FLIC - a Functional Language Intermediate Code

Simon L. Peyton Jones

Department of Computing Science,
University of Glasgow,
Glasgow,
G12 8QQ.

Email: simonpj@dcs.glasgow.ac.uk

Mike Joy

Department of Computer Science,
University of Warwick,
Coventry,
CV4 7AL.

Email: M.S.Joy@dcs.warwick.ac.uk

Revised August 1989, June 1990

ABSTRACT

FLIC is a Functional Language Intermediate Code, intended to provide a common intermediate language between diverse implementations of functional languages, including parallel ones. This paper gives a formal definition of FLIC's syntax and semantics, in the hope that its existence may encourage greater exchange of programs and benchmarks between research groups.

1. The Aims of FLIC

It is a common observation that high level functional languages are all "sugared" version of the lambda calculus, which suggests that it should be possible to define a common intermediate code into which they could all compile. Widespread use of such an intermediate language would

(i) help alleviate the bad effects of a proliferation of high level languages, without unduly constraining their different strengths;

(ii) give wider access to fast implementations;

(iii) make benchmark programs available to a wider community;

(iv) provide a language in which some compiler passes can be regarded as source to source transformations (e.g. lambda lifting, strictness analysis).

There appears to be widespread support for, and interest in, the development of such an intermediate code. This paper is an attempt to define an intermediate code, called FLIC, in a form acceptable to as many potential users as possible. It has been substantially improved upon during discussions held on the FP electronic mailing list. In the end, however, it represents a personal view, driven by the requirement to establish a standard for use in the GRIP project [5] .

A previous version of this document has appeared as University College London Department of Computer Science Internal Note 2048.

This document describes a slightly revised version of FLIC, incorporating changes discussed in early 1989. A brief summary of the changes appears as section 7 of this document.

Prototype compilers for FLIC already exist at University College London, Imperial College and Chalmers University. FLIC is also being used in experimental work at GEC's Hirst Research Lab, Birmingham University, Warwick University and the Los Alamos National Laboratory.

1.1. Related work

FLIC is similar in spirit to the FP/M intermediate language used at Imperial College [1] but is rather more general and higher-level. The IF1 language is a intermediate language for dataflow compilers, used by the Manchester dataflow project [3] and has a similar purpose to FLIC in a different arena.

Another related language is DACTL (Declarative Alvey Compiler Target Language) [2] which is a more expressive intermediate form than FLIC, and is designed to serve as a target for any declarative language. FLIC is less ambitious, and hence is rather simpler to implement than DACTL.

2. The Objectives of FLIC

Based on the aims outlined above, the next two sections describe first the specific objectives and assumptions of FLIC, and then the detailed design decisions.

(i) FLIC should be completely specific to functional languages, but should not favour any particular language, type system, implementation or architecture. In particular, it should be as suitable for **parallel** machines as for sequential ones.

(ii) It should be possible to modify the front end of almost any compiler for a pure functional language to generate FLIC with only a rather small amount of work. A similar small amount of work should suffice to take FLIC as input to the back end.

(iii) The syntax should contain few constructs and be easily machine readable. Nevertheless, the syntax should specify a printable (as opposed to binary) representation for FLIC programs, for maximum portability.

(iv) As well as having a small syntax, FLIC should have a small semantics. This boils down to saying that FLIC should not have too many primitive (built-in) functions.

(v) A FLIC program should be portable from one FLIC implementation to another, without an unacceptable loss in efficiency.

There is a direct conflict between the last two objectives. Typically, a compiler front end will generate FLIC which contains references to functions which are recognised and treated specially by the back end. To port such a program to another implementation, FLIC definitions for these functions will have to be provided, with a consequent loss in efficiency.

One way to minimise the loss in efficiency is to define a suitable vocabulary of primitive functions as part of FLIC, chosen so that the efficiency loss of using them alone is as small as possible. A very small vocabulary of primitives would give rise to simple but slow implementations; a large vocabulary would produce complex but fast implementations.

The main effort in designing FLIC has gone into picking a set of primitives which preserves opportunities for back-end optimisations without becoming too large and baroque.

3. Major Design Decisions

Before proceeding to a detailed definition of FLIC, this section outlines the major design decisions in FLIC and justifies them.

3.1. Computational Model

FLIC's computational model is the **typed lambda calculus**, augmented with **primitive functions and constants**, using **normal order** reduction.

Normal order evaluation implies **non-strict** semantics. The semantics of any particular program can be made strict by adding calls to the **STRICT** primitive (see Section 5.9). It may, however, be difficult to adapt a back end designed solely for a strict language to implement FLIC.

Some languages, Lisp-based ones in particular, have the notion of an uncurried function; that is a function which is applied to several arguments at once, and for which partial applications are illegal. FLIC does not support this directly; all function applications are curried.

It is clear that the only reasons for wanting uncurried functions are (powerful) efficiency reasons. We can therefore regard uncurried functions as efficient implementations of curried functions, in circumstances where the function is always applied to the right number of arguments, and the proper place for such information is as an annotation (see Section 3.3).

3.2. Primitives

For efficiency reasons, FLIC augments the lambda calculus with a collection of primitive functions, or **primitives**, which must be supported by all FLIC implementations.

Of course, the implementation is free to support each primitive either directly or in terms of other primitives. Indeed, an extremely inefficient FLIC implementation could support the primitives using lambda abstraction alone.

3.3. Annotations

A FLIC program contains two sorts of information:

(i) Semantic: that is, what is to be computed.

(ii) Pragmatic: that is, how to compute it efficiently.

These two kinds of information are kept completely distinct in FLIC, by putting all pragmatic information in lexically distinguishable **annotations**. The meaning of an annotated program is therefore defined to be the meaning of the FLIC program obtained by deleting all the annotations. Annotations may be implementation-specific, whereas the semantic information is entirely implementation-independent. Annotations therefore provide a way of conveying arbitrary information from one part of a compiler to another.

