Converting Call-by-reference to Call-by-value: Fortran and Lisp Coexisting

Richard J. Fateman
Computer Science Division
EECS Department
University of California
Berkeley, CA

Raymond Toy
Ericsson Mobile Platforms
Research Triangle Park, NC

January 13, 2003

Abstract

Programs written in languages supporting call-by-reference continue to be of interest in functional programming circumstances where call-by-value is standard. In particular, if we can find a neat interface, by converting Fortran to Lisp we allow programmers to take advantage of an interactive functional symbolic system while running legacy numerical code. We show this can be done without unacceptable loss of efficiency. In building a combined symbolic-numeric environment such a conversion and combination may increase the synergy between the two approaches to scientific computing. This paper builds upon earlier reported work on `f2cl`.

1 Introduction

This paper primarily addresses concerns of people who wish to use the Fortran and Lisp languages in some synergistic way. For many users of computer algebra systems (CAS) there is a perception that the built-in numeric facilities are in some way inferior to those that can be found outside the CAS by separate numerical libraries in Fortran or C, or in numeric packages like Matlab. If there were in effect a Fortran compiler that compiled Fortran source code into Lisp with acceptably efficient run-time performance, problems involved with load libraries, reading/writing problems, and error treatment would be alleviated. In particular we see the following issues being addressed:

1. "Already working" numeric programs available in source form can be adapted wholesale for use in a CAS, without worrying about "foreign function" linkages which are non-standard. Once translated into Common Lisp, porting to different Lisps should be simpler.

2. Subroutine libraries can be linked to an interactive read-eval-print loop in Lisp, replacing the compiled "main" program otherwise needed. The interaction allows for better debugging, testing, error control, and can allow immediate re-start of a program.

3. (Somewhat speculatively) numeric programs can perhaps be adapted to new domains which might be partly symbolic, or using non-conventional types such as exact integers, exact rationals, intervals, etc.

Although almost every Common Lisp has a foreign function interface to allow the dynamic loading and linking of libraries not written in Lisp, each system has differences in the details. In our experience this tends to be somewhat delicate, depending as it must on parts of the Lisp language and the underlying operating system (loader) that are not standardized in ANSI Common Lisp. A partial solution to this is UFFI, a universal FFI design by Kevin Rosenberg [6] that defines a common supported subset. The foreign-function interface typically does not address directly two parts of the Fortran calling convention: array displacements and (in effect) "output parameters" through the call-by-reference mechanism. One can argue that it is unfair to rely on data models in Fortran that in fact violate basic premises of Lisp data, but if it is the Fortran program specification that is set in concrete, then Lisp must yield.

2 Differences in the Calling Model

Consider a Fortran subroutine `F(D)` which treats its parameter or "dummy argument" `D` as an array (accessing it say from `D(1)` to `D(10)`). `F` can be called with an actual argument `A(11)`. This means that `A(11)` to `A(20)` correspond to `D(1)` to `D(10)` inside `F`. It means that the address of `A(11)` must be passed to `F` and bound to the name `D`. It means that changes to `D(5)` must be changes to `A(15)`.

Why can't one produce a pointer to the appropriate memory location in Lisp, and pass it to Fortran? For a starter, if we use as a parameter `A(11)` in Lisp[1], we extract a value from the array, and pass that to the subroutine by value. We do not pass the address of the array. Furthermore, if we somehow obtain an address where (a pointer to the value of) `A(11)` is stored, that is, the location of the 11th indexed item in a Lisp array data structure `a`, then *that location is not fixed for all time.* Indeed, many Lisp systems routinely move arrays in storage during a "garbage collection". Thus, unless one can disable garbage collection for the appropriate duration[2], or fix some arrays in memory[3], then referring to sub-arrays by location is hazardous. Of course if all further work on this array will be done within Fortran (likely), Fortran itself cannot trigger a garbage collection, and so one might be safe in assuming the array will stay put[4].

There is a notation in Common Lisp for the underlying concept of a piece of an array: an array can be "allocated" as a displaced array somewhere within a previously allocated array. This provides the facility without asserting a particular storage model; in fact there can be a different array header, if that is what the Lisp implementation requires. There is an efficiency problem in any use of a header — one may not be able to refer to a raw array entry quite so efficiently as Fortran: since at least some effort must be made to relate the array header to the associated storage.

