WILLIAMS COLLEGE WILLIAMSTOWN, MASSACHUSETTS 01267

DEPARTMENT OF MATHEMATICAL SCIENCES
Bronfman Science Center
(413) 597-2438

20 April 1982

John H. Williams
IBM Research Laboratory
5600 Cottle Road
San Jose, California 95193

Dear John:

When you were giving a series of lectures at M.I.T. last year, John Guttag remarked that you had written an interpreter for FP in LISP. Some of my students and I are interested in studying FP, especially with regard to specifying its semantics and developing a method of mechanically verifying programs written in FP. I would be very grateful if you could send me a listing or perhaps a tape of your interpreter. Our LISP was developed by the University of Wisconsin for use on the Univac 1170 so there probably are differences between your LISP and ours. If it is possible to get a listing or tape of your interpreter please let me know what I need to do and any charge there might be.

I have copies of Backus' original paper on FP and two of your papers: "Formal Representations for Recursively Defined Functional Programs" and "On the development of the algebra of functional programs." If you have other papers or documentation for the language, I would be very grateful to receive them. Thanks for your help.

Sincerely,

Kim B. Bruce
Assistant Professor

KB:ejs

International Business Machines Corporation

5600 Cottle Road
San Jose, California 95193

K51/61F

408/256-7593

April 29, 1982

Professor Kim B. Bruce
Department of Mathematical Sciences
Williams College
Williamstown, MA 01267

Dear Kim,

Thanks for your letter inquiring about FP. The "interpreter" that was mentioned in Guttag's class was not written "in" Lisp but rather "on" Lisp; that is, I defined a collection of Lisp functions to act as the FP primitives and combining forms. Having made those definitions, one can view the Lisp interpreter as a sort of FFP interpreter. While this is probably not what you had in mind, I will enclose the definitions anyway; perhaps you will find them interesting. The Lisp I used is one that was developed at our Yorktown lab. It has much the same style and flavor of Norman's U.W. Lisp, so you can probably figure out what the various built-in functions do.

I have also enclosed some sample program definitions that I tried on the resulting interpreter. I tried three different versions of function definitions:

a) a straightforward "define" approach in which program names get bound to s-expressions. Thus, everytime a defined FP function gets applied, we must first "construct" it by applying the combining forms that produce it. For example, if def f = g ∘ h, then everytime we evaluate something like f:x, the system must first apply comp to g and h.

b) a version in which the application of the combining forms happens only once -- when the function name is defined.

c) a compiled version of b)

As I recall there was a noticable improvement in b) over a), but c) didn't add much (not surprisingly, I guess, since the Lisp compiler can't do much with a "function object" that contains lots of free function variables).

Another thing I considered doing, but never got around to, was to change the Lisp reader to accept an infix version of these expressions. For example, if the reader/parser would accept

$$/+ \circ \alpha \overline{1}$$

and produce the s-expr

(comp (insert +) (alpha (const 1))) ,

the whole thing would look more like an actual FP interpreter. Some of your students might be interested in doing this there.

Of course, the whole point of going to FP rather than using a restricted subset of Lisp (i.e. adopting an FP - programming methodology within Lisp) is to have a language processor that is designed specifically for the given (few) combining forms of FP and that doesn't have to provide for the full generality of lambda abstraction. (We have taken to calling lambda abstraction the "go to" of applicative programming languages.) The FP implementation we are using here is currently a huge thing, written in PL/I and running under VM/CMS. It was written to provide a vehicle for trying out different storage schemes for FP objects and consequently is highly modular (and highly inefficient).

I would be delighted to see an FP oriented mechanical verification system. The rich collection of identities (the algebra of John's CACM paper) should make such a system very powerful and should suggest many interesting investigations. The work that Guttag, Horning, and I have been doing on adding data types to FP is now at a stage where a mechanical algebraic checker and simplifier would be very useful as we proceed to try example programs to fine-tune our type system. (I've enclosed a copy of a paper that describes that work.) I have also enclosed a copy of some lecture notes I used last summer at Newcastle. They are essentially abridged versions of sections of the two papers you have, but there is some new material in them that you might find interesting.

I should like to be put on your mailing list for reports describing your work on FP there at Williams College.