If a compiler compiles code on the basis of information contained in annotations, and the annotations give incorrect or inconsistent advice, then the compiler may perhaps generate incorrect code. For example, if a non-strict function is incorrectly annotated to say that it is strict, a compiler might compile code to use call-by-value.

Hence, **whilst correct annotations may have benign effects, incorrect or inconsistent annotations can have pernicious effects**. This convention allows a compiler to annotate the code consistently with all manner of implementation-specific information, while still allowing the same program to be run on a different system by simply ignoring all the annotations.

3.4. Type-analysis and Type-checking

A FLIC program must be well-typed. By this is meant that no run-time type checking need be performed, because it is guaranteed that functions will only be applied to arguments of the appropriate type. It does **not** mean that FLIC programs are compelled to include type information explicitly, nor does it mean that FLIC programs can be type-checked.

If explicit type information were compulsory, then a language for such type information would have to be provided, which would unreasonably constrain those who are experimenting with the type system itself. Insisting on type-checkability would be worse still, because it would imply a commitment to one particular type system.

Some compilation algorithms (such as strictness analysis) may nevertheless require type information to be present, and many compilers will wish to carry type information in their FLIC programs. This is precisely what annotations are for - semantically irrelevant information which is useful for the compiler.

Legislating that FLIC programs should be well-typed (in the sense that they cannot "go wrong" at run-time) seems at first to rule out SASL, KRC and other weakly typed languages. However, all that needs to be done is to declare a structured type, Universal, to unite numbers, booleans, characters, lists and so on. Then a version of addition can be written which checks the type of its argument, and similarly for all the other functions. In other words, each object gets tagged with its type (by the sum), which is just what happens in the implementation of untyped languages. The reason for this design decision is that it allows typed languages to run with no penalty, and untyped languages to run with the penalty of tagging, which they incur in any implementation.

3.5. Data Types

FLIC has only two built-in data types, the floating-point numbers (Float) and the integers (Int). All other domains are built up using either lambda abstraction (to build functions) or data construction (to build data structures).

3.5.1. Data Construction

Functional programming languages often allow **structured types** to be declared. For example,

tree ::= LEAF num | NODE tree tree

might declare an object of type "tree" to be either a LEAF, in which case it has a single field of type "num", or a NODE in which case it has two fields both of type "tree". Each alternative form is associated with a **constructor** (LEAF or NODE, in this case) and a number of **fields**. The number of fields associated with a constructor is called its **arity**; in the example, LEAF has arity 1 and NODE has arity 2.

The types of lists, characters and booleans can all be defined using structured type declarations, and this is the approach taken in FLIC (efficiency considerations are discussed below).

These structured objects may conveniently be represented by **tagged tuples**. A tagged tuple consists of a non-negative integer **tag**, t, together with an ordered sequence of elements, $x_1,...,x_n$. For semantic purposes, such tagged tuples will be denoted thus:

$< t \mid x_1,...,x_n >$

The constructors of a structured type are associated with distinct integer tags in the range 0..r-1, where r is the number of constructors in the type. Then objects of the type can then be represented by tagged tuples whose size is the arity of the constructor. For example, if LEAF was associated with tag 0 and NODE with 1, the objects of type "tree" would be represented as

$< 0 \mid v >$ or $< 1 \mid t_1, t_2 >$

where v is of type "num" and t_1 and t_2 are of type "tree".

FLIC provides primitives to build and manipulate tagged tuples. Technically, the domain construction involved is the coalesced sum of lifted product. The primitives are:

PACK	construction
UNPACK	lazy destruction
UNPACK!	strict destruction
SEL	extract a field
CASE	perform case analysis
TAG	extract tag

This is not the minimal possible set of primitives, but a smaller set seems to lead to serious loss in efficiency. The details of these primitives are given in Section 5.3.

It is important, for efficiency reasons, for compilers to be able to spot a number of special cases, which are now discussed.

3.5.2. Tuples

Many functional languages provide **tuple** types, which are structured types with only one alternative in the type. In this case no tag is required, and hence a more compact representation may be possible than in the general case.

It would be possible to distinguish tuples using an annotation, but it is more direct to provide a family of functions:

TUPLE	tuple construction
UNTUPLE	lazy tuple destruction
UNTUPLE!	strict tuple destruction
SEL-TUPLE	extract a field

The semantics of these functions is defined directly in terms of their more general counterparts, but their use allows the compiler to use a more efficient data representation. The fact that the program is well-typed ensures that this optimisation does not affect the semantics of the program.

The idea that some knowledge about the number of constructors in the type may allow more efficient implementations is taken up again in the detailed description of the language which follows, by providing an annotation to express the number of constructors in the type (Section 6.2).

3.5.3. Enumeration Types

A boolean type might be declared thus:

bool ::= TRUE | FALSE

All the constructors have arity zero, and this property defines the **enumeration types**. Characters constitute another example of an enumeration type.

As in the case of tuples, a compiler can often use a more compact representation for an object of an enumeration type than for general structured objects, so FLIC provides a specialised set of primitives to handle enumeration types:

ENUM	construction
CASE-ENUM	perform case analysis
TAG-ENUM	extract tag

3.6. Arrays

It is clear that at some stage FLIC should support some array-manipulation primitives, and this is the subject of much current debate. The situation is not yet stable enough for a suitable set to be specified, but this should happen eventually.

4. Core Syntax and Semantics

FLIC's concrete syntax can denote the lambda calculus, plus "let", "letrec", numbers, characters and strings. Lambdas can only have one bound variable, and everything is curried.

