# The FFA library

Tamás K Papp (tpapp@princeton.edu)

September 14, 2007

## 1 Introduction

Even though Common Lisp has extensive libraries, sometimes the need arises to call functions written in other languages, especially C. While CFFI provides a comfortable and unified interface for most purposes, using functions that expect to find or output arrays at locations specified by a pointer still doesn't have a common interface. Constructing an array at a memory location is always possible by allocating a chunk of memory and copying the array elements in and out manually, but some implementations provide direct access to an unboxed array at a memory location (eg SBCL).

The FFA (Foreign Friendly Array) library provides an interface that allows the user to map Lisp arrays into a specified memory location for the body of a macro call. This macro has well-defined semantics, explained in Section 3, which is implemented differently for various implementations to take advantage of implementation-specific optimizations offered.

Note that the approach of this package is to keep arrays in Lisp, and provide access to arrays mapped to a given pointer on demand, incurring the possible overhead at the time. A different approach is taken by CL-BLAPACK, which keeps vectors in foreign memory, where they are untouched by the GC. Each design choice has advantages and disadvantages. Keeping the arrays in Lisp was chosen because

- access to array elements is fast, several implementations optimize `aref` quite well
- existing code doesn't have to be made aware of some special array type and can continue to use plain Lisp arrays
- foreign memory is usually more scarce than memory available in Lisp[1]

[1] This is relevant even in implementations that require copying. Imagine that you have 100 matrices, and you are calling a foreign function that needs a few of them at a time.

There are some disadvantages: (1) some implementations require copying, introducing an overhead, (2) sometimes you need to have an array that stays in one place for a long time (eg for callback functions), and (3) Common Lisp arrays are row-major, while some languages and libraries expect column-major arrays by default.

Regarding (1), FFA doesn't claim to be superfast in all implementations — it simply *works*. If you find that copying is an unbearable overhead, you can either switch implementations, or ask the authors of your implementation for arrays that can be pinned. There is simply no way to make array access fast in every situation without the right facilities. For (2), the answer is that FFA is not the right library for you: simply allocate a chunk of memory and free it manually when you are done.

Regarding the row-major vs. column-major issue: some foreign libraries (eg BLAS, Atlas) accept both kinds of arrays, so there is no need to transpose. In the worst case, you can easily write a transpose function in either C or Lisp, the latter is provided in this package.

# 2 Flat arrays

For the purposes of this package, flat arrays are arrays that have a rank of one, ie are one-dimensional. Flat arrays are particularly important because most implementations that allow direct access to unboxed arrays require that these arrays are simple-arrays of rank one. Fortunately, Common Lisp has displaced arrays, so you can always create a flat array and make it the target of a displaced array of the desired dimension.

This is what `make-ffa` does. It uses the following syntax:

```
make-ffa dimensions element-type &key
        (initial-element 0 initial-element-p)
        (initial-contents nil initial-contents-p)
```

where all the arguments are the same ones that would be provided for `make-array`, except that `initial-element` and `initial-contents` will be coerced to the desired type, and `element-type` (which is mandatory, not optional) is also allowed to be one of

```
(:int8 :uint8 :int16 :uint16 :int32 :uint32 :float :double)
```

in which case the corresponding Lisp type is chosen. If `dimension` is a list with more than one element, a flat array is created, and a displaced array is returned. `make-ffa` does not guarantee that the resulting array is unboxed, because this is always implementation-dependent, but it guarantees to try its best.

Even if you don't use foreign functions, arrays displaced to flat arrays are quite handy. For example, if we want to sum elements in an array, we just find the original array (see `find-original-array`) and call reduce. If the original array is not flat, `find-or-displace-to-flat-array` will provide a displaced flat one, but benchmarks indicate that calling reduce on this is not as efficient as calling reduce on a flat, non-displaced array of the same dimension.

Some handy operations are written in CL and provided in `operations.lisp`, including `array-reduce` (which has an option for "ignoring" `nil` values), and the derived functions `array-max`, `array-min`, `array-sum`, `array-product`, `array-count` and `array-range` — see the functions for details. You can get a generalized outer product using `outer-product`. See the source code for more useful operations.

You can make a copy of an array using `array-copy`, which, however, does not descend into the elements (eg if you are using this array to store lists, the elements of the new array will reference the same lists as the elements in the old one). You can also map arrays (elementwise) into arrays of the same dimensions (but not necessarily the same element-type) with `array-map`, which you can also use for a deep copy with the appropriate function. The function `array-convert` is a specialized version of `array-map`, converting array elements using `coerce`.

# 3 with-pointer-to-array

The key macro is

```
(with-pointer-to-array (array pointer cffi-type length
   direction)
  &body body)
```

Its semantics are defined as follows. Within the body of the macro, the `pointer` will be a CFFI pointer pointing to a contiguous region in memory, which contains `length` elements of type `cffi-type`. If direction is either `:copy-in` or `:copy-in-out`, the area is guaranteed to contain the elements of the array at the beginning of the body of the macro. If direction is one of `:copy-out` or `:copy-in-out`, the array is guaranteed to contain the elements of the memory location after the body of the macro ends.

Note that `cffi-type` doesn't have to match the element-type of array. It is advised that it does, because otherwise the macro will try to coerce the elements to the desired foreign type before copying, which can result in possible errors or an efficiency loss.

Also note that `array` does not have to be a simple, flat or unboxed array. If the implementation can't provide direct access to that array type, the elements will be copied. The note above about efficiency also applies here.

To make this more clear, in the example below an array is created using `make-array` with element-type `integer`. Then at runtime, elements are coerced to `:int32` (if they are small enough, which holds here). This is not efficient, so in SBCL, a warning is issued.

```
(defun cffi-fill-int32 (pointer size)
  "Fill␣array␣at␣pointer␣with␣size␣integers."
  (dotimes (i size)
    (setf (cffi:mem-aref pointer :int32 i) i)))

(let* ((a (make-array '(2 9) :element-type 'integer))
            (a-d (displace-array a 9 3)))
  (with-pointer-to-array (a-d pp :int32 7 :copy-in-out)
    (cffi-fill-int32 pp 7))
  a) ; => #2A((0 0 0 0 1 2 3 4 5) (6 0 0 0 0 0 0 0 0)))
```

If frequently happens that you have to pass more than one array to a foreign function. For those cases, you can use the convenience macro

```
(with-pointers-to-arrays ((a a-pointer a-type a-length a-dir)
                          (b b-pointer b-type b-length b-dir)
                          ...)
  ...body...)
```

which expands into nested calls of `with-pointer-to-array` using the given parameters.