Sincerely,

John H. Williams

JHW/smb

```
(COMP370 "(
 (TAIL (LAMBDA(X)(CDR X)))
 (S1   (LAMBDA(X)(CAR X)))
 (S2   (LAMBDA(X)(CADR X)))
 (S3   (LAMBDA(X)(CADDR X)))
 (SR1  (LAMBDA(X)(LAST  X)))
 (SR2  (LAMBDA(X)(CADR(REVERSE X))))
 (SR3  (LAMBDA(X)(CADDR (REVERSE X))))
 (TL   (LAMBDA(X)(TAIL X)))
 (TLR  (LAMBDA(X)(REVERSE (CDR (REVERSE X)))))
 (IOTA (LAMBDA(N)(IOTA1 N NIL)))
 (IOTA1(LAMBDA(N L)(PROG () LBL(COND((EQUAL N 0) (RETURN L))
        (T (SETQ L (CONS N L))))
       (SETQ N (SUB1 N))
       (GO LBL))))
 (ROTL (LAMBDA(X)(COND ((EQ X NIL)NIL)(T(APPEND (CDR X)(LIST (CAR X)))))))
 (ROTR (LAMBDA(X)(REVERSE (ROTL (REVERSE X)))))
 (DISTL (LAMBDA(X)(MAPCAR (CADR X)"(LAMBDA(Y)(LIST (CAR X) Y)))))
 (DISTR (LAMBDA(X)(MAPCAR (CAR X)"(LAMBDA(Y)(LIST Y (CADR X) )))))
 (APNDL(LAMBDA(X)(CONS (CAR X) (CADR X))))
 (APNDR(LAMBDA(X)(APPEND (CAR X) (CDR X))))
 (UN   (LAMBDA(X)(APPEND (CAR X) (CADR X))))
 (APPLY(LAMBDA(X)((CAR X) (CADR X))))
 (NULL (LAMBDA(X)(COND((EQ X NIL)"T)(T "F"))))
 (ID   (LAMBDA(X)X))
 (=    (LAMBDA(X)(COND((EQUAL (CAR X) (CADR X))"T)(T "F"))))
 (EQ0  (LAMBDA(X)(COND((EQUAL X "0)"T)(T "F"))))
 (LE1  (LAMBDA(X)(COND((GREATERP 2 X)"T)(T "F"))))
 (GT   (LAMBDA(X)(COND((GREATERP (CAR X) (CADR X))"T)(T "F"))))
 (GE   (LAMBDA(X)(COND((EQUAL (CAR X)(CADR X)) "T)
            ((GREATERP (CAR X) (CADR X)) "T) (T "F"))))
 (EVEN (LAMBDA(X)(COND ((EQUAL 0 (REMAINDER X 2)) "T)(T "F"))))
 (DIV2 (LAMBDA(X)(CAR (DIVIDE X 2))))
 (+    (LAMBDA(X)(LPLUS (CAR X) (CADR X))))
 (-    (LAMBDA(X)(DIFFERENCE (CAR X) (CADR X))))
 (*    (LAMBDA(X)(TIMES (CAR X) (CADR X))))
 (DIV  (LAMBDA(X)(DIVIDE (CAR X) (CADR X))))
 (A    (LAMBDA(X)(PLUS X 1)))
 (S    (LAMBDA(X)(SUB1 X )))
 (TRANS (LAMBDA(Y)(EVA1(APPEND "(MMAPCAR LIST)
                  (MAPCAR Y "(LAMBDA(X)(LIST "QUOTE X )))))))
 (CONST(LAMBDA(X)(FUNCTION (LAMBDA (Y)X))))
 (COMP (LAMBDA L (FUNCTION (LAMBDA(Z) (COMPAUX L Z)))))
 (COMPAUX (LAMBDA (FNS ARG)(COND ((EQ FNS NIL) ARG)
                  (T((CAR FNS) (COMPAUX (CDR FNS) ARG))))))
 (FCONS (LAMBDA L (FUNCTION (LAMBDA(Z) (CONSAUX L Z)))))
 (CONSAUX (LAMBDA (FNS ARG) (COND ((EQ FNS NIL) NIL)
                  (T (CONS((CAR FNS) ARG)(CONSAUX (CDR FNS) ARG))))))
 (FCOND (LAMBDA(X Y Z)(FUNCTION (LAMBDA(W) (COND
     ((EQ "T (X W)) (Y W)) (T (Z W)))))))
 (ALPHA(LAMBDA(Y)(FUNCTION (LAMBDA(X) (MAPCAR X Y)))))
 (/    INSERT)
 (INSERT(LAMBDA(X)(FUNCTION(LAMBDA(L)(INSL X (CAR L) (CDR L))))))
 (INSL  (LAMBDA(G A L)(PROG () LBL(COND ((EQ L NIL) (RETURN A))
                  (T (SETQ A (G (LIST A (CAR L))))))
                 (SETQ L (CDR L))
                 (GO LBL))))
 (/L   INSERT)
 (/R   INSERTR)
 (INSERTR(LAMBDA(Y)(FUNCTION(LAMBDA(X)(INS Y X)))))
 (INS  (LAMBDA (G Z)(COND ((EQ (CDR Z) NIL)(CAR Z))
```

