array-lib egg

Provides a SRFI-63 workalike
Extension for Chicken Scheme
Version 1.7

Kon Lovett

Table of Contents

1 About this egg

1.1 Version history

`1.7` Bug fix for ec

`1.6` Removal of common code, partial support for csi describe

`1.5` Exports

`1.4` Replaced #<rank>A print syntax with #A read/print syntax, bug fix for returning underlying storage object

`1.3` Optimized array construction

`1.2` Bug fix for unbound-variable array:store

`1.1` Record optimization

`1.0` Indexing optimization

`0.9` more SLIB procedures

`0.8` SLIB procedures

`0.7` Added procedures

`0.6` Added procedures

`0.5` Added procedures

`0.4` Added procedures

`0.3` Bug fix for empty arrays

`0.2` Bug fix for rank 0 arrays

`0.1` Initial release

1.2 Usage

Load this egg like so:

```
(require-extension array-lib)
```

1.3 Requirements

This egg requires the following extensions:

```
misc-extn
```

2 Documentation

The array-lib purports to be a Chicken workalike for SRFI-47, SRFI-63, and the SLIB array, subarray & arraymap modules.

Note: this array package is *not* compatible with the default array library SRFI-25 that is provided with the base system. Do not use these two libraries together in the same program. However it does provide a similar `shape` feature, but with a SRFI-63 compliant API.

2.1 SRFI-63 Discussion

The signatures of all SRFI-63 procedures are supported by this API, only with extended interpretations of some arguments.

Both SRFI-47 and SRFI-63 array prototypes are supplied, including complex prototypes (see below).

Array constructors will return the underlying storage object, rather than an array object, when the constructed array is 0-origin & rank <= 1, per SRFI-63.

Where dimensions are expected a shape is also accepted. Where a rank is expected dimensions or a shape are also accepted.

Dimensions are a list of the number of elements for each dimension, or a dimension object (see below). For example, `'(1 2)` are dimensions of a rank 2 array.

A shape is a list of intervals, [lower upper), or a shape object (see below). For example, `'((0 1) (0 2))` is a shape of a rank 2 array, with bounds [0 0] [0 1].

The dimensions `'(2 0 3)` are equivalent to the shape `'(0 2 0 0 0 3)`.

The accepted list form of dimensions or shape for procedures taking these as tail arguments is as an 'exploded' list. Each element as an individual argument. For example, `(make-array '(1 2) '(-8 -5))`, not `(make-array '((1 2) (-8 -5)))`.

All indices/dimensions/bounds/ranks **must** be fixnum. Not much of a restiction from SRFI-63, just a clarification.

An empty dimension, a dimension of 0, is not actually empty. The empty dimension will have a single element, the subarray "inside". However, no storage is consumed by an empty dimension.

2.2 Read Syntax

A SRFI-10 printer and reader is supplied.

An array specific read/print syntax is supplied: `#A<rank><elements>`, where `<rank>` is the array dimensionality and `<elements>` are a rank-nested list consisting of all the elements, in row-major order.

The SRFI-10 print-form preserves the underlying storage model and shape information, unlike the #A form, which only preserves the rank and element values. So, reading the #A form may not reconstruct the exact printed array.

Use `(current-array-print-form ...)` to select the desired form.

A limit is placed on the number of array elements printed, which may be overridden. See `(current-array-print-count ...)`.

2.3 Parameters

`current-array-bounds-check` [parameter]

```
(current-array-bounds-check [FLAG #t])
```

Sets or returns whether array indices are bounds-checked.

`current-array-element-check` [parameter]

```
(current-array-element-check [FLAG #f])
```

Sets or returns whether array elements are immediately type-checked before setting. When false the underlying storage type performs any validation.

`current-array-print-count` [parameter]

```
(current-array-print-count [COUNT 50])
```

Sets or returns the number of array elements to print. A `COUNT` of `#f` will set the count 0, and `#t` will set the count to a very big number. When `COUNT` is a number it must a zero or positive fixnum.

`current-array-print-form` [parameter]

```
(current-array-print-form [FORM 'SRFI-10])
```

Sets or returns the format for printing an array, either `'SRFI-10`, `'A`, or `#f` which turns off array element printing.

