The Translation of High-Level Functional Languages to FLIC

Mike Joy

Department of Computer Science
University of Warwick
Coventry
CV4 7AL
U.K.

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

May 1989

ABSTRACT

FLIC is designed as a "standardised" intermediate-level functional language. It is an interface between high-level functional languages and implementation software so that with a single FLIC implementation many high-level functional languages can be supported. In this paper we examine some aspects of the translation of high-level functional languages to FLIC with specific reference to two translators - one from FP one from SASL.

1. Introduction

FLIC [6] is an intermediate-level functional language. It contains basic data structures such as integers real numbers and sum-product domains together with a minimum syntax. Its intended use is as a "back-end" to high-level functional language implementations so that with a single FLIC implementation several functional languages can be supported with minimal extra software development. It is presumed to be much simpler to write a program to translate a functional language to FLIC than to write a compiler or interpreter for that functional language from scratch.

As part of the Functional Language Implementation Project [2] we have written translators from high-level functional languages to FLIC. This translation process is sub-optimal - the translators produce somewhat verbose FLIC code using unsophisticated algorithms. No attempt is made to optimise the code as we feel that such optimisation can and should be performed by a FLIC interpreter/compiler. It is evident that such translators will in general produce FLIC code which takes longer to evaluate than "hand-coded" FLIC. Observing the degree of and reasons for this introduced inefficiency will give us some understanding both of the usefulness of our own translators and of the possible directions available to us to optimise FLIC code automatically.

We have translators for two languages FP [13] and SASL [7] to FLIC; we shall examine the two translators in turn and then present data which will give some idea of the amount of inefficiency introduced when compared with hand-coded FLIC code.

Please note that specification of FLIC used when preparing this document is not the final specification for the language. Also the software used is under development. Therefore the statistics presented here will not be reproducible *exactly* with future releases of the software.

2. The FLIC Interpreter

The FLIC interpreter used to obtain the results below is a graph reduction machine using GCODE [4] as a low-level graph representation. The FLIC code is read in LETs and LETRECs are removed and an acyclic graph containing lambdas and Y combinators is produced. The lambdas are then abstracted out using Turner's algorithm [8] (which is optimal to within a constant factor [5]) producing a graph containing (i) combinators (ii) operators (iii) basic data (integers reals tuples sums) and (iv) functional application. This graph is then reduced using lazy evaluation the reduction rule for the Y combinator producing *acyclic* graphs.

3. The SASL to FLIC Translator

The SASL language accepted by our translator did not include Real Number data types nor ZF expressions; no attempt was made to include these structures. The translator was written as a back-end to a SASL interpreter already in existence. It proved fairly straightforward to include routines to generate FLIC code as the structure of both languages is generally similar. The main problem lay in *correctly* translating the definitions of recursively defined functions as the scope rules differ in both languages and this proved somewhat inelegant although not difficult.

3.1. "Tagging"

Since SASL includes polymorphic operators such as "=" it was necessary to "tag" objects in order to denote their type (FLIC has few polymorphic operators). For instance the integer `1` would be translated to `(PACK 1 3 1)` and the operator `SEL 1 0` used to extract the integer value in order to do arithmetic.

This clearly introduces a large overhead. For example consider the SASL program

```
5 + 6
```

which translates to the FLIC program

```
(\x \y PACK 1 3 (INT+ (SEL 1 0 x)(SEL 1 0 y)))(PACK 1 3 5)(PACK 1 3 6)
```

and after compilation to combinator code requires 13 reduction steps to evaluate.

3.2. Other Considerations

Apart from tagging the translation was fairly straightforward. SASL relies on "curried" representation of functional application as does FLIC so the majority of the extra code which needed to be written was for specific SASL functions. In particular the "prelude" - the list of predefined SASL functions - is quite lengthy roughly 6k when written in FLIC.

Since both SASL and FLIC are lazy there was no need to introduce significant extra code to control evaluation order.

3.3. Expected Performance

In practice we have found that the vast majority of FLIC programs - whether or not generated from SASL or FP - evaluate with approximately 75% of the reduction steps being combinator reduction steps. Thus by assuming that the structure of a SASL and a hand-coded FLIC program are similar and assuming a factor of 13 applied to reductions steps for "basic" operators the net performance degradation would be a factor of around

$0.75 + (0.25 * 13) = 4.$

This is indeed borne out by the statistics in section 5.1.

