A Standard for a Graph Representation for Functional Programs

Mike Joy

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

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

Tom Axford

Department of Computer Science,
University of Birmingham,
Birmingham,
B15 2TT.

Email: T.H.Axford@cs.bham.ac.uk

June 1987

ABSTRACT

The data structures used in the authors' functional language graph reduction implementations are described, together with a standard notation for representing the graphs in a textual format. The graphs employed are compatible with FLIC and with the functional languages in use at Birmingham and Warwick. The textual format is designed to be transmittable easily across networks.

1. Introduction

Work is progressing at Warwick and at Birmingham into graph reduction techniques for functional programs. In order to facilitate cooperation between the establishments it has been necessary to standardise the data structures used and certain aspects of their implementations, as well as a format for transmitting them across networks. At Birmingham, a functional language currently used for teaching purposes [1] resembles the languages *SASL* [6] and *Pifl* [3] currently in use at Warwick. Both are translated into graphical form before evaluation. In the case of the SASL interpreter, the intermediate code [5] produced is *FLIC*. The forms of the graphs eventually produced were found to be almost identical. Therefore the need was isolated for a standard definition for the graphs. It is envisaged that any functional language could be translated to such a graph and then executed by either the Birmingham or the Warwick machine. In order to move graphs between the two machines a standard for describing the graphs in a machine-independent and printable way was also found to be necessary.

The graphical representation described in this paper may be thought of primarily as an internal representation of the intermediate code, FLIC. It is more general than that, however. Although the FLIC operator set is preferred, other sets of basic operators may be used if required. If portability is to be achieved, definitions of these operators in terms of FLIC operators should be provided. Alternatively, it is often not difficult to write a translator program to transform a graph using one set of operators into an equivalent graph using another set of operators (partial reduction of the graph incorporating the definitions of the old operators in terms of the new operators is usually quite effective).

For these reasons we consider it more important to standardise the structure of the graph and the types of nodes allowed and their meaning. The basic data types and structures also follow FLIC, except that we allow the data structures of cartesian sum and product domains (i.e. unions and tuples, respectively) to be used separately, while FLIC permits products and sums of products only. It is not possible for the user to extend this set of basic types: they must represent their own data types and structures purely in terms of those provided. They can, however, annotate objects so that additional information is not lost (the annotation must not affect the meaning, but it can provide guidance as to efficient implementation and other pragmatic information).

2. The Graph

A functional program is represented as a connected, directed, and possibly cyclic, graph in which each node has out-degree 0, 1 or 2. There are eleven different node types, described below.

2.1. Node Types

2.1.1. Integer and Real

Nodes of types *integer* and *real* represent known constants of the usual types integer and real, respectively. The value of the constant is stored in the node, which has out-degree 0. No limits are specified for maximum integer size or accuracy of real numbers, although a particular machine implementation will, of course, have such limits.

2.1.2. Operator

Nodes of type *operator* represent basic operators which are predefined as part of the language, for example, the ordinary arithmetic operators. A code for the operator is stored in the node, which has out-degree 0. See the appendix below for a description of some of the standard operators.

2.1.3. Apply

Nodes of type *apply* represent function application. These nodes have out-degree 2, the left pointer is to the function and the right pointer is to the argument. Functions are assumed to be curried if they take more than one argument.

2.1.4. Lambda

This type of node represents lambda abstraction and has an out-degree of 2. The left pointer is to the bound variable and the right pointer is to the body of the lambda abstraction (which may be of any type).

2.1.5. Variable

Nodes of type *variable* represent the bound variables of lambda abstraction, but can also be used to represent free variables (if the programming language which is being represented allows completely free variables). They have an out-degree of 0.

2.1.6. Sum

These nodes represent cartesian sums (discriminated unions), and have an out-degree of 1. The integer tag is kept in the node, together with a single pointer to the value associated with the sum (which may be of any type).

2.1.7. Product

These nodes represent cartesian products (tuples) and have an out-degree of 2. The left pointer is to the first element of the product (which may be of any type) and the right pointer is to the rest of the product (i.e. another node of type *product*), or it is a null pointer. The number of elements in the product is also stored in the node. A 0-tuple is represented by a null pointer, which has the address 0.

2.1.8. Undefined

Nodes of this type represent ⊥ (bottom). They have out-degree 0.

2.1.9. Unique

Nodes of type *unique* represent unspecified primitive values. Thus a test for equality between a *unique* node and any other well-defined object will always give false. Each *unique* node is equal only to itself. It is useful for representing objects such as "nil" (list terminator) explicitly.

2.1.10. Recursive Reference

A node of type *recursive reference* is used whenever a pointer introduces a cycle into the graph (i.e. in recursive definitions). Semantically, the *recursive reference* node (with out-degree 1) simply denotes the node to which it points. Its presence is solely a label that the pointer here is different (we call it a weak pointer). Graph traversal and memory management algorithms that would not work on cyclic graphs can then be implemented simply by ignoring weak pointers (see next section).

