GCODE: A Revised Standard for a Graph Representation for Functional Programs

Mike Joy

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

E-mail: msj@cs.warwick.ac.uk

Tom Axford

School of Computer Science,
The University of Birmingham,
PO Box 363,
Birmingham,
B15 2TT,
UK.

E-mail: AxfordTH@bham.ac.uk

August 1990

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 data structures employed are tailored to the intermediate functional language FLIC. The textual format is designed to be transmittable easily across networks. This document details the changes in the notation that have evolved since the previous specification [1] was published.

1. Introduction

A common standard for the graphical representation of functional programs was developed at Warwick and Birmingham in 1987 to facilitate cooperation in functional language research and the interchange of functional programs [1] .

At Birmingham, the functional language used for teaching purposes [2] has similarities with the languages SASL [3] and Pifl [4] used at Warwick. It was found possible to represent all of these in a common graphical form and to share programs using a common printable representation of these graphs. In the light of the experience gained, however, it has become apparent that some changes to the graphical representation are desirable.

This paper contains a description of our revised graphical representation for functional programs, which is called GCODE, and concentrates on the changes introduced from the 1987 specification [1] .

1.1. Reasons for Revising the Representation

Originally, we used the standard graphical representation in a number of distinct ways. Firstly, our programs for graph reduction, etc. (which were written in the programming language "C") all used exactly the same data structures for representing the functional programs that they were processing. The graphical representation of a functional program could also be printed in an ASCII text form, which was called GCODE. This form could be easily transmitted from one machine to another using email or other data transfer facilities. It was in a format that was reasonably easily readable by humans and so could be used to write programs directly in their graphical form for test purposes and also for diagnostic output to monitor the progress of graph reduction and other processing operations.

The chosen graphical representation fulfilled all these functions, but some problems arose from trying to get it to do too much. In particular, the operators appropriate to the different functional languages were not identical, so we defined the standard graphical representation to include the union of these operator sets. Each implementation then handled only a subset of the defined operators. Thus programs could be transferred from one system to another only if the operators used in one system were translated into the operators used in the other system. This was done; it is not difficult, although it introduces some inefficiency.

New directions in the continuing research at both Warwick and Birmingham have slightly different needs again, with the unattractive prospect of the proliferation of yet more subsets of the standard representation.

This problem has arisen largely from insisting on the use of common internal data declarations in all our software. In practice, however, this degree of compatibility has been of very limited value to us. We have not often exchanged software at the individual procedure level (although that was done in the very early stages of the work to get things off the ground more quickly). It has usually been much more convenient to share complete programs which accept GCODE input or produce GCODE output, and which achieve compatibility in this way.

So, our revised standard graphical representation is intended purely for the purposes of external data interchange (the 'data' is, of course, a functional program). It is not intended to be used as an internal representation (although it may often be convenient to use a very similar internal graphical representation, and that will have the added advantage of making the translation to or from GCODE particularly easy).

The revised standard is also designed primarily to be efficiently computer readable rather than human readable (although easy human readability has been a secondary aim). Human generated test input and human readable diagnostic output should ideally use a more suitable syntax or a more pictorial representation of the graph, although GCODE could be used for these purposes at a pinch.

1.2. The Revised Standard Graphical Representation

The SASL interpreter developed at Warwick uses FLIC [5] as an intermediate code. FLIC has been defined as a standard intermediate code for functional languages and is used in the implementation of several other functional languages at various places throughout the world. It is the most widely used intermediate code for functional languages and hence is an obvious choice on which to base GCODE. The graphical representation on which GCODE is based may be thought of essentially as an internal representation of FLIC.

The basic data types and structures follow FLIC. It is not possible for users 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).

All the FLIC predefined operators are supported by GCODE. Turner combinators are also included. Thus GCODE is able to represent FLIC programs both before and after translation to combinator code. Provision has been made to introduce extra operators at a future date, should the need arise.

A full list of operators is not included - anyone wishing to have such a list, or requiring a copy of the extended version [6] of this document, may obtain a copy from either of the authors.

2. The Graph

The abstract structure of a functional program in GCODE is a connected, directed, and possibly cyclic, graph. It is necessary to understand this underlying structure to understand GCODE. The semantics of this graphical representation is based on the semantics of FLIC (see [5] for a definition of FLIC).

There are nine different node types, described below.

2.1. Node Types

2.1.1. Integer

Nodes of type *integer* represent known constants of the usual type integer. The value of the constant is stored in the node, which has out-degree 0. No limits are specified for maximum integer size, although a particular machine implementation may well have such limits.