Following our particular example, the proper Lisp-language way to pass this address is via a displaced array. That is, if `A` were originally allocated as (say) `(setf A (make-array 100 :element-type 'single-float :initial-element 0.0))` then to pass the location of the 11th (since Lisp arrays, by default, begin with 0, this one is indexed by `(aref A 10)`) we do this: `(setf A10 (make-array 90 :element-type 'single-float :displaced-to A :displaced-index-offset 10))` This explicitly identifies `(aref A10 0)` with `(aref A 10)` for all storage and access purposes.

Another technique available to simulate the Fortran environment including its memory model and its calling sequences in the Lisp environment. This is likely to produce rather opaque programs. For example all of Fortran memory may be modeled by one array, and references to all values, including those in arrays would be done by array computations. All subroutine calls would, in effect, merely pass integer indexes into (say) integer, character, single-precision, double-precision, (and complex) memories. These indexes would denote the locations of the values associated with variables, as well as locations denoting the beginning of arrays or some location within them. Some tricks that are possible in Fortran, such as `EQUIVALENCE` overlaying integers and floats, and using integer operations on float formats, will not work the same way; in particular, arrays of fixnums may be encoded differently from arrays of Fortran integers. (A typical Lisp trick is to shift

[1]actually `(aref a 11)` is the syntax.

[2]Fortunately, most Lisp implementations provide this.

[3]Sometimes this too is provided.

[4]This assumes there is only one process thread in the Lisp, namely the one running Fortran, and the programmer has refrained from using a "call-back" facility where Fortran can call Lisp. In some systems there may be other threads that could trigger the garbage collection.

all small integers left by 2 bits; the trailing "00" is a tag to mark the data as immediate number objects rather than pointers.)

If one has all the Fortran source code of interest available for recompilation, then all of these techniques can be subsumed in the compiler and run-time model. In fact, the Lisp machine architectures (e.g. Symbolics Inc.) provided a Fortran 77 compiler which used a Lisp execution model back-end. It implemented all of Fortran 77 but with some extensions that may seem odd to traditional Fortran users. In particular, integers could be of any size, since the underlying machine supported arbitrary precision "bignums".

We would prefer to avoid translation to a primitive kind of Fortran-engine implemented in Lisp. Once altered to this new model, further modification or adaptation of the translated program would be rather difficult; debugging of programs in translation might require knowledge of Fortran, Lisp, and their relationship.

Thus we don't endorse this as a "good" solution to compiling Fortran to Lisp. While in principle one could write a compiler that would produce reasonably efficient code for this model, the idea that any program could alter almost any location in memory inhibits some useful optimizations. Dealing with "dusty deck" or "legacy" code may require such treatment.

Another route is to provide source-to-source translation while attempting to retain some semblance of readability (for someone familiar with Lisp). The goal is to keep as much of the Fortran names and operations as possible. While this can be done for many of the routine arithmetic statements encountered, there are several problems in the automatic translation of function or subroutine calls.

The primary concern here is modeling of Fortran's passing parameters by reference. Lisp provides a style of call-by-value[5].

3 Some Examples Where Call by Reference Matters

3.1 Assigning to Parameters

Consider the Fortran procedure

```
subroutine addone(i)
i=i+1
return
end
```

And the use of it via

```
j=0
call addone(j)
```

which results in the setting of `j` to 1.

The apparently corresponding Lisp code,

```
(defun addone(i) (setf i (+ i 1)))
```

does not have the corresponding effect;

```
(setf j 0)
(addone j)
```

[5]Why Fortran uses call-by-reference may have more to do with the history of the implementation than aesthetics. In retrospect, language designers might view Fortran's machine memory model as a defect, but this appears to be irrelevant to most Fortran programmers.

does not affect the value of j at all. The local variable i inside addone has been set to 1, and then vaporized on exit. The value of j is still 0.