2.4 Predicates

`array?` [procedure]

```
(array? ARRAY)
```

Returns whether the object is an array object, or accepted as an array. Acceptable objects are vector, string, byte-vector, and SRFI-4 vectors.

`array-strict?` [procedure]

```
(array-strict? ARRAY)
```

Returns whether the object is *only* an array object.

`equal?` [procedure]

```
(equal? ARRAY1 ARRAY2)
```

Are the two arrays equal in all properties and elements?

`array-equal?` [procedure]

```
(array-equal? ARRAY1 ARRAY2)
```

Are the two arrays equal in dimensionality and elements?

Signals an error when either of the arguments is not an array.

`array-empty?` [procedure]

```
(array-empty? ARRAY)
```

Returns `#t` for an empty array, `#f` for anything else.

`array-dimensions?` [procedure]

```
(array-dimensions? DIMENSIONS)
```

A set of dimensions, as produced by `make-array-dimensions`?

`array-shape?` [procedure]

```
(array-shape? OBJECT)
```

A shape, as produced by `make-array-shape`?

`array-in-bounds?` [procedure]

```
(array-in-bounds? ARRAY INDEX ...)
```

Are the indices valid for the array?

Signals an error when not an array.

2.5 Array Properties

`array-store` [procedure]

```
(array-store ARRAY)
```

Returns the underlying storage object for `ARRAY`, or `#f` when not an array. Use this at your peril.

`array-start` [procedure]

```
(array-start ARRAY RANK)
```

Returns the start, or lower bound, of `ARRAY` along dimension `RANK`, or `#f` when not array.

`array-end` [procedure]

```
(array-end ARRAY RANK)
```

Returns the end, one beyond the upper bound, of `ARRAY` along dimension `RANK`, or `#f` when not an array.

`array-rank` [procedure]

```
(array-rank ARRAY)
```

Returns the number of dimensions for `ARRAY`, or `0` when not an array, an empty array, or a rank 0 array.

`array-dimensions` [procedure]

```
(array-dimensions ARRAY)
```

Returns the list of dimensions for `ARRAY`, or `#f` when not an array. The elements are the actual length of each dimension, so a dimension of 3, [0 3), is 3 and [1 3) is 2.

`array-bounds` [procedure]

```
(array-bounds ARRAY)
```

Returns the bounds list for `ARRAY`, or `#f` when not an array. A list of pairs, one for each dimension. Each pair is the interval [low high].

`array-shape` [procedure]

```
(array-shape ARRAY)
```

Returns the shape list for `ARRAY`, or `#f` when not an array. The list can be used as `(apply make-array-shape LIST)`.

To access the compiled shape information of an array use `(array-bounds ...)`.

2.6 Array Shape & Dimensions

`make-array-dimensions` [procedure]

```
(make-array-dimensions DIMENSION ...)
```

Returns a dimensions object.

`make-array-shape` [procedure]

```
(make-array-shape START END ...)
```

Returns a shape object suitable for use with the array construction procedures.

A shape is composed of an even count series of intervals, [start end), for each dimension. Thus an index is within the bounds for a dimension if start <= index < end. To indicate an empty dimension use start = end, a special case.

The interval may be on any integer, not just positive and 0. The start must still be strictly less than the end. So [-5 -2) is legal but [-2 -5) is not.

2.7 Array Operations

`array-print` [procedure]

```
(array-print ARRAY [COUNT <very-big-number>] [PORT (current-output-port)])
```

Prints `COUNT` elements of the array `ARRAY` on the specified port `PORT`.

`array-copy` [procedure]

```
(array-copy [PROTOTYPE] ARRAY)
```

Returns a copy of the array. When prototype is missing a deep copy is performed. When the prototype and the source array are the same, `(eq? <prototype> <array>)`, a fresh array is returned with the same shape, using the array as the prototype. Otherwise a fresh array is returned with the specified prototype and elements from the array.

The prototype and source array element types must be assignment compatible. Even so, precision maybe lost.

`make-array` [procedure]

```
(make-array [PROTOTYPE] [SHAPE/DIMENSION ...])
```

Creates and returns an array of type prototype with dimensions or shape, and filled with elements from prototype.

