Title

Array

Author

Aubrey Jaffer

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-47@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2003-11-11
- Draft: 2003-11-11--2004-02-11
- Final: 2004-06-14

This SRFI has been superseded by SRFI-63, "Homogeneous and Heterogeneous Arrays".

Abstract

"slib/array.scm" synthesizes array ideas from Common-Lisp and Alan Bawden with homogeneous vector ideas from SRFI-4 and SCM. The result portably integrates homogeneous and heterogeneous arrays into Scheme.

Issues

- The Scheme reports mention hardly any specific precisions for numbers. Uniform arrays necessarily must have finite precision. I followed Feeley's SRFI-4 lead for homogeneous-type precisions and names, although without the `"vector"` suffixes.

- The `array-set!` argument order is different from the same-named procedure in SRFI-25. Type dispatch on the first argument to `array-set!` could support both SRFIs simultaneously.

- The `make-array` arguments are different from the same-named procedure in SRFI-25. Type dispatch on the first argument to `make-array` could support both SRFIs simultaneously.

- An attraction of SRFI-47 procedures' argument orders is their uniformity:

```
       (make-array  proto        bound1 bound2 ...)
(make-shared-array  array mapper bound1 bound2 ...)
       (array-set!  array obj    index1 index2 ...)
 (array-in-bounds?  array        index1 index2 ...)
        (array-ref  array        index1 index2 ...)
```

- All the procedures except `make-array` and `equal?` originate from Alan Bawden's "array.scm". SRFI-47's `array-set!` argument order is that of Bawden's package. SLIB adopted "array.scm" in 1993. This form of `array-set!` has also been part of the SCM Scheme implementation since 1993.

- In response to comments, I have removed the array read-syntax from SRFI-47. The remaining package of procedures is completely portable to R4RS and R5RS Schemes as implemented by slib/array.scm.

- I have replaced `array=?` with an array-augmented version of R5RS `equal?`.

Rationale

"Array" incorporates all the homogeneous vector types from SRFI-4 "Homogeneous numeric vector datatypes", and adds complex types composed of two 32-bit or two 64-bit floating-point numbers, a uniform character array type, and a uniform boolean array type. Multi-dimensional uniform-arrays subsume homogeneous vectors as the one-dimensional case, obviating the need for SRFI-4.

- The prototype argument to `make-array` seamlessly supports as many uniform array types as an implementation provides, defaulting to arrays built on vectors (and strings).

- Byte arrays can be implemented independently and distinctly from strings.

- `make-shared-array` creates arrays which overlay a subsection of a given array; allowing reversed indexes;
- Strings and vectors are arrays; generalizing the array concept beyond that in SRFI-25.
- Strings and vectors as arrays, arrays in general, and the procedures `array-rank` and `array-dimensions` are compatible with Common-Lisp.

Implementations are required to define all of the prototype procedures. Those which the platform supports will have platform-dependent definitions; the others will be defined identically to the next larger prototype implemented; defaulting to `vector` if there are none. All implementations must support the string array type using a string.

This arrangement has platforms which support uniform array types employing them, with less capable platforms using vectors; but all working compatibly.

Conversions

- All the elements of arrays of type au8, au16, au32, au64, as8, as16, as32, or as64 are exact.
- All the elements of arrays of type ar32, ar64, ac32, or ac64 are inexact.
- The value retrieved from an exact array element will equal (=) the value stored in that element.
- Assigning a non-integer to array-type au8, au16, au32, au64, as8, as16, as32, or as64 is an error.
- Assigning a number larger than can be represented in array-type au8, au16, au32, au64, as8, as16, as32, or as64 is an error.
- Assigning a negative number to array-type au8, au16, au32, or au64 is an error.
- Assigning an inexact number to array-type au8, au16, au32, au64, as8, as16, as32, or as64 is an error.
- When assigning an exact number to array-type ar32, ar64, ac32, or ac64, the procedure may report a violation of an implementation restriction.

- Assigning a non-real number (eg. `real?` returns `#f`) to an ar64 or ar32 array is an error.
- An implementation may reduce the precision of a number assigned to an inexact array.