4. The FP to FLIC Translator

The language FP differs from most functional languages in relying on defining functions by means of "functional forms" i.e. composition of already defined functions without reference to their formal arguments. FP is *not* "sugared lambda calculus".

Two significant problems were encountered in writing routines to translate FP to FLIC. Firstly the FP interpreter we had available proved quite difficult to incorporate translation routines into and all source code files had to be altered. Secondly the translation algorithms themselves were not all obvious - even functional application required specific FLIC code to be written!

4.1. "Tagging"

FP as implemented at Warwick includes real numbers thus all arithmetic operators are polymorphic. Thus as with SASL all data must be tagged.

4.2. Strictness

The FP interpreter contains a data type "undefined" (printed as **?**); the translator introduces a specific node given by

```
= UNDEFINED (PACK 0 5)
```

and an FP program when translated to FLIC may be considered as being strongly typed (so that the FLIC interpreter will never crash when fed with such a program). This introduces overheads so that for instance / must examine its argument then check

(i) that it is a list

(ii) that it has 2 elements

(iii) that each of the two elements is *either* an integer *or* a real

(iv) that a FLIC `INT/` or `FLOAT/` should be generated as appropriate and an `INT->FLOAT` may need to be applied to one of the arguments and lastly

(v) that the second element of the list is non-zero.

The resulting FLIC is:

```
(\arg CASE 6
/* Null-list */ UNDEFINED
/* List */      (= tailarg (TAIL arg) IF (IS-NIL tailarg) UNDEFINED
                 (IF (NOT (IS-NIL (TAIL tailarg))) UNDEFINED
                  (= h (HEAD arg) = t (HEAD tailarg) = hh (TAG h)
                   = ht (TAG t) = th (SEL 1 0 h) = tt (SEL 1 0 t)
                   IF (INT= 3 hh) (IF (INT= 3 ht)
                    (IF (INT= 0 tt) UNDEFINED
                     (PACK 1 4 (FLOAT/ (INT->FLOAT th)
                                       (INT->FLOAT tt))))
                    (IF (INT= 4 ht) (IF (FLOAT= 0.0 tt) UNDEFINED
                     (PACK 1 4 (FLOAT/ (INT->FLOAT th) tt))) UNDEFINED))
                   (IF (INT= 4 hh) (IF (INT= 4 ht)
                    (IF (FLOAT= 0.0 tt) UNDEFINED
                        (PACK 1 4 (FLOAT/ th tt)))
                    (IF (INT= 3 ht) (IF (INT= 0 tt) UNDEFINED
                      (PACK 1 4 (FLOAT/ th (INT->FLOAT tt)))) UNDEFINED))
                         UNDEFINED))))
/* Boolean */   UNDEFINED
/* Integer*/    UNDEFINED
/* Real */      UNDEFINED
/* Undefined */ UNDEFINED
/* Object */    arg)
```

When the FP program

```
/ : <1.2 3>
```

is fed to the translator and evaluated 243 reduction steps are needed!

4.3. Other Considerations

Clearly we are going to be faced with a very severe performance degradation. It was considered abandoning the `UNDEFINED` data type and letting the FLIC interpreter deal with undefinedness via `ABORT`. However we felt that this would yield a very messy user interface and would produse screen output which would differ from that produced by the original FP interpreter. Even had we taken that course of action the overhead would still have been very high.

4.4. Expected Performance

Assuming as in the previous section that 25% of the reduction steps relate to basic functions we get a performance degradation of around

$0.75 + (0.25 * 243) = 61.5$.

This is indeed borne out by the statistics in section 5.1 below; the actual figures involved are generally somewhat higher as extra code is required for functional composition for formatting the output and for checking strictness in lists. The cases where performance degradation is much smaller - such as bubblesort - involve programs where nearly all the basic operations are list manipulations.