A valid prototype is any object the meets `(array? ...)`.

If the prototype has no elements, then the initial elements of the returned array are unspecified. Otherwise, the returned array will be filled with the element at the origin of prototype.

`(make-array [<prototype>])` will construct an empty array, not a rank 0 array. Such arrays cannot be used with any setter or getter. However, property queries will work.

When the prototype is missing, vector is assumed.

`array` [procedure]

```
(array [PROTOTYPE] RANK/SHAPE/DIMENSIONS [ELEMENT ...])
```

Creates and returns an array of type prototype with dimensions, or shape, or rank and filled with the elements.

When a rank is specified, it must be 0 or positive. When positive it is interpreted as the dimensions ((- rank 0) (- rank 1) ... (/ <# of elements> (factorial rank))). So rank 1 is (<#>), 2 is (2 <#>/2), 3 is (3 2 <#>/3x2), and so on. This is of dubious utility.

When the prototype is missing, vector is assumed.

`make-shared-array` [procedure]

```
(make-shared-array ARRAY MAPPER [SHAPE/DIMENSION ...])
```

Can be used to create shared subarrays of other arrays. The mapper is a function that translates coordinates in the new array into coordinates in the old array. A mapper must be linear, and its' range must stay within the bounds of the old array, but it can be otherwise arbitrary.

`list->array` [procedure]

```
(list->array SHAPE/DIMENSIONS/RANK PROTOTYPE LIST)
```

The list must be a rank-nested list consisting of all the elements, in row-major order, of the array to be created.

Returns an array of rank, or shape, or dimensions and type prototype consisting of all the elements, in row-major order, of the list. When the rank is 0, list is the lone array element; not necessarily a list.

`array->list` [procedure]

```
(array->list ARRAY)
```

Returns the array elements as a rank-nested list. For rank 0 arrays returns just the element. Will generate an error when not an array.

`vector->array` [procedure]

```
(vector->array VECTOR PROTOTYPE [SHAPE/DIMENSION ...])
```

The vector must be of length equal to the product of the dimensions, or shape element sizes.

Returns an array of type prototype consisting of all the elements, in row-major order, of the vector. In the case of a rank 0 array, the vector has a single element.

`array->vector` [procedure]

```
(array->vector ARRAY)
```

Returns the array elements as a new unpacked vector. Will generate an error when not an array.

`array-ref` [procedure]

```
(array-ref ARRAY INDEX ...)
```

Returns the array element at indices. Will generate an error when not an array or out-of-range condition.

`array-set!` [procedure]

```
(array-set! ARRAY OBJECT INDEX ...)
```

Sets the array element at indices to the object. Will generate an error when not an array, out-of-range condition, or the object is unsuitable as an array element.

2.8 Sub-Array Operations

These are not part of the basic package. Be sure to `(use array-lib-sub)` for access.

`subarray` [procedure]

```
(subarray ARRAY [SELECT ...])
```

Returns a subarray sharing elements with `ARRAY`. For an array of rank N, there must be at least N `SELECT` arguments. For 0 <= I < N, `SELECT`[I] is either an integer, a list of two integers within the range for the Ith index, or `#f`.

When `SELECT`[I] is a list of two integers, then the Ith index is restricted to that subrange in the returned array.

When `SELECT`[I] is `#f`, then the full range of the Ith index is accessible in the returned array. An elided argument is equivalent to `#f`.

When `SELECT`[I] is an integer, then the rank of the returned array is less than `ARRAY`, and only elements whose Ith index equals `SELECT`[I] are shared.

`array-trim` [procedure]

```
(array-trim ARRAY [TRIM ...])
```

Returns a subarray sharing elements with `ARRAY` except for slices removed from either side of each dimension. Each of the `TRIM` arguments is an exact integer indicating how much to trim. A positive `TRIM` trims the data from the lower end and reduces the upper bound of the result; a negative `TRIM` trims from the upper end and increases the lower bound.

`array-row-ref` [procedure]

```
(array-row-ref ARRAY INDEX ...)
```

Returns the array row, as an unpacked vector, at indices. Will generate an error when not an array or out-of-range condition. The number of indices must be one less than the rank.