The syntax is defined using extended BNF, with angle brackets (<>) to delimit nonterminal symbols, square brackets ([]) to delimit optional symbols, star (*) to indicate zero or more repetitions, and plus (+) to denote one or more repetitions. These meta-symbols are enclosed in single quotes when they are used as terminal symbols. To avoid confusion over quoting, the symbols SINGLE-QUOTE and DOUBLE-QUOTE are used to stand for the terminal symbols **'** and **"** respectively, and the symbol SPACE to stand for a blank space (ASCII 32).

The semantics of FLIC is given in the language of denotational semantics, using **D** to give the semantic interpretation of expressions, and **C** to give the interpretation of primitives and constants.

FLIC code segments and primitives will be written in Helvetica Bold font. A name in italics stands for the semantic interpretation of the corresponding FLIC primitive. In other words,

C[[**PRIM**]] = *PRIM*

for any primitive **PRIM**. In general, all semantic values will be written in italics. For example, the semantics of the FLIC primitive **SEQ** are given by

$SEQ\ \perp\ b = \perp$
$SEQ\ a\ b = b$

(All such equations should be read from top to bottom, picking the first equation which matches the arguments.)

Where no ambiguity arises, explicit reference to semantic mappings **C** and **D** will be omitted.

The semantics given below takes no account of annotations. The semantics of an annotated FLIC program is defined to be the semantics of the program obtained by omitting all annotations in the original program (Section 3.3).

4.1. BNF Syntax

Here is the BNF for the concrete syntax of FLIC. The major constructs are function application, lambda abstraction, let-expressions and letrec-expressions, the last three of which are introduced with single-character keywords.

The top level of a FLIC program is a mutually recursive scope of name-value pairs, one of which must bind the identifier **MAIN**. The value of the program is the value bound to **MAIN**. Thus the top level is a syntactic variant of the letrec-expression.

A FLIC program is a <Program>, thus:

```
<Program>     ::= <Binding>+
<Binding>     ::= <Name> <Simple>

<Expr>   ::= <Simple>
           | <Expr> <Simple>                       /* Application */
           | = ( <Name>+ ) ( <Simple>+ ) <Expr>    /* let <Name>s = <Simple>s in <Expr> */
           | & ( <Name>+ ) ( <Simple>+ ) <Expr>    /* letrec <Name>s = <Simple>s in <Expr> */

<Simple>    ::= ( <Expr> )
              | ( <Abstraction> )
              | <Annotation> <Simple>
              | <Name>
              | <Number>
              | <Character>
              | <String>

<Abstraction>    ::= \ <Name> <Expr>           /* lambda <Name> . <Expr> */
                   | \ <Name> <Abstraction>

<Annotation>     ::= '[' <Name> [<Simple>] ']'
```

Note that abstractions bracket in the conventional way:

(\x x x) is the same as **(\x (x x))**, and

(\x \y \z x y y) is the same as **(\x (\y (\z (x y y))))**.

4.1.1. Let and Letrec

FLIC provides special syntactic forms for let- and letrec-expressions, despite the fact that they can be translated into a form which uses only lambda binding.

The reason for this decision is as follows. Let(rec)-expressions constitute a particular form of abstraction in which the value bound to the newly-defined name is known precisely, in contrast to lambda abstractions where the bound variable is only bound when the abstraction is applied to an argument. Substantial performance improvements are possible if let(rec)s are treated specially by the back end. The provision of syntax for let(rec)s allows the distinction to be made explicit, rather than relying on the compiler to spot a particular pattern of lambda abstraction and application (which is particularly complex in the case of letrecs).

One other feature in the syntax of let(rec)s deserves a mention: the list of names to be bound comes **before** the list of values, rather than interleaved as they do in Lispkit. Thus:

& (X Y) ((CONS 1 Y) (CONS 2 Y))

binds **X** to **(CONS 1 Y)**. This is convenient for compilers, because it means that a name is never encountered before it has been declared.

Note that in the bindings at the top level the names and values are interleaved - see the BNF definition above. The reason for this is that it makes it easier to merge files of FLIC code.

4.1.2. Annotations

An annotation is enclosed in square brackets, and consists of a name followed by an optional expression in FLIC syntax. It annotates the expression that follows it. The name uniquely distinguishes different sorts of annotation from one another.

The (optional) expression inside the annotation must be in FLIC **syntax**, but there is no assumption that it possesses FLIC **semantics**. It follows that names need not be bound, and annotations are not referentially transparent. An annotation could, for example, denote a type-expression.

4.2. Semantics

FLIC uses the standard semantics for the lambda calculus, augmented with let and letrec. **D** is the main semantic function, **P** gives the semantics at the top level, and **C** gives the interpretation of constants and primitives.

$$\mathbf{P}\llbracket\ v_1\ E_1\ \ldots\ v_n\ E_n\ \rrbracket = \mathbf{D}\llbracket\ \textbf{MAIN}\ \rrbracket\ (\mathrm{fix}(\lambda\rho.[v_1{=}\mathbf{D}\llbracket\ E_1\ \rrbracket\ \rho,\ldots,v_n{=}\mathbf{D}\llbracket\ E_n\ \rrbracket\ \rho]))$$