One proposal [7] is to model call by reference parameters as pairs, an accessor and a mutator. Under such a model, a variable such as j really has two associated functions. For convenience in operating with them, we can cons them into a Lisp pair:

```
(defun setupvar (initial)
  ;; create a lexical closure with given initial value, 2 accessors
 (let ((j initial))
    (cons #'(lambda() j) #'(lambda(value)(setf j value)))))

;; accessing the value in that closure

(defmacro valueof (x)
  `(funcall (the function (car ,x)))) ;accessor

(defmacro setter (x v)
  `(funcall (the function (cdr ,x)) ,v)) ;setting program
```

Then the program we wish to emulate can be written in Lisp this way:

```
(defun addone(j)  ;; add one to j.
  (setter j (+ 1 (valueof j))))
```

and the main program is

```
(setf j (setupvar 0)) ;initialize j to 0
(addone j)
```

A version of addone that might compile into more efficient code (your compiler's mileage may vary, as they say) could be written:

```
(defun addone(j)
  (declare (optimize (speed 3)(safety 0)(debug 0)))
  (setter j (+ 1 (the fixnum (valueof j)))))
```

Another possibility is to store the data item in an array, say

```
(setq ar (make-array 1 :element-type 'fixnum :initial-element 0))
```

and then addone looks like this:

```
(defun addone(a)
  (declare (type (simple-array fixnum (*)) a))
  (setf (aref a 0) (+ 1 (the fixnum (aref a 0)))))
```

Another possibility is to use the technique in f2cl [3], using Lisp's multiple-value-bind to simulate call by value-return.

Assume we know nothing about the subroutine sub except that it takes four arguments. In fact the definition of sub may be in a separate file. If we encounter this statement,

```
call sub(a,34,b+c,d)
```

this call could potentially change values of a and b. Assume the definition of sub looks like

```
subroutine sub(x,y,z,w)
...
  return
...
end
```

Then the call to `sub` is converted to this:

```
(multiple-value-bind (n1 n2 n3 n4)
    (sub a 34 (+ b c) d)
    (when n1 (setf a n1))
    (when n2 (setf d n4))
  ...
  )
```

and the definition of `sub` is converted to this:

```
(defun sub(x y z w)
  (block nil
   ...
  (return (values x y z w))
  ...))
```

Note that the setting of `y` and `z` in the case where they are bound to numeric or boolean constants or to expressions can be reflected in the setting of only a subset of the parameters to their return values. For the other arguments, a check is generated to determine if `n1` is non-`nil` because that is how the called function indicates that the corresponding argument value is not modified.

While the check for a `nil` return value may seem extraneous, some existing compilers are smart enough to delete the entire `when` expression if the return types of the `sub` routine are known at the time the call is generated. Likewise, the test can be deleted, leaving only the `setf` if it is known the return value can never be `nil`. Therefore, there is no additional cost for the caller to check. CMUCL is smart enough to do this.

Indeed, if the routine `sub` does not modify the `y` and `z` arguments, then the routine is actually converted to

```
(defun sub(x y z w)
  (block nil
   ...
  (return (values x nil nil w))
  ...))
```

If the definition of `sub` is known prior to its use and it can be determined that `sub` does not ever change a particular input argument, then that argument is removed by `f2cl` from the `multiple-value-bind`. Indeed, if it can be determined that none of the arguments are ever altered inside `sub`, the Fortran call is correctly modeled by an ordinary Lisp call. However, this feature is not currently implemented in `f2cl`, mostly because of a lack of an appropriate user interface to the functionality.

To the experienced Lisp programmer, the use of `multiple-value-bind` "just in case" something is changed seems quite wasteful in the sense that usual foundation of all Lisp computation is a functional model, and the redistribution of parameters after a call/return looks painful. Partly the traditional Lisp program tends to be very short with many calls, and so the pain would be repeated. Fortran programs, on the other hand, are ordinarily considerably longer and spend more time between call interfaces. In any case, it is worth exploring whether the fears of the Lisp programmer are well-founded.

How long do these variations take to run in a modern Lisp implementation? Compare these situations:

- an empty loop (doing nothing but looping),
- a loop incrementing an integer (`fixnum`) value probably in a register or incrementing two integers,
- using the `funcall` of accessors,
- passing an array and changing a value in it
- using a simple-vector, a simpler form of array.
- putting together the kind of program which would be automatically generated, using the `multiple-value-bind values` work-around suggested above.