`array-row-set!` [procedure]

```
(array-row-set! ARRAY OBJECT INDEX ...)
```

Sets the array row at indices to the object. Will generate an error when not an array, out-of-range condition, or the object is unsuitable as an array row. The number of indices must be one less than the rank.

The row object may be an one dimensional array, a list, or an atomic value. The row object length must match the target array row length. When an atomic value this operation is row fill.

`array-split` [procedure]

```
(array-split ARRAY OUTER-RANK)
```

Return a list of fresh subarrays, split along the specified outer rank, in ascending order by outer index.

`array-split/shared` [procedure]

```
(array-split/shared ARRAY OUTER-RANK)
```

Return a list of shared subarrays, split along the specified outer rank, in ascending order by outer index.

`array-join` [procedure]

```
(array-join PROTOTYPE OUTER-SHAPE/DIMENSIONS [INNER-SHAPE/DIMENSIONS] [ARRAY ...]
```

Construct a fresh array with specified prototype, outer dimensions, and optional inner dimensions, from the specified inner array(s).

The outer dimension maybe a number (indicating the single, outer, dimension), a list of dimensions, a shape, or `#f` (indicating the count of the inner arrays is the outer dimension).

If specified the inner dimension maybe a list of dimensions, or a shape. If missing the first inner array shape is used as the inner shape for the result array.

The result array shape is the concatenation of the outer and inner shapes. The rank of the result array must be greater than the rank the inner array(s).

All joined arrays must be of the same dimensionality.

`array-reverse` [procedure]

```
(array-reverse ARRAY)
```

Returns a fresh array with the top-level dimension reversed.

2.9 Higher Order Operations

These are not part of the basic package. Be sure to `(use array-lib-hof)` for access.

`make-array-index-generator` [procedure]

```
(make-array-index-generator SHAPE/DIMENSIONS/ARRAY)
```

Returns a zero argument procedure which returns an indices list (a full index) for the specified bounds upon each invocation, or `#f` when the indices are exhausted.

An indices list is suitable for use with `(apply array-ref/array-set! ... indices)`.

`array-for-each-index` [procedure]

```
(array-for-each-index PROCEDURE [SHAPE/DIMENSIONS/ARRAY ...])
```

Invokes the procedure with each set of indices for the specified bounds. The procedure arity is equal to the number of bounds. Each argument is a list of the indices, with rank length (length of the bounds).

When an array is specified the bounds of the array are used.

An indices list is suitable for use with `(apply array-ref/array-set! ... indices)`.

All arrays should be of the same dimensions. The operation will terminate when any of the set of indices is exhausted.

`array-for-each` [procedure]

```
(array-for-each PROCEDURE [ARRAY ...])
```

Apply the procedure to each element of the array(s). The procedure arity is equal to the number of the array(s).

All arrays must be of the same dimensionality.

`array-map!` [procedure]

```
(array-map! ARRAY PROCEDURE [ARRAY ...])
```

Apply the procedure to each element of the array(s). The procedure arity is equal to the number of the array(s).

Returns the first array with the mapped elements. An in-place map.

All arrays must be of the same dimensionality.

`array-map` [procedure]

```
(array-map PROTOTYPE PROCEDURE ARRAY ...)
```

Apply the procedure to each element of the array(s). The procedure arity is equal to the number of the array(s).

Returns a fresh array from the `PROTOTYPE` and the shape of the first `ARRAY` with the mapped elements.

All arrays must be of the same dimensionality.

`array-fold` [procedure]

```
(array-fold PROCEDURE SEED ARRAY ...)
```

Apply the fold procedure to each element of the array(s). The procedure arity is 1 + the number of array(s). The first argument is the current accumulated value, or the initial seed. The rest of the arguments are the array elements.

Returns the accumulated value.

All arrays must be of the same dimensionality.

`array-project` [procedure]

```
(array-project PROTOTYPE RANK/SHAPE/DIMENSIONS PROJECTOR ARRAY)
```

Conform an N dimensional source array to a fresh M dimensional destination array.

The projector takes 2 arguments, the destination row vector and the source row vector. The destination vector length is M and the source vector is length N. The projector is not required to use the supplied destination vector. However, it must return some vector of length M.