2.2. Graph Structure

One of the authors [2] has shown that a simple reference counting scheme for cyclic graphs of functional programs is practicable. The graph structure supports this scheme of reference counting (although it does not require it: mark-scan garbage collection schemes could be used if preferred, and the reference counts ignored).

If the reference counting scheme is to be used, the graph must satisfy certain requirements, the main one being:

(i) If weak pointers (i.e. pointers from *recursive reference* nodes) are ignored, the graph is acyclic and connected.

A further condition is required to ensure that graph reduction operations do not generate graphs which break this rule:

(ii) There must be exactly one point of entry to any cycle, which will be the node pointed to by one or more weak pointers. That is, there must be only one node in the cycle pointed to by weak pointers and that node must also be the only node in the cycle which is pointed to by any nodes outside the cycle.

Axford has shown that these conditions are not difficult to satisfy and that, provided they are satisfied, reference counting of strong pointers only is all that is needed for safe memory management.

2.3. Portability

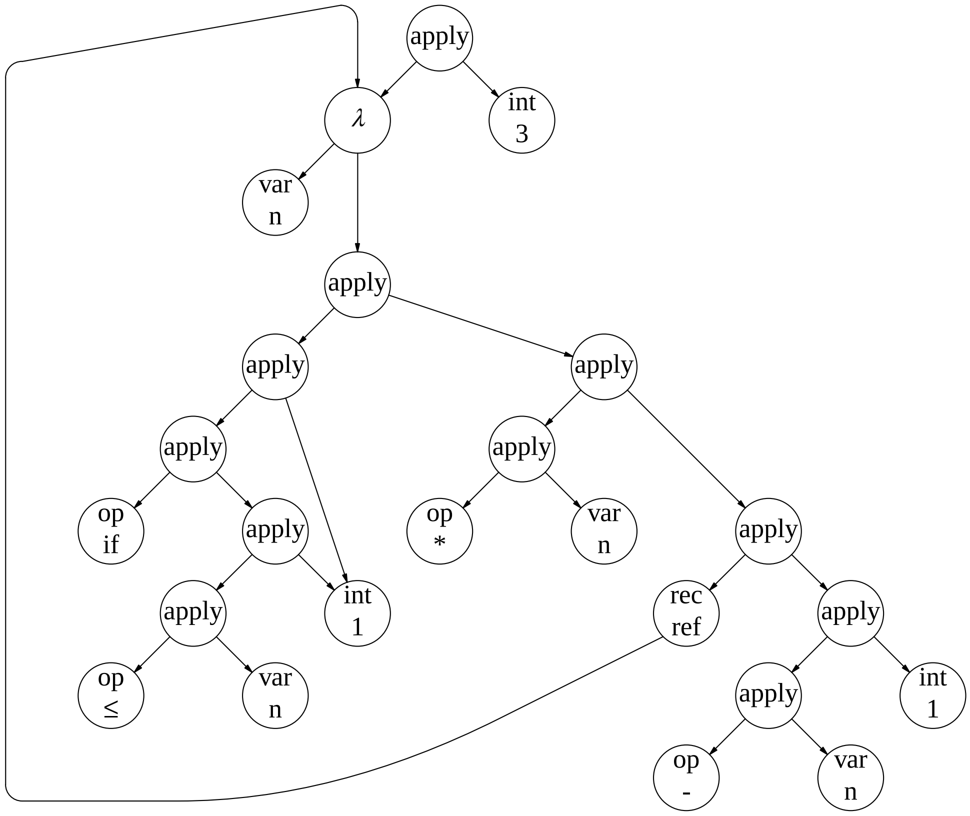
Since the operators used in GCODE may be supplemented by locally defined operators, it is suggested that GCODE may be thought of at two distinct levels - (i) with only the standard operators defined, so that user-defined functions must be defined by lambda abstraction, and (ii) with "other" operators defined, so that it can be used to communicate between machines with a common knowledge of non-standard operators.

2.4. Example

Consider the graph representing "factorial 3" *before* any graph reduction has taken place. Assuming we have

def fac = λn. **if** n ≤ 1 **then** 1 **else** n * fac (n-1)

the graph becomes (or *could* become - it is not unique, due to possible code-sharing):



The nodes representing the variable "n" are shared, but for clarity we have shown them as separate in this diagram.