$$\mathbf{D}\llbracket\ k\ \rrbracket\ \rho = \mathbf{C}\ \llbracket\ k\ \rrbracket$$
$$\mathbf{D}\llbracket\ v\ \rrbracket\ \rho = \rho\ v$$
$$\mathbf{D}\llbracket\ \langle\mathrm{Annotation}\rangle\ E\ \rrbracket\ \rho = \mathbf{D}\llbracket\ E\ \rrbracket\ \rho$$
$$\mathbf{D}\llbracket\ E_1\ E_2\ \rrbracket\ \rho = (\mathbf{D}\llbracket\ E_1\ \rrbracket\ \rho)\ (\mathbf{D}\llbracket\ E_2\ \rrbracket\ \rho)$$
$$\mathbf{D}\llbracket\ \backslash\ v\ E\ \rrbracket\ \rho = \lambda w.\ \mathbf{D}\llbracket\ E\ \rrbracket\ \rho[v{=}w]$$
$$\mathbf{D}\llbracket\ \mathbf{=}\ \mathbf{(}v_1\ \ldots\ v_n\mathbf{)}\ \mathbf{(}E_1\ \ldots\ E_n\mathbf{)}\ B\ \rrbracket\ \rho = \mathbf{D}\llbracket\ B\ \rrbracket\ \rho[v_1{=}\mathbf{D}\llbracket\ E_1\ \rrbracket\ \rho,\ldots,v_n{=}\mathbf{D}\llbracket\ E_n\ \rrbracket\ \rho]$$
$$\mathbf{D}\llbracket\ \mathbf{\&}\ \mathbf{(}v_1\ \ldots\ v_n\mathbf{)}\ \mathbf{(}E_1\ \ldots\ E_n\mathbf{)}\ B\ \rrbracket\ \rho = \mathbf{D}\llbracket\ B\ \rrbracket\ (\mathrm{fix}(\lambda\rho'.\rho[v_1{=}\mathbf{D}\llbracket\ E_1\ \rrbracket\ \rho',\ldots,v_n{=}\mathbf{D}\llbracket\ E_n\ \rrbracket\ \rho']))$$

where
k is a constant or primitive
v, v_1 ... v_n are variables
E, E_1...E_n, B are expressions

4.3. Lexical conventions

4.3.1. Layout and comments

Anything between **{** and **}** is treated as a comment, and ignored.

Layout is not significant. Any sequence of white-space characters (spaces, tabs, newlines) is treated as a single separator.

4.3.2. Names

A name is an <Initial-Char> followed by zero or more <Sequel-Char>s.

<Name>	::= <Initial-Char> <Sequel-Char>*
<Initial-Char>	::= *an alphabetic character (upper and lower case distinct)* | *one of the characters:* **!@$%^*_-+~‘:;<>,.?/|** | <Escape-Char>
<Sequel-Char>	::= <Initial-Char> | *a digit* (**0**-**9**)

| *one of the characters:* **=&**

An <Escape-Char> is one of the following sequences:

#n	newline
#s	space
#t	tab
#f	form feed
#d	delete (rubout)
#xdd	the character whose ASCII code is dd (hexadecimal)
##	the character '#', and similarly for any other character

In a name the following characters must be escaped:
()[]#{} SINGLE-QUOTE DOUBLE-QUOTE SPACE

For example, the following are valid and distinct names:

fred
Fred
Name# with# space
#(Bracketed# name#)
#3

4.3.3. Characters

Characters are enclosed in single quotes.

<Character> ::= SINGLE-QUOTE <Sequel-Char> SINGLE-QUOTE

The same escapes work in characters as in names, though only SINGLE-QUOTE and **#** must be escaped. Examples of characters are: **'a'**, **'Z'**, **'#n'**, **'#''**. The last example is the single-quote character.

As mentioned earlier, in the discussion of data types, characters are interpreted as members of an enumeration type:

C[[**'c'**]] = *ENUM n*, where *n* is the ASCII code for c

The function *ENUM* is defined formally in Section 5.3. The ASCII code of a character can, of course, be obtained by extracting its tag using **TAG-ENUM**.

4.3.4. Strings

A string is enclosed in double quotes, and may be empty.

<String> ::= DOUBLE-QUOTE <Sequel-Char>* DOUBLE-QUOTE

The same escapes work inside strings as in names, except that only DOUBLE-QUOTE and **#** must be escaped. For example, the following are valid strings:
"hello", **"New#nline"**, **"Double# quote# #"# in# string"**.

The interpretation of a string is a list of characters:

C[[**"c**$_1$ ... **c**$_n$**"**]] = *CONS* **C**[[**c**$_1$]] (... (*CONS* **C**[[**c**$_n$]] *NIL*) ...)

4.3.5. Numbers

<Number> ::= <Digit>+ [**.** <Digit>*] [**-**] [**E** [**-**] <Digit>+]

Examples of numbers are: **1, 6-, 3.4, 5E-4, 8.9-E3**. Notice that the sign follows the mantissa of negative numbers, so that numbers always start with a digit.

In systems which support integers and floating point numbers as separate types, numbers with no **E** or decimal point are taken to be integers, all others are taken as floats.

4.4. Examples

Here are some example FLIC expressions:

```
3
INT+ 3 4
= (f) ((\ x INT+ x 1)) (f 3)
& (A B) ([Annotation] (CONS 1 B) (CONS 2 A)) A
```

5. Primitives

In this section the primitives are defined that must be supported by any FLIC implementation. They have been chosen to be sufficiently expressive to allow efficient compilation, without becoming too numerous.

The definitions are given for meaningful arguments to the primitives. Primitives given incorrectly typed arguments produce results which are implementation-dependent.

5.1. Families of Primitives

Section 5 defines a number of primitives requiring a minimum number of integer constants as arguments. For instance, the primitive **PACK** requires at least two integer arguments, for example

(PACK 3 4)

These arguments are *mandatory*, and cannot be replaced by expressions which evaluate to integers. The main alternative would have been to use infinite indexed families of primitives, such as, for example, **PACK-3-4** instead of **(PACK 3 4)**, as in previous versions of FLIC.

The following primitives each require a certain number of mandatory arguments.
K, **CASE**, **CASE-ENUM**, **ENUM**, **PACK**, **SEL**, **SEL-TUPLE**, **TUPLE**, **UNPACK**, **UNPACK!**, **UNTUPLE**, **UNTUPLE!**.