Returns the array projection.

`array-index-map!` [procedure]

```
(array-index-map! ARRAY PROCEDURE)
```

Applys the `PROCEDURE` to each list of indexes of the `ARRAY` and replaces the existing element with the result of the procedure application.

`array-indexes` [procedure]

```
(array-indexes ARRAY)
```

Returns an array of lists of indexes for the `ARRAY`.

`array-copy!` [procedure]

`(array-copy! DESTINATION-ARRAY SOURCE-ARRAY)`

Copy the elements of the source array to the corresponding elements of the destination array. The source and destination arrays must have the same dimensions, but not necessarily the same shape.

The source and destination element types must be assignment compatible. Even so, precision maybe lost.

2.10 Eager Comprehensions

These are not part of the basic package. Be sure to `(use array-lib-ec)` for access. Requires SRFI-42.

`array-ec` [macro]

`(array-ec PROTOTYPE SHAPE/DIMENSIONS [QUALIFIER ...] EXPRESSION)`

Returns an array with specified `PROTOTYPE`, and `SHAPE/DIMENSIONS`. Each element of the fresh array is produced by evaluating the `EXPRESSION` for every state generated by the set of `QUALIFIER`.

Should more states be generated than elements of the array they are ignored.

`:array` [macro]

`(:array VARIABLE [(index INDEX-VARIABLE ...)] ARRAY ...)`

Generate each element of each `ARRAY`, binding the element to `VARIABLE`. Should the indices be wanted there must be as many `INDEX-VARIABLE` items as the rank of greatest dimensioned array.

2.11 Prototypes

`a:floc128b` [procedure]

`(a:floc128b [INITIAL <undefined>])`

Returns an inexact 128 bit flonum complex uniform-array prototype.

`a:floc64b` [procedure]

`(a:floc64b [INITIAL <undefined>])`

Returns an inexact 64 bit flonum complex uniform-array prototype.

`a:floc32b` [procedure]

`(a:floc32b [INITIAL <undefined>])`

Returns an inexact 32 bit flonum complex uniform-array prototype.

`a:floc16b` [procedure]

`(a:floc16b [INITIAL <undefined>])`

Returns an inexact 16 bit flonum complex uniform-array prototype.

`a:flor128b` [procedure]

`(a:flor128b [INITIAL <undefined>])`

Returns an inexact 128 bit flonum real uniform-array prototype.

`a:flor64b` [procedure]

`(a:flor64b [INITIAL <undefined>])`

Returns an inexact 64 bit flonum real uniform-array prototype.

`a:flor32b` [procedure]

`(a:flor32b [INITIAL <undefined>])`

Returns an inexact 32 bit flonum real uniform-array prototype.

`a:flor16b` [procedure]

`(a:flor16b [INITIAL <undefined>])`

Returns an inexact 16 bit flonum real uniform-array prototype.

`a:floq128b` [procedure]

`(a:floq128b [INITIAL <undefined>])`

Returns an exact 128 bit decimal flonum rational uniform-array prototype.

`a:floq64b` [procedure]

`(a:floq64b [INITIAL <undefined>])`

Returns an exact 64 bit decimal flonum rational uniform-array prototype.

`a:floq32b` [procedure]

`(a:floq32b [INITIAL <undefined>])`

Returns an exact 32 bit decimal flonum rational uniform-array prototype.

`a:fixz64b` [procedure]

`(a:fixz64b [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 64 bits of precision.

`a:fixz32b` [procedure]

`(a:fixz32b [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 32 bits of precision.

`a:fixz16b` [procedure]

`(a:fixz16b [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 16 bits of precision.

`a:fixz8b` [procedure]

`(a:fixz8b [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 8 bits of precision.

`a:fixn64b` [procedure]

`(a:fixn64b [INITIAL <undefined>])`

Returns an exact non-negative binary fixnum uniform-array prototype with at least 64 bits of precision.

`a:fixn32b` [procedure]

`(a:fixn32b [INITIAL <undefined>])`

Returns an exact non-negative binary fixnum uniform-array prototype with at least 32 bits of precision.