this realization of conditional is a little more defined that it should be in strict FP

(over)

965 - 19 M - 03

```
          (T(G (LIST (CAR Z) (INS G (CDR Z))))))))
 (TIME (LAMBDA NIL ((LAMBDA(X)(DIFFERENCE (SETQ OLDTIME
              (TEMPUS-FUGIT)) X)) OLDTIME)))
))
(SETQ
 OLDTIME 0)
(FIN)
```

```
(DEFINE "(
 (COP1 (COMP (ALPHA S1) DISTL (FCONS S1 (COMP IOTA S2))))
 (CART2 (COMP (INSERT UN) (ALPHA DISTL) DISTR))
 (LENG (COMP (/ +) (ALPHA (CONST 1))))
 (POWER (COMP (ALPHA (/ UN)) CART (ALPHA (FCONS (FCONS ID)
                    (CONST NIL)))))
 (CART (COMP (/ (COMP (ALPHA UN) CART2)) (ALPHA (ALPHA (FCONS ID)))))
))
(FIN)
```

compare — def leng = / + ∘ α1

Version 1

here, the function names are bound to S-exprs.

```
(DEFINE (LIST
 (LIST "COP1 (COMP (ALPHA S1) DISTL (FCONS S1 (COMP IOTA S2))))   replicate arg1, arg2 times.
 (LIST "CART2 (COMP (INSERT UN) (ALPHA DISTL) DISTR))        Cartesian product
 (LIST "LENG (COMP (/ +) (ALPHA (CONST 1))))                 Length of a sequence
 (LIST "POWER (COMP (ALPHA (/ UN)) CART (ALPHA (FCONS (FCONS ID)   Power set of a set.
                  (CONST NIL)))))
 (LIST "CART (COMP (/ (COMP (ALPHA UN) CART2)) (ALPHA (ALPHA (FCONS ID))))) generalized
 (LIST "MATMPY (COMP (ALPHA (ALPHA IP)) (ALPHA DISTL) DISTR        Cartesian product
               (FCONS S1 (COMP TRANS S2))))
 (LIST "IP (COMP (/ +) (ALPHA * ) TRANS))                    Matmpy of Backus' paper.
))
(FIN)
```

version 2

here, the function names are bound to "function objects", ie the results of applying the combining forms (or program forming operations) to the (primitive) functions.

```
(COMP370 (LIST
 (LIST "COP1 (COMP (ALPHA S1) DISTL (FCONS S1 (COMP IOTA S2))))
 (LIST "CART2 (COMP (INSERT UN) (ALPHA DISTL) DISTR))
 (LIST "LENG (COMP (/ +) (ALPHA (CONST 1))))
 (LIST "POWER (COMP (ALPHA (/ UN)) CART (ALPHA (FCONS (FCONS ID)
                    (CONST NIL)))))
 (LIST "CART (COMP (/ (COMP (ALPHA UN) CART2)) (ALPHA (ALPHA (FCONS ID)))))
 (LIST "MATMPY (COMP (ALPHA (ALPHA IP)) (ALPHA DISTL) DISTR
               (FCONS S1 (COMP TRANS S2))))
 (LIST "IP (COMP (/ +) (ALPHA * ) TRANS))
))
(FIN)
```

Version 3

Here, the objects of version 2 are "compiled" before binding the function names to them