Each of the mandatory arguments must (i) be a non-negative integer, and (ii) be a constant (not an expression which evaluates to an integer). This is a *syntactic* requirement, the mandatory arguments must follow *textually* after the primitive. We allow extra brackets to be introduced, thus both

PACK (3) (4) and **((PACK 3) 4)**

are legal FLIC (in the concrete syntax).

We do not disallow the possibility that some FLIC implementations may relax rule (ii) and so accept an extended FLIC language, but portable "standard" FLIC *must* obey them.

5.2. Selection Primitives

There is a primitive **K** which selects one argument out of several. For $n > 0, 0 \leq i < n$:

$$K\ n\ i\ x_0 \ldots x_{n-1} \quad = x_i$$

K must have **two** mandatory integer arguments (See Section 5.1 above).

5.3. Operations on Structured Data

The earlier discussion should have motivated the following definitions of primitives operating on structured data. As in the previous section, **SEL**, **SEL-TUPLE** and **PACK** require two mandatory integer arguments at compile-time, and **UNPACK**, **UNPACK!**, **UNTUPLE**, **UNTUPLE!**, **CASE**, **TUPLE**, **ENUM** and **CASE-ENUM** require one.

For $n \geq 0, i < n, r \geq 0, d < r$

$$PACK\ n\ d\ x_0 \ldots x_{n-1} \qquad = <d \mid x_0, \ldots, x_{n-1}>$$

$$SEL\ n\ i <d \mid x_0, \ldots, x_{n-1}> \qquad = x_i$$
$$SEL\ n\ i\ \bot \qquad = \bot$$

$$
\begin{array}{ll}
\mathit{UNPACK!}\ n\ f\ \langle d \mid x_0, \ldots, x_{n-1} \rangle & = f\ x_0\ \ldots\ x_{n-1} \\
\mathit{UNPACK!}\ n\ f\ \perp & = \perp \\
 & \\
\mathit{UNPACK}\ n\ f\ e & = f\ (\mathit{SEL}\ n\ 0\ e)\ \ldots\ (\mathit{SEL}\ n\ (n\text{-}1)\ e) \\
 & \\
\mathit{CASE}\ r\ e_0\ \ldots\ e_{r-1}\ \langle d \mid x_0, \ldots\ x_{n-1} \rangle & = e_d \\
\mathit{CASE}\ r\ e_0\ \ldots\ e_{r-1}\ \perp & = \perp \\
 & \\
\mathit{TAG}\ \langle d \mid x_0, \ldots, x_{n-1} \rangle & = d \\
\mathit{TAG}\ \perp & = \perp
\end{array}
$$

In the special case of tuples, the following definitions hold, for $n \geq 2,\ 0 \leq i < n$

TUPLE n	:	*PACK n 0*
SEL-TUPLE n i	:	*SEL n i*
UNTUPLE n f t	:	*UNPACK n f t*
UNTUPLE! n f t	:	*UNPACK! n f t*

Notice that the semantics of the tuple-specific functions are defined directly in terms of the semantics of their more general counterparts.

In the special case of enumeration types, the following definitions hold, for $d \geq 0,\ r \geq 1$

ENUM d	:	*PACK 0 d*
CASE-ENUM r	:	*CASE r*
TAG-ENUM	:	*TAG*

Whilst the semantics of these specialised primitives is given in terms of their more general counterparts, they must be used consistently. Objects built with the construction primitive of one family (general, tuple or enumeration type) can only be examined and taken apart with other primitives from that same family. In fact, the specialised families are best regarded as an annotations of the general primitives (see Section 6.2).

5.4. Booleans

The boolean truth-values are denoted using the names **TRUE** and **FALSE** and have the interpretations

TRUE = ENUM 1
FALSE = ENUM 0

There are the usual boolean operators **IF**, **NOT**, **AND**, **OR** and **XOR**:

IF c x1 x2	*= CASE-ENUM 2 x2 x1 c*
NOT x	*= CASE-ENUM 2 TRUE FALSE x*
AND x y	*= CASE-ENUM 2 FALSE y x*
OR x y	*= CASE-ENUM 2 y TRUE x*
XOR x y	*= AND (OR x y) (OR (NOT x) (NOT y))*

5.5. Equality

In many functional programs the availability of polymorphic equality and comparison operators makes programming considerably simpler, particularly when building lookup tables or trees. FLIC therefore provides polymorphic equality and comparison primitives,

POLY=, **POLY>**, **POLY<**, **POLY<=**, **POLY>=**, **POLY!=**,

which may be applied to any data object constructed from the built-in types (Int and Real) using the data construction operations. They are not applicable to functions, or to data structures containing functions; this is another aspect of the assumption that programs are well-typed.

The comparison primitives induce a total order on each data type, but an implementation is otherwise free to choose whatever ordering it pleases. For example, an implementation may simply perform lexicographic comparison of representation, which may induce a different ordering for the integers than that induced by the integer comparison primitives.

5.6. Numbers

The usual arithmetic and relational operators over the integers are provided:

> **INT_** (unary negation), **INT+**, **INT-**, **INT***, **INT/**, **INT%** (remainder),
> **INT<**, **INT<=**, **INT=**, **INT>=**, **INT>**, **INT!=** (not equals).

All are strict in their arguments. Division rounds towards zero, and remainder takes the same sign as the divisor.

The corresponding family is provided for floating point numbers: **FLOAT_**, **FLOAT+**, etc. with the addition of **FLOAT^** ("to the power of").

In addition, there are the following exponential and transcendental functions, which operate on floating point numbers:

> **SQRT**, **SIN**, **COS**, **TAN**, **ARCSIN**, **ARCCOS**, **ARCTAN** (for the inverse trigonometric functions, result lies between $\pm\pi/2$),
> **EXP** (natural exponential), **LN** (natural log).

INT->FLOAT and **FLOAT->INT** convert between integers and floating point numbers, the latter rounds toward zero.

