.) $\equiv$ f:<x,y>

In general, a definition can have the form

Def f(P₁)(P₂)...(P_n) $\equiv$ E

where f is any function name and each P_i is a pattern. Its effect is identical to the definition

Def f $\leftarrow$ P₁^P₂^...^P_n $\equiv$ $\lambda(P_1)\lambda(P_2)\dots\lambda(P_n)E$

even when f bombs. Thus putting the lambda information on the left side of the definition has the effect of guarding the lambda expression so that it will never see an argument that does not satisfy all of the patterns. Without this guard, it is possible for one of the functions defined by a pattern P_i to bomb.

User defined types

The FL type system is based on predicates. Its primitive types correspond to certain primitive total predicates (defined for all values); for example, isnum is true for all numbers and false for all other values, isint is true for integers, ischar for chars, isseq for sequences, etc. Using predicate combining forms, the user can write total predicates for other classes of values; for example, seqof:isint is true for sequences containing only integers and [ischar,isint] is true for pairs whose first element is a char and whose second is an integer.

Representation declarations If the user wishes to introduce a new kind of object and operations on such objects, he must represent these new objects by some already existing ones. For example, if one wishes to introduce complex numbers and represent them by pairs of numbers, the following representation declaration will define a new function mkcomplex that will transform a pair of numbers <x,y> into a complex number c.

```
Rep complex <=> [isnum, isnum]
```

Although $c = \text{mkcomplex}:\langle x, y \rangle$ represents the pair of numbers $\langle x, y \rangle$, it is not this pair itself, it is something of type `complex`; for example `s1:c` yields an exception, not `x`. Here `mkcomplex` is the constructor for the type `complex`. This declaration also defines two other functions for the type, its destructor, `umcomplex` (for un-make complex), and its predicate, `iscomplex`. The constructor `mkcomplex` is defined only for values that satisfy the predicate on the right of the **Rep** declaration, i.e., only for pairs of numbers; the predicate `iscomplex` is defined for all values but is true only for complex numbers; and the destructor `umcomplex` is defined only for complex numbers and yields their representation $\langle x, y \rangle$.

Suppose now that one wants to define complex addition, `cadd`:

```
Def cadd + [iscomplex, iscomplex]  $\equiv$  mkcomplex  $\circ$  vectoradd  $\circ$   $\alpha$ :umcomplex
```

The left side of this definition assures that `cadd` is defined only on pairs of complex numbers. The right side first turns both of these complex numbers into a pair of numbers by applying `umcomplex` to both of them (α :`umcomplex`), then adds the two pairs elementwise (`vectoradd`) and finally turns the resulting pair of numbers into a complex number (`mkcomplex`). We shall see later a way to extend the ordinary arithmetic operators so that they are defined for complex numbers and other user defined types as well as for ordinary numbers.

If one had wished to also define selector functions on complex numbers, where `real:c` is the real part of the complex number `c` and `imag:c` is its imaginary part, one could write

```
Def real  $\equiv$  s1  $\circ$  umcomplex
```

```
Def imag  $\equiv$  s2  $\circ$  umcomplex
```

But since it will be common to want such selectors, one can use a pattern on the right side of a representation declaration to accomplish the same goal. Thus if we write

Rep complex <=> [real.isnum, imag.isnum]

this declaration will define mkcomplex, iscomplex and umcomplex as before, and it will also define real and imag as above, that is, the functions defined by the pattern will be composed with the destructor of the type, umcomplex.

General form of Rep declarations. The declaration

Rep name <=> pattern

defines mkname, unname, isname, plus all the functions defined by pattern (note that pattern may define no functions, in which case it is just a predicate). mkname is defined for any x for which the pattern is true and yields a element of type name. isname is defined for all normal values and is true only for values of type name. unname is defined only for values of type name and yields values for which pattern is true. Any function name defined by pattern is defined for any value of type name and yields the corresponding element of its representation.