5. Statistics

The following statistics display the relative efficiencies of the languages. The columns relate to:

FLIC: hand-coded (and optimised if possible) FLIC code

SASL: hand-coded SASL passed to the SASL interpreter and automatically translated to FLIC and

FP: hand-coded FP passed to the FP interpreter and automatically translated to FLIC.

Actual times for evaluation are not presented here; however when run on a Sun 4 the GCODE graph reducer will perform approximately 20k reductions per second.

5.1. FLIC Reduction Steps

In the following statistics the percentage of reduction steps which are combinators (as opposed to other delta reductions) is 75.5% ± 2.3% (with one exception which was 69.3%).

For FLIC the actual number of reduction steps (that is basic graph rewrites one for each basic operation performed) is given. In the other two columns the figures relate to the number of FLIC reduction steps performed when the program written in SASL or FP is translated automatically to FLIC and then evaluated by the FLIC interpreter. The number presented is the *ratio* of the number of steps to the number taken by the hand-coded FLIC program in column 2.

	FLIC (reduction steps)	SASL (ratio)	FP (ratio)
factorial	142	4.12	67.49
ackermann	44485	2.95	75.96
bubblesort	54879	2.10	7.87
quicksort	20993	2.12	24.86
primes	2205	3.76	93.95
edigits	40860	2.60	7.04

5.2. Graph Size

These statistics relate to the size of FLIC graph when initially read in (before lambda abstraction). The FLIC column gives the actual graph size the other two columns are ratios as in the previous section.

	FLIC (actual graph size)	SASL (ratio)	FP (ratio)
factorial	29	4.79	10.13
ackermann	61	5.03	11.06
bubblesort	781	9.87	10.02
quicksort	483	16.35	23.26
primes	217	42.72	47.76
edigits	475	4.28	51.34

5.3. "Basic Operations"

In this section we compare the "raw" interpreters (the SASL and FP code is *not* translated to FLIC before evaluation). We use as metrics the most basic concept of "one evaluation step" available to us. This is of course dependent on the particular interpreters we are using but is useful in that they show that the hand-coded FLIC is as efficient as the SASL and FP interpreters to within an order of magnitude when using this metric and that since the "basic operations" of the FLIC interpreter (namely combinator reductions) are very "small" operations that our FLIC interpreter is not inefficient.

These statistics have in the FLIC column the number of graph reduction steps (calls to the procedure *doonestep()*) in the SASL column the number of calls to the procedure *eval()* and in the FP column the number of calls to the *execute()* procedure. In the following two columns these figures are expressed as a ratio to the figure in the FLIC column. For further details of the precise actions of these procedures the interested reader is referred to the source code for the software.

The basic operations for the SASL and FP interpreters are more complicated than for the FLIC interpreter and it is thus not surprising that the majority of the figures in the latter two columns are below 1.0.

	FLIC (graph reductions)	SASL (eval())	FP (execute())	SASL (ratio)	FP (ratio)
factorial	142	108	161	0.76	1.13
ackermann	44485	21832	67010	0.49	1.51
bubblesort	54879	7062	5236	0.13	0.10
quicksort	20993	5770	9090	0.27	0.43
primes	2205	1760	1362	0.80	0.62
edigits	40860	12030	5122	0.29	0.13

6. Conclusions

We should emphasise that the results contained in this paper are *not* intended to be a definitive benchmarking of our software. They are preliminary results designed to highlight the *inefficiencies* of the SASL and FP to FLIC translation algorithms and to attempt to put them in perspective.

SASL and FLIC are similar languages in structure and the translation between the two appears to be somewhat as expected. The translator is proving to be a very useful tool.

In section 5.1 however we see that FP produces a very severe performance degradation when translated to FLIC.

It is hoped in future work to investigate algorithms to optimise FLIC and GCODE and to reduce the inefficiencies introduced by these translators.

References

1. J. Backus "Function Level Programs as Mathematical Objects " *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture Portsmouth NH* pp. 1-10 ACM New York (1981).

2. M.S. Joy *Description of the FLIP Software* Department of Computer Science University of Warwick Coventry UK (1987). Revised 1990.