3. The "Printable" Format for Describing a Graph

The structure for the textual representation of a graph is a sequence of lines, each representing a separate node in the graph, of the form:

address type usage_count annotation "name" [other fields]

where the fields are separated by blanks or tabs. The field *address* is an unsigned integer representing the storage location of the node. The field *type* is an unsigned integer representing the type of the node (integer, real, application, etc.). The field *usage_count* is an unsigned integer used for garbage collection purposes (reference counts, etc.), and represents the number of *strong* pointers to that node in the graph. The field *annotation* is an integer currently not assigned, but may be used in the future by particular implementations. The meaning of a program should be unchanged if all the annotations are ignored. The "name" field is a character string naming the node, usually null.

For example, the node at address 111 which is an apply node called "fred", with left and right descendants at addresses 222 and 333 respectively, usage count of 1, with no annotation, would be represented by:

111 5 1 0 "fred" 222 333

Standard C language [4] conventions apply in the name, thus for example a node called "aSilly\012\013\n\tName" would be acceptable. Similarly all other fields use the appropriate "C" lexical conventions.

The available types are

Undefined	0
Integer	1
Real	2
Sum	3
Product	4
Apply	5
Recursive Reference	6
Operator	7
Variable	8
Lambda	9
Unique	10

3.1. The Type "Integer"

address 1 *usage_count annotation* "name" *value_of_the_integer*
where *value_of_the_integer* is a (signed) integer. No restriction on the size of the integer is imposed, though machine dependencies will naturally come into play.

3.2. The Type "Real"

address 2 *usage_count annotation* "name" *value_of_the_real_number*
where *value_of_the_real_number* is a real number written using "C" conventions.

3.3. The Type "Sum"

address 3 *usage_count annotation* "name" *tag address_of_value*
where a sum domain is considered as associating with a node an unsigned integer *tag* in a finite range, and *address_of_value* is the address of the node which is tagged.

3.4. The Type "Product"

address 4 *usage_count annotation* "name" *size first second*
where a product domain is thought of as a tuple, implemented as a linked list. *size* is the size of the tuple, *first* is the address of the head of the tuple and *second* is the address of the tail. The address of the null tuple is 0, rather than an explicit 0-tuple node.

3.5. The Type "Apply"

address 5 *usage_count annotation* "name" *left right*
where *left* and *right* are the addresses of the descendants of the apply node. We can think of *left* as a function taking one argument (*right*).

3.6. The Type "Recursive Reference"

address 6 *usage_count annotation* "name" *recref*
where *recref* is the address of the node which is used recursively, that is, which is pointed to by a *weak* pointer.

3.7. The Type "Operator"

address 7 *usage_count annotation* "name" *operator qualifier*
where *operator* is an integer (at least 16-bit) representing a predefined operator, and the last field is an integer for use in the case where there is a family of operators all of the same name (such as the "SELECT-i" of FLIC).

3.8. The Type "Variable"

address 8 *usage_count annotation* "name"
and is used for bound or free variables.

3.9. The Type "Lambda"

address 9 *usage_count annotation* "name" *bv body*
where *bv* and *body* are integers which are the addresses of the bound variable of the lambda node and the body respectively.

3.10. The Type "Unique"

address 10 *usage_count annotation* "name"
which is an unspecified primitive value different from all other values.

3.11. The Type "Undefined"

address 0 *usage_count annotation* "name"
which is a non-strict ⊥.

3.12. Example: The Factorial Function

1	5	1	0	""	3	21
2	6	1	0	""	3	
3	9	1	0	"factorial"	11	4
4	5	1	0	""	5	13
5	5	1	0	""	6	12
6	5	1	0	""	7	8
7	7	1	0	""	16176	
8	5	1	0	""	9	12
9	5	1	0	""	10	11
10	7	1	0	""	8978	
11	8	4	0	"n"		
12	1	2	0	""	1	
13	5	1	0	""	14	16
14	5	1	0	""	15	11
15	7	1	0	""	8466	
16	5	1	0	""	2	17
17	5	1	0	""	18	20
18	5	1	0	""	19	11
19	7	1	0	""	8465	
20	1	1	0	""	1	
21	1	1	0	""	3	

4. Execution

0-Tuples are not used - all 0-tuples have address 0. Logical and character nodes follow the FLIC conventions in which they are represented as sums of products. At the start of a program execution, the node with address 1 is assumed to be the 'root-node'. Lazy evaluation is assumed, unless applicative order is (a) specified by an operator such as "STRICT" in FLIC, (b) implied by an operator such as integer plus, or (c) specified by an annotation.

5. Conclusions

A graphical representation of a functional program in the FLIC intermediate code, or a similar language, has been suggested as a standard internal form for functional programs. Full details of the internal format have not been specified because they are likely to be somewhat machine dependent and, in any event, communication of programs between different sites is not likely to take place via direct memory dumps! Instead, a precisely defined printable format for the graph has been given and this is the level at which communication is expected.