Integers should have at least 32-bit precision. Real numbers, when represented as $\pm r\mathrm{E}10^{\pm n}$, with $0 \leq r < 1$, should have $n \leq 37$, and r be representable to an accuracy of at least 6 decimal places. Thus, we expect reals to fit into a "standard" 32-bit address space, and to conform to IEEE floating point (short) format. Particular implementations may provide greater accuracy/range for integers and reals; in such cases, it would be desirable for these features to be documented. Implementations are encouraged, but not compelled, to check for exceptions, such as overflow and division by zero, but the effect of such exceptions is undefined.

5.7. Lists

The primitives **CONS**, **HEAD**, **TAIL**, **IS-NIL** and **NIL** manipulate lists, which are defined as a structured type in which **NIL** has tag 0, and **CONS** has tag 1.

CONS = PACK 2 1
NIL = PACK 0 0

HEAD x = CASE 2 ABORT (SEL 2 0 x) x
TAIL x = CASE 2 ABORT (SEL 2 1 x) x
IS-NIL x = CASE 2 TRUE FALSE x

5.8. Sequencing

Those who have written interactive functional programs will be familiar with the phenomenon that the program prints something like:

> Enter data: Result is:

where the initial segment of the result message comes out before the user has entered his input. This is quite right, since the "Result is:" message does not depend on the input, but to get the right sequential operational behaviour we want to impose a dependence. This can be achieved with the **SEQ** primitive, defined by the equations:

$$SEQ\ \perp\ b = \perp$$
$$SEQ\ a\ b = b$$

The implementation of **SEQ** is simple - evaluate the first argument, then evaluate and return the second.

On a parallel machine the evaluation of both arguments may proceed in parallel, but of course the second must not be returned until the first has terminated.

5.9. Strict Functions

In a strict language (that is, a language in which all functions are strict) the equation

$f \perp = \perp$

must hold for all functions, regardless of whether f actually uses its argument or not. The **STRICT** primitive is designed to allow this to be expressed neatly. **STRICT** is defined by the following semantic equations:

$STRICT\ f \perp = \perp$
$STRICT\ f\ x = f\ x$

It is normally used at the point of definition of the strict function, by applying **STRICT** to a lambda abstraction. For example, here is a let-binding which defines **SEQ** in terms of **STRICT**

= (SEQ) ((STRICT (\x \y y)))

(Of course, **STRICT** can be defined in terms of **SEQ** also.) In a sequential implementation **STRICT** can be implemented by evaluating the argument first; in a parallel implementation the constraint is that *f* must not return until evaluation of the argument has completed.

5.10. Input and Output

FLIC takes a minimal approach to I/O.

The result of the whole program is assumed to be its output, which may be tagged to indicate the device to which the output should be sent.

The function **INPUT** takes a list of characters, which is intended to denote a file name or device name, and returns a list of characters, which is intended to be the contents of that file or device.

INPUT s = <the list of characters referenced by s>

The result of **INPUT** is implementation-dependent, and may be time-dependent (that is, it may give different results depending on when the program is run).

5.11. Errors

FLIC makes no attempt to specify error handling (this is a weakness). If a program contains an error, the behaviour of the implementation is not specified; in particular, it may deliver an incorrect result without other warning.

For example, type errors may not be detectable at run-time, because objects of different types may share a common concrete representation. The polymorphic equality primitive is a similar case. Arithmetic overflow, underflow or division by zero may not necessarily be detected by an implementation.

There is one exception: the primitive **ABORT** is guaranteed to cause a run-time error indication if it is evaluated.

$ABORT = \perp$

6. Annotations

This section suggests annotations for two particular situations.

6.1. Strictness Information

Compilers for non-strict languages sometimes perform **strictness analysis** to discover when a function is sure to evaluate its argument, in which case the analysis phase annotates the function with strictness information.

There is an important difference between these strictness annotations and the use of the **STRICT** function described earlier: strictness annotations can be discarded without changing the meaning of the program. In contrast, discarding applications of **STRICT** might make a program terminate which did not do so before.

There are two places where strictness information is useful: at a function definition and at a function application. The annotation **[!]** is used in both cases. An application **(E1 E2)** is annotated as strict like this:

[!] (E1 E2)

This indicates that the function **E1** is strict, so that **E2** can safely be evaluated before (or in parallel with) applying **E1** to **E2**. A lambda abstraction may be annotated thus

[!] (\ x E)

to indicate that the function denoted by **(\x E)** is strict.

For example, the expression

= (f) ([!](\ x [!](\ y (INT- y x))))
[!] ([!] (f x) y)

6.2. Tag Sizes

For the efficient implementation of structured objects, it may be desirable to vary the representation depending on how many alternatives there are in the type, which determines the maximum tag size. This can be achieved by annotating the primitives which manipulate structured data.

The annotation **[TAGS** r**]**, where r is a positive integer, indicates that the primitive thus annotated is being used to manipulate a structured data object which has exactly r alternative forms. For example, a FLIC let-binding for **CONS** could be

= (CONS) (([TAGS 2] PACK 2 1))

The **[TAGS** r**]** annotation can be applied to any of the primitives **PACK**, **UNPACK**, **UNPACK!**, **SEL**, **CASE**, **TAG**, **ENUM**, **CASE-ENUM** and **TAG-ENUM**. (The tuple-manipulation primitives are implicitly annotated with **[TAGS 1]**.)

The annotation must be performed consistently, otherwise a variant might be **PACK**ed in an efficient representation, but **UNPACK**ed by a function which did not expect that representation. As always, inconsistent or incorrect annotations may mislead the compiler (see section 3.3). Notice that if **[TAGS** $\mathbf{r}_1$**]** annotates **CASE** $\mathbf{r}_2$ then necessarily $\mathbf{r}_1 = \mathbf{r}_2$; but the annotation may still be necessary to indicate that the argument is in an efficient representation.