3. M.S. Joy *FP Language Manual* University of Warwick Coventry UK (1988). Revised 1990.
4. M.S. Joy and T.H. Axford "A Standard for a Graph Representation for Functional Programs" *ACM SIGPLAN Notices* 23 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).
5. M.S. Joy V.J. Rayward-Smith and F.W. Burton "Efficient Combinator Code" *Computer Languages* 10 3/4 pp. 211-224 (1985).
6. 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).
7. D.A. Turner *SASL Language Manual* University of Kent Canterbury UK (1979). Revised 1983 1989 and 1990.
8. D.A. Turner "Another Algorithm for Bracket Abstraction" *Journal of Symbolic Logic* 44 3 pp. 67-70 (1979).

7. Appendix: Source Code

7.1. The Factorial Function

The standard recursive algorithm is used with argument 10.

```
||SASL Factorial using standard recursive algorithm

DEF
factorial n = n > 1 -> n * factorial (n-1)
              1
?
factorial 10 ?

#FP Factorial using standard recursive algorithm

{factorial (<= @ [id  %1] -> %1;
            * @ [id  factorial @ - @ [id  %1] ])}
factorial : 10

/*FLIC Factorial using standard recursive algorithm */

& (factorial) ( (\n IF (INT>= 1 n) 1 (INT* n (factorial (INT- n 1)))))
factorial 10
```

7.2. Ackermann's Function

```
||SASL Ackermann's Function using deep recursion

DEF
ackermann 0 n   = n+1
ackermann m 0   = ackermann (m-1) 1
ackermann m n   = ackermann (m-1) (ackermann m (n-1))
?
ackermann 3 3 ?

#FP Ackermann's Function using deep recursion

{sub1 - @ [id  %1]}
{eq0 eq @ [id  %0]}
{ack (eq0 @ 1 -> + @ [%1  2];
     (eq0 @ 2 -> ack @ [sub1 @ 1  %1];
      ack @ [sub1 @ 1  ack @ [1  sub1 @ 2]]))}
ack : <3 3>

/*FLIC Ackermann's function using deep recursion */

& (ackermann) ((\m \n IF (INT= 0 m) (INT+ 1 n)
                      (IF (INT= 0 n) (ackermann (INT- m 1) 1)
                      (ackermann (INT- m 1) (ackermann m (INT- n 1))))))
ackermann 3 3
```

7.3. The Bubblesort Function (and test data)

The test data is a fairly random (but badly ordered!) sequence.

```
||SASL Bubblesort

DEF
bubblesort x =  x = () -> ()
                bubblesort (allbutlast y) ++ ( (lastof y) : ())
                WHERE
                y = bubble x
                bubble () = ()
                bubble (a : ()) = a : ()
                bubble (a : b : c) = a > b -> b : bubble(a : c)
                                     a : bubble (b : c)
                lastof (a:()) = a
                lastof (a:b) = lastof(b)
                allbutlast (a:()) = ()
                allbutlast (a:b) = a : allbutlast b
testdata = (1 2 3 4 5 20 19 18 17 16 6 7 8 9 10 15 14 13 12 11)
?
bubblesort testdata ?

#FP Bubblesort

{swap
    concat@[ [2 1] tl@tl ]}
{step
    (>@[1 2] -> swap ; id)}
{pass
    (<@[length %2] -> id; apndl@[1 pass@tl]@step)}
{bubblesort
    (<@[length %2] -> id; apndr@[bubblesort@tlr last]@pass)}

bubblesort : <1 2 3 4 5 20 19 18 17 16 6 7 8 9 10 15 14 13 12 11>

/*FLIC Bubblesort */
& (append) ((\l \m IF (IS-NIL l) m (CONS (HEAD l) (append (TAIL l) m))))
& (bubble) ((\l IF (IS-NIL l) NIL (IF (IS-NIL (TAIL l))l
        (IF (INT> (HEAD l) (HEAD (TAIL l))) (CONS (HEAD (TAIL l))
             (bubble (CONS (HEAD l) (TAIL (TAIL l)))))
               (CONS (HEAD l) (bubble (TAIL l)))))))
& (lastof) ((\l IF (IS-NIL (TAIL l)) (HEAD l) (lastof (TAIL l))))
& (allbutlast) ((\l IF (IS-NIL (TAIL l)) NIL (CONS (HEAD l)
                          (allbutlast (TAIL l)))))
& (bubblesort) ((\x IF (IS-NIL x) NIL
            ( = y (bubble x) append (bubblesort (allbutlast y))
            (CONS (lastof y) NIL))))

= testdata
(CONS 1 (CONS 2 (CONS 3 (CONS 4 (CONS 5 (CONS 20 (CONS 19 (CONS 18
(CONS 17 (CONS 16 (CONS 6 (CONS 7 (CONS 8 (CONS 9 (CONS 10 (CONS 15
(CONS 14 (CONS 13 (CONS 12 (CONS 11 NIL))))))))))))))))))))
bubblesort testdata
```