2.1.2. Real

Nodes of type *real* represent known constants of the usual type real. As for integers, the value of the constant is stored in the node, which has out-degree 0. No limits are currently specified for accuracy of real numbers.

2.1.3. Sum-Product

These nodes represent tagged Cartesian products and have variable out-degree. The first element is the size of the product (the out-degree), the second is the "tag" of the sum, and the rest are pointers to the consecutive elements of the product (which may be of any type). See [5] for further details of the semantics of such nodes.

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. 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. All functions are curried.

2.1.6. Recursive Reference

A node of type *recursive reference* is used whenever a pointer introduces a cycle into the graph (e.g. 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 there 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 subsection).

2.1.7. Operator

Nodes of type *operator* represent basic operators which are predefined as part of the language. A code for the operator is stored in the node, which has out-degree 0. Provision is made for families of operators (such as `PACK` in FLIC) and up to 2 integer qualifiers can be stored in the node. For a list of the standard operators please consult [6] .

2.1.8. 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 free variables). They have an out-degree of 0.

2.1.9. Undefined

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

2.2. Graph Structure

It has been shown [7] 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.

These conditions are not difficult to satisfy and, provided they are satisfied, reference counting of strong pointers only is all that is needed for safe memory management. See [7] for more details.

2.3. 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 is (or *could* be - it is not unique, due to possible code-sharing), as shown in figure 1 below.

The operator `case` is a generalised conditional (see [5] for a detailed definition), which is used to represent *if...then...else*.

When represented as GCODE, this may appear as shown in figure 2 below.

The full definition of GCODE is given in [6] , but briefly, each line defines one node, the first column being a number to identify the node (starting with 0 for the root), the second column is a code from the type of node, and remaining columns are more parameters to describe that particular node.

3. Changes to the 1987 Specification of GCODE

The following principal changes to GCODE from the 1987 [1] definition are:

* Type "Unique" has been abolished and the integers representing the types have been renumbered.
* Types "Sum" and "Product" have been abolished and replaced by a single type "Sum-Product".
* The "comment" symbol has been changed from * to #.
* The 'root node' is now numbered 0 (not 1).
* Operators now have small integer codes (not 4-digit hex codes).