3.2 Timing Assignments to Parameters

Here are some times we have collected. The code used for the tests follows below:

```
;; All compiled with optimization set for highest speed.

(declaim (optimize (speed 3) (safety 0)))

;;; The ``empty loop'' for comparing timing purposes

(defun test-empty-loop (n)
  (declare (fixnum n))
  (dotimes (i n)
    (declare (fixnum i))
    ))

;; Count with one local counter
(defun test-regular-count (j n)
  (declare  (fixnum j n))
  (let ((localj j))
    (declare (fixnum localj))
    (dotimes (i n)
      (declare (fixnum i))
      (setf localj (+ 1 localj)))
    localj))

;; Count with 2 local counters (see notes)
(defun test-rc2 (n)
  (declare  (fixnum n))
  (let ((localj 0)(localk 0))
    (declare (fixnum localj localk))
    (dotimes (i n)
      (declare (fixnum i))
      (setf localj (+ 1 localj))
      (setf localk (+ 1 localk)))
    localj))

;; Extract value, increment, set via funcalls
(defun test-funcalls (j n)
  (declare  (fixnum n))
  (dotimes (i n)
    (declare (fixnum i))
    (setter j (the fixnum (+ 1 (the fixnum (valueof j)))))))

;; Pass an array, change its 0th location
(defun test-array (a n)
  (declare  (fixnum n) (type (simple-array fixnum (*)) a))
  (dotimes (i n)
    (declare (fixnum i))
    (incf (the fixnum (aref a 0)))))


(defun array-callee (a)
  (declare (type (simple-array fixnum (*)) a))
  (incf (the fixnum (aref a 0))))

;; Like test-array, but call a routine to do it
(defun test-array-call (a n)
  (declare  (fixnum n) (type (simple-array fixnum (*)) a))
  (dotimes (i n)
    (declare (fixnum i))
    (array-callee a)))

;; Use a simple-vector rather than a specialized array
(defun test-sv (a n)
  (declare (fixnum n) (type (simple-vector *) a))
  (dotimes (i n)
    (declare (fixnum i))
    (incf (the fixnum (svref a 0)))))

(defun sv-callee (a)
  (declare (type (simple-vector *) a))
  (incf (the fixnum (svref a 0))))

;; Like test-sv, but call a routine to do it
(defun test-sv-call (a n)
  (declare (fixnum n) (type (simple-vector *) a))
  (dotimes (i n)
    (declare (fixnum i))
    (sv-callee a)))

(defun callee (h)
  (declare (fixnum h))
  (values (incf h)))

;; Multiple-value-bind and set for each call
(defun test-call (n)
  (declare (fixnum n))
  (let ((res 0))
    (declare (fixnum res))
    (dotimes (i n)
      (declare (fixnum i))
      (multiple-value-bind (a)
          (callee res)
        (setf res a)))
    res))

(defparameter *k* (setupvar 0))
(defparameter *ar* (make-array 1 :element-type 'fixnum :initial-element 0))
(defparameter *s* (make-array 1 :element-type t :initial-element 0)) ;simple

(defparameter *n* 536870910) ;a large fixnum integer, 5.37E8
```

Tests were done in two systems. ACL is Allegro CL 6.2 on an Intel Pentium 3 running at 933 MHz under the Windows 2000 operating system. CMUCL is Carnegie Mellon Common Lisp CMUCL version pre-18e, on an Intel Pentium 3 running at 866 MHz. See Table 1 for the results. A variation of a few percent on repeated tests is typical.