Specification

Function: **array?** *obj*

> Returns `#t` if the *obj* is an array, and `#f` if not.

Note: Arrays are not disjoint from other Scheme types. Strings and vectors also satisfy `array?`. A disjoint array predicate can be written:

```
(define (strict-array? obj)
  (and (array? obj) (not (string? obj)) (not (vector? obj))))
```

Function: **equal?** *obj1 obj2*

> `` `Equal?' `` recursively compares the contents of pairs, vectors, strings, and **arrays**, applying `` `eqv?' `` on other objects such as numbers and symbols. A rule of thumb is that objects are generally `` `equal?' `` if they print the same. `` `Equal?' `` may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)                             ==>  #t
(equal? '(a) '(a))                         ==>  #t
(equal? '(a (b) c)
        '(a (b) c))                        ==>  #t
(equal? "abc" "abc")                       ==>  #t
(equal? 2 2)                               ==>  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))                ==>  #t
(equal? (make-array (Au32 4) 5 3)
        (make-array (Au32 4) 5 3))         ==>  #t
(equal? (lambda (x) x)
        (lambda (y) y))                    ==>  unspecified
```

Function: **make-array** *prototype k1 k2 ...*

Creates and returns an array of type *prototype* with dimensions *k1*, *k2*, ... and filled with elements from *prototype*. *prototype* must be an array, vector, or string. The implementation-dependent type of the returned array will be the same as the type of *prototype*. except if that would be a vector or string with more than one dimension, in which case some variety of array will be returned.

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

```
(make-array '#(foo) 2 3) => #2A((foo foo foo) (foo foo foo))
```

These functions return a prototypical uniform-array enclosing the optional argument (which must be of the correct type). If the uniform-array type is supported by the implementation, then it is returned; defaulting to the next larger precision type; resorting finally to vector.

Function: **ac64** *z*

Function: **ac64**

Returns a high-precision complex uniform-array prototype.

Function: **ac32** *z*

Function: **ac32**

Returns a complex uniform-array prototype.

Function: **ar64** *x*

Function: **ar64**

Returns a high-precision real uniform-array prototype.

Function: **ar32** *x*

Function: **ar32**

Returns a real uniform-array prototype.

Function: **as64** *n*

Function: **as64**

Returns an exact signed integer uniform-array prototype with at least 64 bits of precision.

Function: **as32** *n*

Function: **as32**

Returns an exact signed integer uniform-array prototype with at least 32 bits of precision.

Function: **as16** *n*

Function: **as16**

Returns an exact signed integer uniform-array prototype with at least 16 bits of precision.

Function: **as8** *n*

Function: **as8**

Returns an exact signed integer uniform-array prototype with at least 8 bits of precision.

Function: **au64** *k*

Function: **au64**

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

Function: **au32** *k*

Function: **au32**

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

Function: **au16** *k*

Function: **au16**

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

Function: **au8** *k*

Function: **au8**

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

Function: **at1** *bool*

Function: **at1**

Returns a boolean uniform-array prototype.

Function: **make-shared-array** *array mapper k1 k2 ...*

`make-shared-array` 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. A simple example:

```
(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
```

Function: **array-rank** *obj*

Returns the number of dimensions of *obj*. If *obj* is not an array, 0 is returned.

Function: **array-dimensions** *array*

Returns a list of dimensions.

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

Function: **array-in-bounds?** *array index1 index2 ...*

Returns `#t` if its arguments would be acceptable to `array-ref`.

Function: **array-ref** *array index1 index2 ...*

Returns the (*index1*, *index2*, ...) element of *array*.

Procedure: **array-set!** *array obj index1 index2 ...*

Stores *obj* in the (*index1*, *index2*, ...) element of *array*. The value returned by `array-set!` is unspecified.

Implementation

slib/array.scm implements array procedures for R4RS or R5RS compliant Scheme implementations with *records* as implemented by slib/record.scm or SRFI-9.

Copyright

Editor: David Rush
Last modified: Sun Jan 28 13:40:35 MET 2007