7. Changes from Previous Version of FLIC

The principal changes are as follows:

* There is now only one LET (**=**) production, with syntax analogous to that for LETREC (**&**).
* At the top level, FLIC becomes a sequence of name-value pairs, of which one must define the name **MAIN**, which is the value of the program. The names so defined are mutually recursive.
* The comment delimiters have been changed to **{** and **}**.
* Names of primitives have been amended so that families (such as **PACK-**n**-**m) are represented without the character **-** (thus **(PACK** n m**)**)
* Minor lexical amendments have been made, principally to ensure no inconsistencies arise from the above changes.

The June 1990 revision simply adds the functions

* **FLOAT^**, **TAN**, **ARCSIN** and **ARCCOS**.

8. Appendix - Domains for Data Construction

This appendix consists of a more discursive discussion of the domains used in FLIC's data constructors, and the reasons for their choice.

As mentioned above, FLIC has only two built-in data types (Int and Float), and all other data types are built using the following domain constructions:

* Arrow, to construct functions.
* Cartesian product, to construct tuples.
* Sum, to construct variants.

Functions are constructed using lambda abstraction, and manipulated using function application. Objects from sum and product domains are constructed and manipulated using primitive FLIC functions, and this appendix is mainly devoted to a discussion of sum and product domains, and the primitive operators required to use them.

8.1. Product Domains

An element of a product domain is a tuple, an ordered set of values. For example, if D_0 and D_1 are domains, then

$$D_0 \times D_1$$

is the domain of ordered pairs (2-tuples) of elements from D_0 and D_1 respectively. This is a direct analogue of the cartesian product of sets:

$$D_0 \times D_1 = \{ \langle d_0, d_1 \rangle \mid d_0 \in D_0 \text{ and } d_1 \in D_1 \}$$

using the notation $\langle d_0, d_1 \rangle$ to stand for the ordered pair of d_0 and d_1. Since a computation whose result is a tuple might not terminate, we actually get the **lifted** domain $(D_0 \times D_1)_\perp$, where

$$(D_0 \times D_1)_\perp = (D_0 \times D_1) \cup \{\perp\}.$$

Notice that in this domain, the elements $\langle\perp,\perp\rangle$ and $\perp$ are distinct. This extends naturally to products of more than two domains:

$$D_0 \times ... \times D_{n-1} = \{ \langle d_0, ..., d_{n-1} \rangle \mid d_i \in D_i \}$$

8.2. Operators for Products

The natural operator to construct tuples is **TUPLE**, where

$$TUPLE\ n\ x_0\ x_1\ ...\ x_{n-1} = \langle x_0, x_1, ...\ x_{n-1} \rangle$$

Notice that *TUPLE n* does not evaluate its arguments.

Why have a family of n-ary tuple constructors, rather than providing a single 2-tuple constructor (**PAIR**), and building n-tuples out of 2-tuples? The reason for preferring **TUPLE** can be expressed either pragmatically or semantically.

Beginning with the former, suppose an implementation constructed a 3-tuple, t, using the expression **(PAIR x (PAIR y z))**. Then if **SECOND** selects the second component of a 2-tuple, **(SECOND t)** would be a perfectly legal construct, yielding a 2-tuple, which could be passed around as a first class object. But this would defeat a clever implementation which constructed the 3-tuple as a single construct, and only allowed the selection of single components. Put another way, there is no way of telling whether a composition of **PAIR**s is meant as an n-tuple or as a composition of **PAIR**s!

In terms of semantics, the 3-tuple domain built by a composition of two **PAIR**s

$$((D_0 \times D_1)_\perp \times D_2)_\perp$$

is not isomorphic to the domain built with **TUPLE 3**

$$(D_0 \times D_1 \times D_2)_\perp$$

since the former contains elements such a <⊥,t>, which do not appear in the latter. FLIC therefore provides n-ary tuple constructors.

How should tuples be taken apart? Two alternatives suggest themselves:

(i) Take the tuple apart and apply a specified function to the components:

$$UNTUPLE\ n\ f < x_0,\ x_1,\ ...,\ x_{n-1} > = f\ x_0\ x_1\ ...\ x_{n-1}$$

(**UNTUPLE** is thus just an uncurrying operation).

(ii) Define an operator **SEL** which takes three arguments, an integer n, an integer i and an n-tuple t, and select the i'th element from t. The reason that a distinct selection function is needed for each size of tuple is that the size of the tuple may be required to locate where the i'th element is held.

A tricky point about **UNTUPLE** is whether or not it is strict in its third argument, the tuple. A straightforward (and efficient) implementation of **UNTUPLE** would evaluate its second argument, expecting to find a tuple, and apply its first argument (the function) to its n components. Thus its complete semantics would be

$$UNTUPLE\ n\ f < x_0,\ x_1,\ ...,\ x_n > \quad = f\ x_0\ x_1\ ...\ x_n$$
$$UNTUPLE\ n\ f\ \bot \quad = \bot$$

Alternatively, a less strict semantics might be:

$$UNTUPLE\ n\ f\ t = f\ (SEL\ n\ 0\ t)\ ...\ (SEL\ n\ (n\text{-}1)\ t)$$

This is less efficient, but lazier, since t is evaluated later.

This is not a trivial question - see Phil Wadler's paper [6] . Briefly, the lazier version corresponds to lazy pattern matching, and can be vital to good program behaviour. This is, however, a matter of taste, and FLIC should not legislate on this subject.

The solution adopted here is to define all three operators: selection (**SEL**), strict untupling (**UNTUPLE!**) and lazy untupling (**UNTUPLE**). It is up to each implementation to decide which ones to implement directly, and which (if any) to implement in terms of the others.