7.4. Quicksort

```
||SASL Quicksort

DEF
quicksort x =    x = () -> ()
        quicksort (below pivot) ++ (pivot ) ++ quicksort (above pivot)
        WHERE
        hd (a:b) = a
        tl (a:b) = b
        pivot = hd x
        above n = above1 (tl x)
                WHERE
                above1 () = ()
                above1 (a:b) = a > n -> a : above1 b
                        above1 b
        below n = below1 (tl x)
                WHERE
                below1 () = ()
                below1 (a:b) = a <= n -> a : below1 b
                        below1 b
testdata = (1 2 3 4 5 20 19 18 17 16 6 7 8 9 10 15 14 13 12 11)
?
quicksort testdata ?

#FP Quicksort

{findgt
    (null @ 1 -> %<>;
          (<= @ [hd @ 1  2] -> apndl @ [hd @ 1  findgt @ [tl @ 1  2]];
         findgt @ [tl @ 1  2]))}
{findle
    (null @ 1 -> %<>;
          (> @ [hd @ 1  2] -> apndl @ [hd @ 1  findle @ [tl @ 1  2]];
         findle @ [tl @ 1  2]))}
{quicksort
    (null -> %<>;
          (null @ tl -> id;
            concat @ [ quicksort @ findgt @ [tl  hd]
            [hd]
            quicksort @ findle @ [tl  hd]]))}
quicksort : <1 2 3 4 5 20 19 18 17 16 6 7 8 9 10 15 14 13 12 11>
```

```
/*FLIC Quicksort */

& (append) ( (\l \m IF (IS-NIL l) m (CONS (HEAD l)
                       (append (TAIL l) m))))
& (quicksort) ( (\l
    = pivot (HEAD l)
    = above (\n & (above1)
            ( (\l IF (IS-NIL l) NIL
                (IF (INT> (HEAD l) n) (CONS (HEAD l)
                (above1 (TAIL l))) (above1 (TAIL l)))))
                 above1 (TAIL l))

    = below (\n & (below1)
            ( (\l IF (IS-NIL l) NIL
                (IF (INT<= (HEAD l) n) (CONS (HEAD l)
                    (below1 (TAIL l))) (below1 (TAIL l)))))
                 below1 (TAIL l))

    IF (IS-NIL l) NIL
        (append (quicksort (below pivot))
                (CONS pivot (quicksort (above pivot))))))
= testdata
(CONS 1 (CONS 2 (CONS 3 (CONS 4 (CONS 5 (CONS 20 (CONS 19 (CONS 18
(CONS 17 (CONS 16 (CONS 6 (CONS 7 (CONS 8 (CONS 9 (CONS 10 (CONS 15
(CONS 14 (CONS 13 (CONS 12 (CONS 11 NIL))))))))))))))))))))

quicksort testdata
```

7.5. Prime Numbers

The algorithm used is test division; the first ten are generated.

```
||SASL Primes using test division

DEF
primes =  firstn 10 (findvalues isprime ints)
findvalues f (a:b) = f a -> a : findvalues f b
                     findvalues f b
ints = ints1 2
        WHERE
        ints1 n = n : ints1 (n+1)
ptest x y = (x*x > y) | ( (y REM x ~= 0) & ptest (x+1) y )
isprime y = ptest 2 y
firstn n (a:b) = n <= 0 -> ()
                 a : (firstn (n-1) b)
?
primes ?
```

The FP algorithm is slightly different as FP is not lazy!

```
#FP Primes using test division
# Print prime numbers from 3 to ?
#
{factors
    &(+@[id %1]@*@[id %2])@iota@div@[id %4]
}
{isprime
    |and@&(~=@[id %0])@&mod@distl@[id factors]
}
{primes
    concat@&(isprime -> [id] ; %<>)@&(+@[id %1]@*@[id %2])@iota
}
primes : 14
```

The FLIC source is lengthy and is omitted here.