Function			Times in seconds	
			ACL	CMUCL
(test-empty-loop		*n*)	1.20	1.27
(test-regular-count	0	*n*)	1.78	1.29
(test-rc2		*n*)	1.79	1.89
(test-funcalls	*k*	*n*)	72.10	30.39
(test-array	*ar*	*n*)	9.65	2.12
(test-array-call	*ar*	*n*)	23.99	11.42
(test-sv	*s*	*n*)	3.03	2.09
(test-sv-call	*s*	*n*)	17.37	11.15
(test-call		*n*)	14.42	11.00

Table 1: Timing results

Our conclusion: the funcall method is slow; arrays or vectors present negligible overhead, enveloping each call in a `multiple-value-bind` is feasible and in this extremely intensive calling situation is actually faster after appropriate optimizations. (This was not so in an earlier version of the Allegro compiler). If an actual application includes calls with many parameters, requiring many variables to be copied over, it might take longer, as a percentage. Redoing `test-call` with two parameters but using only one does not slow it down.

4 Arrays, Parameters, Slicing

In Fortran 77, a formal or "dummy" argument as used in a procedure (subroutine or function) header is a symbolic name that identifies a variable, array or procedure[6]. Each dummy argument is associated with an actual argument, and should agree in number, and have appropriate type. The actual argument for a

[6]An asterisk dummy argument is used for an alternate return specifier. This is excluded from the official subset Fortran 77, and we will ignore it for this discussion.

dummy argument can be (1) An expression, (2) An array name, (3) An intrinsic function name, (4) An external procedure name, (5) A dummy procedure name.

(From the Fortran 77 ANSI standard, page 15-16, section 15.9.3)

> "If an actual argument is an expression it is evaluated just before the association of arguments takes place. If the actual argument is an array element name, its subscript is evaluated just before the association of its arguments takes place. Note that the subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association."

(That is, we have call-by-reference, not call-by-name). There is also an indication that one cannot pass an adjustable array[7] unless the adjustable dimension is in fact associated with an integer value.

It is useful also to resolve the question in Fortran 77 of how "aliasing" of variables is treated. Specifically, one can ask if any legitimate use of the call-by-reference mechanism can be simulated (if convenient) by the call-by-value-result mechanism. The answer is yes: According to section 15.9.3.6,

> "if a subroutine is headed by
> `SUBROUTINE XYZ (A,B)`
> and is referenced by
> `CALL XYZ (C, C)`
> ... then neither `A` nor `B` may "become defined" during this execution of subroutine `XYZ` or by any procedures referenced by `XYZ`."

That is, one could implement

```
SUBROUTINE XYZ (A,B)
```

by setting up two new (distinct) storage locations

```
XYZinternalA=A
XYZinternalB=B
```

compute with these values, and just before returning, set

```
A = XYZinternalA
B = XYZinternalB
```

Consider the Fortran program segment:

```
real x(100)
call sub(x(4))
```

As mentioned earlier, there are essentially two possibilities inside subroutine `sub`. The first is that `x(4)` would be used as a single real value, and possibly even changed and returned. The second possibility is that the parameter in `sub` is itself an array, say `dimension y(10)` in which case locations `x(4)` through `x(14)` may be changed by changing `y(1)` through `y(10)`. Even if the parameter in `sub` is not dimensioned, if the parameter is passed to yet another subroutine, it may be dimensioned there!

If one knows that the first usage is the only one of interest, (and `f2cl` can figure this out given the full definition of the subroutines at the right time), then the code can be conceptually rewritten as follows:

```
real x(100)
real tmp
...
```

[7]one dimensioned as `X(*)`

```
        tmp = x(4)
        call sub(tmp)
C and optionally, if tmp, meaning x(4) itself might be changed,
        x(4) = tmp
```

What if the Fortran program actually makes heavy-duty use of shared array parameters, as

```
        real x(100,100), y(100,100)
        ...
        call copyrow(x(5,*), y(10,*), 100)
        ...
        return
        ...
        end
```

This is a fairly common technique in Fortran code and is typically used with array arguments that are used as scratch space[8]. In fact, MPFUN uses this quite heavily and proved to be a major challenge to f2cl. We will see later in Section 6 how this is handled by f2cl; we introduce the basic idea here.

First note that Fortran arrays are stored in *column-major order* instead of *row-major order* that Lisp and C use. This means that the array is stored in memory in the order x(1,1), x(2,1), x(3,1),. . . . Lisp usually stores the array x(1,1), x(1,2), x(1,3), ... To model this correctly, f2cl actually uses 1-dimensional arrays for everything and generates the appropriate index from the multiple indices to access the desired element.