8.3. Sum Domains

A sum domain contains labelled elements from each component domain. For example,

$$D_0 + D_1$$

contains all the elements from D_0 and all the elements from D_1, but labelled so that by looking at an element from the sum domain it is possible to tell from which component domain it came (imagine the elements of D_0 being painted blue, and the elements from D_1 being painted red).

We use the term "variant" (or "variant object") to describe an element of a sum domain, by analogy with Pascal's variant records (there appears to be no generally agreed term for such things). Variants may be represented by tagging the values of the component domains. Formally, we can write

$$D_0 + D_1 = \{0{:}d_1 \mid d_1 \in D_0\} \cup \{1{:}d_2 \mid d_2 \in D_1\}$$

using the notation "t:v" to mean "the value v tagged with t", and using the non-negative integers as tags. Sums can be extended in the natural way to n-ary sums, and we define the "width" of the sum domain to be the number of domains which are being added together.

Most typed functional languages allow user defined data types, such as

tree : LEAF num | NODE tree tree

This can be modelled by a domain which is a sum of products:

tree = Num + (tree $\times$ tree)

Objects of such types can be represented as tagged tuples, LEAFs being tagged with 0, and NODEs with 1. This fits with the idea that the sum tags the tuples made by the product.

8.4. Sum-of-Product Domains

Domain theory provides us with two sorts of sum: "separated sum" and "coalesced sum". In the latter, the bottom elements of all the domains are identified together with the bottom of the sum domain; in the former, a new bottom element is adjoined, below the bottom elements of the component domains. Thus,

$0{:}\perp$ is a member of $D_0 + D_1$

where we are using separated sum, but not where we are using coalesced sum.

These theoretical distinctions have immediate corresponding physical manifestations. For example, the NODEs defined above are tupled and tagged all-at-once, so that the element $1{:}\perp$ does not exist, so we must be using a coalesced sum.

At first it seems attractive to provide FLIC primitives for sum domains and product domains separately, and treat sum-of-product domains by composition. Unfortunately, this leads to a separated sum construction, since there is no way to avoid the possibility of construction of values such as $1{:}\perp$.

It turns out that this difference actually has a significant impact on efficiency. If a variant tuple is built with a single sum-of-product constructor, it may be assumed that as soon as the tag of the object is available, then the components of the tuple are also available (though not necessarily evaluated). This allows a more eficient representation of variant tuples, as a tagged vector of storage locations.

We have therefore come to the conclusion that FLIC should provide primitives for operating on product domains, rather than treating sum and product domains separately and sum-of-products by composition.

8.5. Operators for Sum-of-Products Domains

The tag which distinguished variant tuples from one another within a domain is a non-negative integer, ranging from 0 to r-1 where r is the width of the domain. Variant tuples are constructed with the operator **PACK**:

$$PACK\ n\ d\ x_0\ ...\ x_{n-1} = \langle d \mid x_0\ ,...,\ x_{n-1} \rangle$$

where d is the tag, n is the number of components in the tuple, and the notation $\langle d \mid x_0, ..., x_{n-1} \rangle$ denotes the tagged tuple $d{:}\langle x_0, ..., x_{n-1} \rangle$. **PACK** is rather like **TUPLE**, except that it tags the tuple it builds.

When taking variants apart, we should provide a case for each possible tag, which suggests the **CASE** family, where

$$\begin{aligned} &CASE\ r\ e_0\ e_1\ ...\ e_{r-1}\ \ \langle d \mid x_0\ ,...,\ x_{n-1} \rangle && = e_d \\ &CASE\ r\ e_0\ e_1\ ...\ e_{r-1}\ \ \perp && = \perp \end{aligned}$$

CASE evaluates its last argument (the tagged tuple), extracts the tag d, and returns its d'th argument; r is the width of the sum domain. This is rather like a multi-way version of the ordinary two-way conditional.

Having decided in which component of the sum the variant lies, it can be taken apart using **UNPACK**:

$$\begin{aligned} &UNPACK\ n\ f\ \langle d \mid x_0\ ,...,\ x_{n-1} \rangle && = f\ x_0\ ...\ x_{n-1} \\ &UNPACK\ n\ f\ \perp && = \perp \end{aligned}$$

Notice that it is assumed that the number of components in the variant matches the unpack operator being used, and no tag check is performed (remember, FLIC assumes that programs are well-typed).

Together these can implement pattern-matching in an efficient manner (see Peyton Jones' book [4] for a full explanation). For example,

f (LEAF x) = g x
f (NODE t_1 t_2) = h t_1 t_2

could compile to

f x = CASE 2 (UNPACK 1 g x) (UNPACK 2 h x) x

assuming that LEAFs are tagged with tag 0, and NODEs with tag 1.

For the efficient implementation of variants, it may be desirable to vary the representation depending on how many elements there are in the sum, which determines the maximum tag size. This efficiency improvement is provided by means of annotations (see above).

References

1. A.J. Field, *The Compilation of FP/M Programs into Conventional Machine Code,* Department of Computer Science, Imperial College, London, UK (1985).
2. J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep, "DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction," SYS-C87-03, School of Information Systems, University of East Anglia, Norwich, UK (1987).
3. J.R. Gurd, C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM,* 28, 1, pp. 34-52 (1985).
4. S.L. Peyton Jones, *The Implementation of Functional Programming Languages,* Prentice Hall, London, UK (1987). ISBN 0-13-453325-9 (pbk).
5. S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie, "GRIP - A High-Performance Architecture for Parallel Graph Reduction" in *Functional Programming Languages and Computer Architecture,* ed. G. Kahn, pp. 98-112, Springer-Verlag, Berlin, DE (1987). ISBN 3-540-18317-5; Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.
6. P.L. Wadler, *A Splitting Headache and its Cure,* Programming Research Group, Oxford, UK (1985).