`a:fixn16b` [procedure]

`(a:fixn16b [INITIAL <undefined>])`

Returns an exact non-negative binary fixnum uniform-array prototype with at least 16 bits of precision.

`a:fixn8b` [procedure]

`(a:fixn8b [INITIAL <undefined>])`

Returns an exact non-negative binary fixnum uniform-array prototype with at least 8 bits of precision.

`a:bool` [procedure]

`(a:bool [INITIAL <undefined>])`

Returns a boolean uniform-array prototype.

`ac64` [procedure]

`(ac64 [INITIAL <undefined>])`

Returns an inexact 64 bit flonum complex uniform-array prototype.

`ac32` [procedure]

`(ac32 [INITIAL <undefined>])`

Returns an inexact 32 bit flonum complex uniform-array prototype.

`ar64` [procedure]

`(ar64 [INITIAL <undefined>])`

Returns an inexact 64 bit flonum real uniform-array prototype.

`ar32` [procedure]

`(ar32 [INITIAL <undefined>])`

Returns an inexact 32 bit flonum real uniform-array prototype.

`as64` [procedure]

`(as64 [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 64 bits of precision.

`as32` [procedure]

`(as32 [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 32 bits of precision.

`as16` [procedure]

`(as16 [INITIAL <undefined>])`

Returns an exact binary fixnum uniform-array prototype with at least 16 bits of precision.

`as8` [procedure]

```
(as8 [INITIAL <undefined>])
```

Returns an exact binary fixnum uniform-array prototype with at least 8 bits of precision.

`au64` [procedure]

```
(au64 [INITIAL <undefined>])
```

Returns an exact non-negative binary fixnum uniform-array prototype with at least 64 bits of precision.

`au32` [procedure]

```
(au32 [INITIAL <undefined>])
```

Returns an exact non-negative binary fixnum uniform-array prototype with at least 32 bits of precision.

`au16` [procedure]

```
(au16 [INITIAL <undefined>])
```

Returns an exact non-negative binary fixnum uniform-array prototype with at least 16 bits of precision.

`au8` [procedure]

```
(au8 [INITIAL <undefined>])
```

Returns an exact non-negative binary fixnum uniform-array prototype with at least 8 bits of precision.

`at1` [procedure]

```
(at1 [INITIAL <undefined>])
```

Returns a boolean uniform-array prototype.

3 Issues

SRFI-25 indexing is 2.4 times faster! However, 'array-lib' has faster indexing than the reference SRFI-63 implementation.

Should signal specific conditions rather than an error.

Array aggregate operations are implemented in an element by element fashion.

The SRFI-42 general dispatcher `(: ...)` does not recognize arrays.

The 64-bit prototypes are currently implemented as unpacked vectors.

The prototypes will play well with the 'numbers' egg.

Needs a comprehensive test suite.

4 Examples

```
(use array-lib)

(current-array-print-form 'A)

(array-dimensions (make-array '#() 3 5))
=> (3 5)

(define fred (make-array '#(#f) 8 8))
(define freds-diagonal (make-shared-array fred (lambda (i) (list i i)) 8))
(array-set! freds-diagonal 'foo 3)
(array-ref fred 3 3)
=> foo
(define freds-center (make-shared-array fred (lambda (i j)
                                              (list (+ 3 i) (+ 3 j)))
                                        2 2))
(array-ref freds-center 0 0)
=> foo

(list->array 2 '#() '((1 2) (3 4)))
=> #A2((1 2) (3 4))
(list->array 0 '#() 3)
=> #A0 3

(array->list (array '#() '(2 3) 'ho 'ho 'ho 'ho 'oh 'oh))
=> ((ho ho ho) (ho oh oh))
(array->list (array '#() 0 'ho))
=> ho

(vector->array (vector 1 2 3 4) '#() 2 2)
=> #A2((1 2) (3 4))
(vector->array '#(3) '#())
=> #A0 3

(array->vector (array '#() (make-array-shape 0 2 0 2) 1 2 3 4))
=> #(1 2 3 4)
(array->vector (array '#() 0 'ho))
=> #(ho)

;;
;; See "array-lib-test.scm" in the unpacked egg for more examples.
;;
```

5 License

Index