Then Lisp's displaced arrays can be used to generate the appropriate array slice. Thus the example above becomes something like

```
        (let ((x (make-array 10000 :element-type 'single-float))
              (y (make-array 10000 :element-type 'single-float)))
          ...
          (let ((x-dis (make-array 9995 :element-type 'single-float
                                   :displaced-to x
                                   :displaced-index-offset 5))
                (y-dis (make-array 9990 :element-type 'single-float
                                   :displaced-to y
                                   :displaced-index-offset 10)))
            (copyrow x-dis y-dis 100))
            ...)
```

5 A Painful Alternative

In the introduction we mentioned the possibilty of implementing Fortran in Lisp using a lower-level simulation of Fortran memory. Fortran and Lisp are computationally "equivalent" in the sense that we can map either of them into a binary memory of a digital computer[9]. Based on this equivalence we can specify that all of "Fortran integer memory" resides in an array in Lisp, fim. All of "Fortran double-float memory" resides in an array fdm, etc. In this case each subroutine which references j, a Fortran integer variable, actually is passed an integer m_j such that j = (aref fim m_j).

That means the subroutine addone in Lisp is passed m_j.

The whole system now begins to look like this

[8]Fortran does not have dynamic allocation so either the scratch space is allocated to a maximum size in the routine itself or the routine has additional parameters for the scratch space

[9]Whether this makes them "Turing equivalent" or equivalent to a finite state machine with very many states, is not an important distinction to make at this time.

```
(defvar *fimsize* 10000) ;or whatever is needed
(defvar *fdmsize* 10000)
(defvar fim (make-array *fimsize* :element-type 'fixnum))
(defvar fdm (make-array *fdmsize* :element-type 'double-float))

(defconstant m_j 1)             ; or where ever j is allocated

(defun addone(h fim dfpm)
  ;;pass memories to each routine, or address them globally
  (declare
           (fixnum h)
           (type (array fixnum (*)) fim)
           (type (array double-float (*)) dfpm)
           (ignore dfpm) ;unused in this routine
           )
  (setf (aref fim h)(+ (aref fim h) 1))) ;(incf (aref fim h)) would do same

;; the main routine would now look like
(setf (aref fim m) 0)           ;set j to 0
(addone m fim fdm)              ;pass memories as extra parameters
```

How fast is this, now? Consider testing this

```
(defun test5 (fim h n)
  (declare (fixnum n)
  (type (simple-array fixnum (*) fim)
        (type (array double-float (*)) dfpm)
        (ignore dfpm)        ;unused in this routine
        (fixnum h n)
        (optimize (speed 3)(safety 0)(debug 0)))
  (dotimes (i n)
    (declare (fixnum i))
    (incf (the fixnum (aref fim h))))))
```

Running `(test5 ar 0 n)` on ACL takes 17.10 seconds using `aref`, and 13.12 seconds using `svref`. This can be compared most directly to our previous time of 14.42 seconds with `test-call` using `multiple-value-bind`. In fact the program we are timing is identical to `test-sv` except that the index into the array `fim` is a parameter `h` here instead of the constant 0. Otherwise the timing would be down to 3 seconds for `svref`. In many cases the references *would be* to constant locations in `fim` and so the advantage might actually appear. This advantage to using `svref`, while useful in boosting speed, might be largely masked by the rest of a normal function call, which in this case adds at least 14 seconds more to our timing. Assuming that there are only a limited references to the parameter values inside the subroutine or the repeated subscript computation is optimized away, this could be significant, though the modification of the source code to being far less readable might also be a factor. In summary, we could gain some speed over `multiple-value-bind` call this way but a more realistic comparison in our previous table, comparing `test-sv-call` and `test-call` suggests that `multiple-value-bind` might be faster in simple cases.

6 Experience with `MPFUN`

`MPFUN` [5] is a large set of Fortran subroutines to support multiple-precision arithmetic (arbitrary but pre-specified length) written by David Bailey. It was provided to Raymond Toy by Richard Fateman as a challenge to his latest version of `f2cl`. Fateman's original interest in `MPFUN` was in using it via a Foreign Function Interface to provide access from Lisp to fast state-of-the-art bigfloat functionality including elementary functions and number theoretic routines. There are now several version of this code, including C and Fortran 90. Our conversion was based on the Fortran 77 version.