The graphical representation can be used with various different sets of basic operators. The FLIC operator set is preferred, but alternative sets can be defined if required. The representation also permits (but does not require) all nodes of the graph to be named and annotated with additional information (which must not affect the meaning of the program in the functional sense).

The aim of this standard graphical representation is to encourage the use of compatible representations in work on functional programming languages which is being carried out at many sites, rather than the proliferation of many incompatible representations which differ in arbitrary, but often quite trivial, ways.

References

1. T.H. Axford, “Lecture Notes on Functional Programming,” Internal Report CSR-86-13, Department of Computer Science, University of Birmingham, Birmingham, UK (1986).
2. T.H. Axford, “Reference Counting of Cyclic Graphs for Functional Programs,” *Computer Journal*, 33, 5, pp. 466-470 (1990).
3. D. Berry, *The Pifl Programmer’s Manual*, Department of Computer Science, University of Warwick, Coventry, UK (1981).
4. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
5. S.L. Peyton Jones and M.S. Joy, “FLIC - a Functional Language Intermediate Code,” Research Report 148, Department of Computer Science, University of Warwick, Coventry, UK (1989). Revised 1990. Previous version appeared as Internal Note 2048, Department of Computer Science, University College London (1987).
6. D.A. Turner, *SASL Language Manual*, University of Kent, Canterbury, UK (1979). Revised 1983, 1989 and 1990.

6. Appendix: FLIC-Compatible Predefined Operators

We present the operators currently supported, all of which agree with the semantics of FLIC. They are assumed to be Curried operators.

Arity	Type of Result	Description	Code (hex)	FLIC Name
1	Integer	integer unary minus	1110	INT_
2	Integer	integer plus	2110	INT+
2	Integer	integer minus	2111	INT-
2	Integer	integer multiply	2112	INT*
2	Integer	integer divide (truncation)	2113	INT/
2	Integer	integer remainder	2114	INT%
1	Real	real unary minus	1220	FLOAT_
2	Real	real plus	2220	FLOAT+
2	Real	real minus	2221	FLOAT-
2	Real	real multiply	2222	FLOAT*
2	Real	real divide	2223	FLOAT/
1	Real	square root	1221	SQRT
1	Real	sin (radians)	1222	SIN
1	Real	cos (radians)	1223	COS
1	Real	inverse tangent (range $\pm\pi/2$)	1224	ARCTAN
1	Real	exponentiation e^x	1225	EXP
1	Real	natural logarithm	1226	LN
2	Sum	integer <	2310	INT<
2	Sum	integer >	2311	INT>
2	Sum	integer <=	2312	INT<=
2	Sum	integer >=	2313	INT>=
2	Sum	integer equality	2314	INT=

2	Sum	integer !=	2315	INT!=
2	Sum	polymorphic equality	23F0	POLY=
2	Sum	polymorphic <	23F1	POLY<
2	Sum	polymorphic >	23F2	POLY>
2	Sum	real <	2320	FLOAT<
2	Sum	real >	2321	FLOAT>
2	Sum	real <=	2322	FLOAT<=
2	Sum	real >=	2323	FLOAT>=
2	Sum	real equality	2324	FLOAT=
2	Sum	real !=	2325	FLOAT!=
1	Real	integer to real conversion	1210	INT->FLOAT
1	Integer	real to integer conversion	1120	FLOAT->INT
2	(unknown)	sequential evaluation	2FF0	SEQ
2	(unknown)	applicative order evaluation	2FF1	STRICT
1	Product	input from a file	1440	INPUT
3	(unknown)	conditional if..then..else	3F30	IF
1	Sum	negation	1330	NOT
2	Sum	logical inclusive OR	2330	OR
2	Sum	logical exclusive OR	2331	XOR
2	Sum	logical AND	2332	AND
(variable)	Sum	create sum-product domain	F310	PACK-n
(variable)	(unknown)	extract elt. from sum-product	FFF0	CASE-n
2	(unknown)	extract elt. from sum-product	2F11	SEL-SUM
1	Integer	extract tag from a sum=product	1130	TAG
2	(unknown)	change tuple to curried application	25F0	UNPACK
2	(unknown)	extract element of tuple	2F10	SEL-TUPLE
(variable)	Product	create a tuple	F4F0	TUPLE-n
2	(unknown)	change tuple to curried application	25F2	UNTUPLE-n
2	(unknown)	as UNTUPLE-n, but strictly	25F3	UNTUPLE!-n
1	Apply	combinator Y	15F0	
3	Apply	combinator S	35F0	
2	(unknown)	combinator K	2FF2	
(variable)	(unknown)	FLIC operator K	FF10	K-n
1	(unknown)	combinator I	1FF0	
3	Apply	combinator B	35F1	
3	Apply	combinator C	35F2	
4	Apply	combinator S’	45F0	
4	Apply	combinator B’	45F1	
4	Apply	combinator C’	45F2	