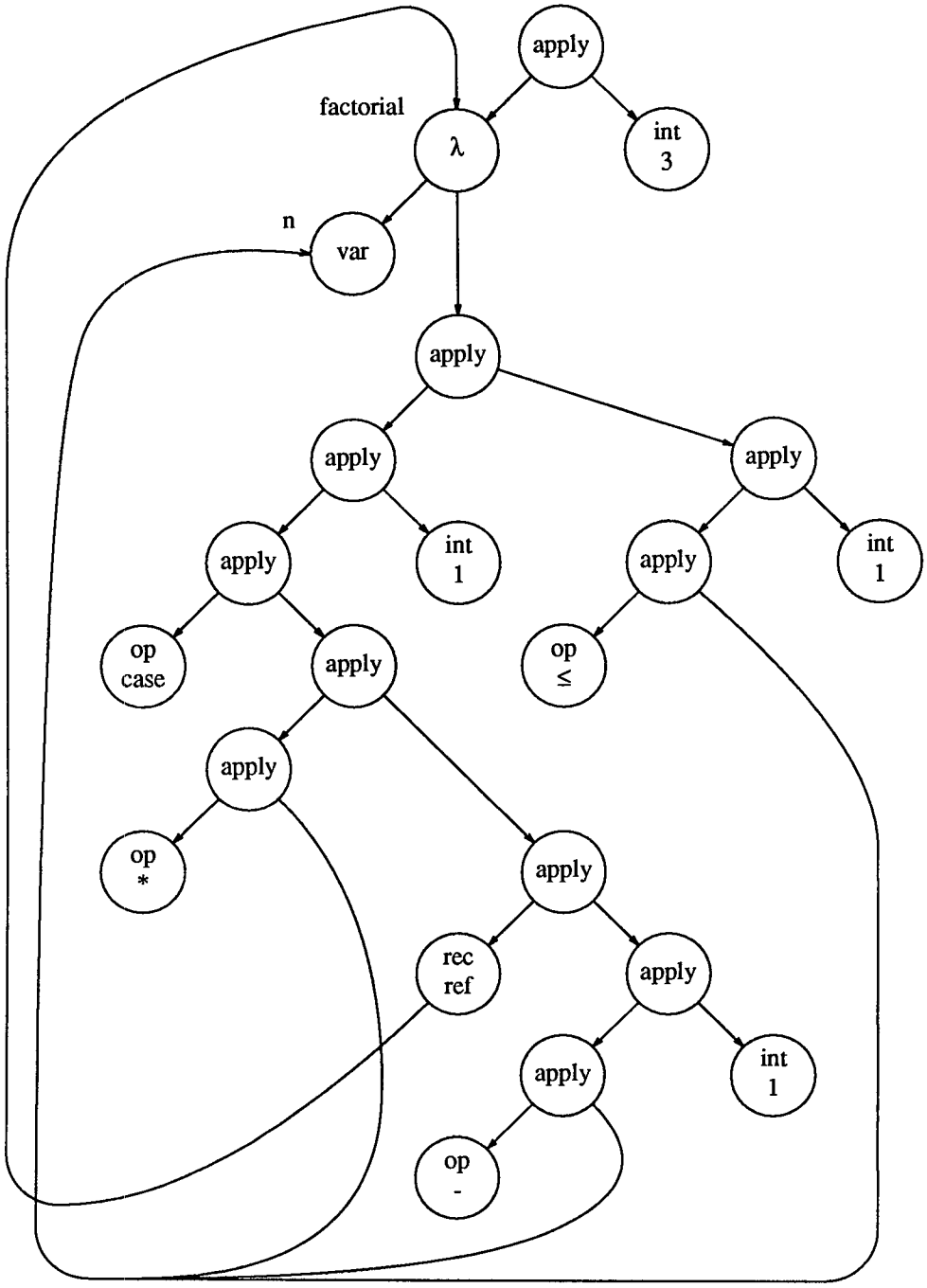

Figure 1: Graph of the program for "factorial 3"

```
#GCODE,TEXT Factorial 3 using standard recursive algorithm
   0  5   1     0 "MAIN" 1 2
   1  4   1     0 "lambda" 3 4
   2  1   1     0 "integer" 3
   3  8   4     0 "n"
   4  5   1     0 "apply" 5 11
   5  5   1     0 "apply" 7 14
   6  5   1     0 "apply" 8 9
   7  5   1     0 "apply" 10 6
   8  5   1     0 "apply" 12 3
   9  5   1     0 "apply" 15 16
  10  7   1     0 "CASE" 36 2
  11  5   1     0 "apply" 13 21
  12  7   1     0 "INT*" 4
  13  5   1     0 "apply" 20 3
  14  1   1     0 "integer" 1
  15  6   1     0 "recursive reference" 1
  16  5   1     0 "apply" 17 18
  17  5   1     0 "apply" 19 3
  18  1   1     0 "integer" 1
  19  7   1     0 "INT-" 3
  20  7   1     0 "INT<=" 17
  21  1   1     0 "integer" 1
```

Figure 2: GCODE for "factorial 3"

4. 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 printable format for the graph has been introduced and this is the level at which communication is expected.

The graphical representation is based on FLIC and supports the FLIC operator set in full. The common Turner combinators are supported as well. 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. M.S. Joy and T.H. Axford, "A Standard for a Graph Representation for Functional Programs," *ACM SIGPLAN Notices*, vol. 23, no. 1, pp. 75-82, 1988. University of Birmingham Department of Computer Science Internal Report CSR-87-1 and University of Warwick Department of Computer Science Research Report 95 (1987).
2. T.H. Axford, "Lecture Notes on Functional Programming," Internal Report CSR-86-13, Department of Computer Science, University of Birmingham, Birmingham, 1986.
3. D.A. Turner, *SASL Language Manual*, University of Kent, Canterbury, 1979. Revised 1983 and 1989.

4. D. Berry, *The Pifl Programmer's Manual,* Department of Computer Science, University of Warwick, Coventry, 1981.

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, 1989. Also University of Glasgow Department of Computing Science Internal Note (1989). Previous version appeared as Internal Note 2048, Department of Computer Science, University College London (1987).

6. M.S. Joy and T.H. Axford, "GCODE: A Revised Standard Graphical Representation for Functional Programs," Research Report 159, Department of Computer Science, University of Warwick, 1990. Also University of Birmingham School of Computer Science Research Report CSR-90-9 (1990).

7. T.H. Axford, *Reference Counting of Cyclic Graphs for Functional Programs,* Department of Computer Science, University of Birmingham, Birmingham, 1987. To appear in "Computer Journal".