6.1 Problems in Conversion

Although `MPFUN` is written in a very portable Fortran 77 style, it proved to be quite a challenge for `f2cl` to generate *efficient* code. The two main problems were array slicing and array access.

6.1.1 Fast Array Access

In many Lisp implementations, *specialized arrays* are available to hold the common (Lisp) data types such as `(signed-byte 32)`, `single-float`, and `double-float`. These are efficient because the array elements are stored sequentially in a contiguous piece of memory without pointer indirection or extraneous information interspersed. To compute the address of an element of the array all that is required is to add an offset to the start of the array. This is very similar to the primitive Fortran and C array access methods.

When the Lisp array is *not* a specialized array, array accesses are significantly more complex. Each array has an array header containing information about the array: the dimensions, whether it is "displaced" or not, a "fill-pointer", etc., and, finally, a pointer to the actual data[10]. Thus, to access a pointer to an element of the array, the pointer to the data must be extracted from the header and added to the appropriate offset.

Thus, several memory reads are needed to finally access the element both delaying access and negatively affecting the CPU memory cache. By comparsion, with specialized arrays, accessing the desired element takes only one memory read. Consequently, if we expect programs to be as efficient as the relatively primitive C or Fortran programs in array accessing, we must use specialized arrays.

6.1.2 Array Slicing

As mentioned earlier, array slicing can be implemented easily in Lisp with displaced arrays. However, displaced arrays are not specialized arrays and thus, any routine that accepts array slices must declare the arrays as arrays of arbitrary un-typed objects instead of specialized arrays[11].

To solve this problem, `f2cl` generates code that accepts un-typed arrays but then uses `array-displacement` to extract the actual array and the index into the array for this displaced array. Since any Fortran array is of a known type, the actual array is, eventually, a specialized array. Thus, for the routine `SUB` above, we might have the translation:

```
(defun sub (x)
  (declare (type (array single-float (*)) x))
  (multiple-value-bind (a offset)
     (array-displacement x)
    ...
    (setf (aref a (+ index offset)) 4.0)
    ...
    ))
```

Of course, the array `A` may also be a displaced array, so we must continue calling `array-displacement` until the final array is not displaced. To complete the code generation we have to accumulate all of the displacement indices to find the final total displacement.

[10]Consult a Common Lisp reference for the detailed meanings of these properties

[11]Such a routine will still run even when specialized arrays are passed in, but significantly slower in either case.

While this seems rather expensive, it turns out that chasing down the array displacements is typically very cheap. The major cost is having to do one extra addition to access an array element. Experiments with `MPFUN` have shown that this is a major efficiency boost compared to using general Lisp array access.

6.2 Efficiency

With the changes and enhancements to `f2cl` mentioned above, the translated `MPFUN` was used to compute π to approximately 7390 digits using the sample program supplied with `MPFUN`.

As a comparison, the Fortran version of the test was compiled using g77 with -O optimization. This test computes our benchmark in approximately 0.9 sec on a Pentium III running at 866 MHz.

The Lisp version of this, using CMUCL pre-18e, ran in 2.63 sec, of which 0.6 sec were spent on GC'ing 34 MB of memory.

Analysis of the runtime of the Lisp code indicates several areas of weakness in this particular implementation:

- In a typical Lisp implementation of a built-in routine, the code is written to expect almost any kind of correct or incorrect argument. The generic argument is tagged in some way; typically for floats, the actual argument is a tagged pointer saying "this is a pointer to a float." (Encodings of small integers may carry their own tags, but larger integers, say more than 28 bits, may also need pointers. In these tests, significant amounts of garbage are generated because integers and floats need to be "boxed" when calling such routines. This boxing convention can sometimes be over-ridden by clever inter-procedural optimization or in-lining, but this is sometimes not done. The key to efficient numerical processing in Lisp is to deal with specially declared arrays where it is possible to store and operate on unboxed floats and integers.