7.6. Digits of the Decimal Representation of 'E'

The first 5 digits of 'E' are produced. For FLIC and SASL infinite lists are used but for FP the algorithm has been modified.

```
||SASL 'E' in a decimal representation

DEF
norm c (h1 : h2 : t2) = (h2 + 9) < c -> h1 : norm (c + 1) (h2 : t2)
            carry c (h1 : norm (c + 1) (h2 : t2))
            WHERE
            carry c (h1:h2:t2) = (h1 + (h2 / c)):(h2 REM c):t2
convert (h : t) = h : convert (norm 2 (0 : (mult t)))
          WHERE
          mult (h : t) = (10 * h) : mult t
edigits n = n > 0 -> find inflist n
        inflist
        WHERE
        inflist = convert (2 : seq 1)
        find (a:b) n = n <= 0 -> ()
               a : find b (n-1)

        seq n = n : seq n
?
edigits 5 ?
```

```
#FP 'E' in a decimal representation

{carry
    (= @ [tl @ 2  %<>] -> [hd @ 2];
       apndl @ [+ @ [hd @ 2  div @ [hd @ tl @ 2  1]]
       apndl @ [mod @ [hd @ tl @ 2  1]  tl @ tl @ 2]])}
{norm
    (= @ [tl @ 2  %<>] -> [hd @ 2];
       (= @ [tl @ tl @ 2  %<>] -> %<>;
          (< @ [ + @ [hd @ tl @ 2  %9]  1] -> apndl @ [hd @ 2
         norm @ [ + @ [1  %1]  tl @ 2]];
         carry @ [1  apndl @ [hd @ 2
         norm @ [ + @ [1  %1]  tl @ 2]]]))))}
{mult
    (null @ id -> %<>;
          apndl @ [* @ [hd  %10]  mult @ tl])}
{convert
    (null @ id -> %<>;
     apndl @ [hd  convert @ norm @ [%2  apndl @ [%0  mult @ tl]]])}
{inflistof
    (= @ [id  %0] -> %<>;
       apndl @ [%1  apndl @ [%1  inflistof @ - @ [id  %1]]])}
{firstn
    (= @ [2  %0] -> %<>;
       (null @ 1 -> %<>;
         apndl @ [hd @ 1  firstn @ [tl @ 1  - @ [2  %1]]]))}
{edigits
    firstn @ [convert @ concat @ [%<2 1 1 1>  inflistof @ id]  id]}
edigits : 5
```

```
/*FLIC 'E' in a decimal representation */

& (carry) ((\c \lis
       = h1 (HEAD lis) = t1 (TAIL lis) = h2 (HEAD t1) = t2 (TAIL t1)
           CONS (INT+ h1 (INT/ h2 c)) (CONS (INT% h2 c) t2)))

& (mult) ((\lis CONS (INT* 10 (HEAD lis)) (mult (TAIL lis))))

& (norm) ((\c \lis
           = h1 (HEAD lis) = t1 (TAIL lis) = h2 (HEAD t1) = t2 (TAIL t1)
           IF (INT< (INT+ 9 h2) c)
             (CONS h1 (norm (INT+ 1 c) t1))
             (carry c (CONS h1 (norm (INT+ 1 c) t1)))))

& (convert) ((\lis
       CONS (HEAD lis) (convert (norm 2 (CONS 0 (mult (TAIL lis)))))))

& (seq) ((\n CONS n (seq n)))

& (find) ((\l \n IF (INT= n 0) NIL (CONS (HEAD l)
                        (find (TAIL l) (INT- n 1)))))

= edigits ((\list \n IF (INT>= 0 n) list (find list n))
        (convert (CONS 2 (seq 1))))

edigits 5
```