- Because of the implementation of arrays using Lisp's displaced arrays, we expected the runtime to be about double the runtime of the Fortran implementation. However, the CMUCL compiler did not keep a pointer to the current element of an array, but always had to add an offset to a pointer to the start of the array. Thus at least three instructions are needed to access an array element.

- Truncation of a single-float to an integer, also represented as a single-float (Lisp's `ftruncate`), was initially the primary consumer of CPU time and producer of garbage. CMUCL reduced this cost by being able to inline this function automatically.

However, we consider the following issues to be positive:

- The FFT routine that forms the heart of the `MPFUN` multiplier was implemented without consing.

- The use of displaced arrays to implement array slices was not, as revealed by more detailed profiing, a significant expense.

- The cost of computing `array-displacement` for array slices was also in the noise.

- Array slicing did not really hurt the performance, except for the need for an additional add operation to access an array element. We view this as an area for improved optimization of the Lisp compiler and not an inherent defect.

7 Conclusions

- Translation from Fortran to Common Lisp is a feasible alternative to foreign-function interfaces and loading of Fortran modules, at least in the case that the (Fortran) source code is available. Given the current compiler technology, the penalty for automatically translated code may be acceptable; further improvements are identifiable.

- The close interaction between symbolic and numeric computation can be maintained while maintaining Common Lisp's close control error handling, debugging, and interaction. For example `f2cl` is being used by the (free, open-source) version of Macsyma (called Maxima [8]) to provide quality numerical evaluation routines for selected special functions.
- We avoid data transformation that is sometimes required in some alternative relatively crude interchanges between symbolic and numeric computations. Sometimes these interchanges even require running in separate processes or separate computers. We are not arguing against all such interchanges, just those born of desperation where it seems the only way to proceed with a computation is to print out an intermediate result as a character string (perhaps a character string that is some bloated XML encoding) and re-parse it elsewhere.
- The resulting Common Lisp code is portable across a variety of Common Lisp implementations, at least those which correspond to an ANSI standard.
- No Fortran compiler is needed at any point in this process.
- The resulting code is still largely readable and reflects the data organization and computation of the original Fortran. It could in principle be used as the basis for more general bigfloat computation (although we would expect revision to be necessary: this is, after all, still Fortran code specifying fixed-precision numerical computation).
- On at least some implementations (with appropriate optimization performed in the compiler), array slicing can be implemented reasonably efficiently with displaced arrays.
- Multiple-value return and binding is an effective and reasonably efficient way to handle Fortran's call-by-reference semantics in Lisp.

8 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

We thank Kevin Broughan *et. al.* for releasing `f2cl` under the GPL license so that further work could be done on it. We thank David Bailey for releasing his MPFUN code as well.

References

[1] American National Standards Institute, Inc ANSI Common Lisp. We don't recommend obtaining it, but the official standard is available as ANSI X3.226-1994 from `http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI+X3%2E226%2D1994`. A more useful reference is Paul Graham: *ANSI Common Lisp*, Prentice Hall, 1995. There are also several on-line references, most notably `http://www.lisp.org/HyperSpec/FrontMatter/` and `http://www.franz.com/search/search-ansi-about.lhtml`. These are almost universally used as resources for the language definition.

[2] American National Standards Institute, Inc ANSI X3.9-1978 ISO 1539-1980 (E) American National Standard Programming Language FORTRAN (1978)

[3] Kevin A. Broughan, Diane M. K. Willcock: "Fortran to Lisp Translation using f2cl." *Software - Practice and Experience 26(10)*: 1127-1139 (1996). See `http://sourceforge.net/projects/clocc` for more recent work.

[4] Common Lisp Open Code Collection. `http://sourceforge.net/projects/clocc`

[5] David Bailey. A multiple precision package available at `http://www.netlib.org/mpfun/`

[6] Kevin Rosenberg. `http://uffi.med-info.com/` Lisp Universal Foreign Function Interface.

[7] Guy L. Steele, Jr. and G.J. Sussman. “Lambda the Ultimate Imperative,” `http://library.readscheme.org/page1.html`

[8] Maxima computer algebra system `http://maxima.sourceforge.net/`