NAME

array - Functional, extendible arrays.

DESCRIPTION

Functional, extendible arrays. Arrays can have fixed size, or can grow automatically as needed. A default value is used for entries that have not been explicitly set.

Arrays uses *zero*-based indexing. This is a deliberate design choice and differs from other Erlang data structures, for example, tuples.

Unless specified by the user when the array is created, the default value is the atom *undefined*. There is no difference between an unset entry and an entry that has been explicitly set to the same value as the default one (compare *reset/2*). If you need to differentiate between unset and set entries, ensure that the default value cannot be confused with the values of set entries.

The array never shrinks automatically. If an index *I* has been used to set an entry successfully, all indices in the range [0,*I*] stay accessible unless the array size is explicitly changed by calling *resize/2*.

Examples:

Create a fixed-size array with entries 0-9 set to *undefined*:

```
A0 = array:new(10).
10 = array:size(A0).
```

Create an extendible array and set entry 17 to *true*, causing the array to grow automatically:

```
A1 = array:set(17, true, array:new()).
18 = array:size(A1).
```

Read back a stored value:

```
true = array:get(17, A1).
```

Accessing an unset entry returns default value:

```
undefined = array:get(3, A1)
```

Accessing an entry beyond the last set entry also returns the default value, if the array does not have fixed size:

```
undefined = array:get(18, A1).
```

"Sparse" functions ignore default-valued entries:

```
A2 = array:set(4, false, A1).
[{4, false}, {17, true}] = array:sparse_to_orddict(A2).
```

An extendible array can be made fixed-size later:

```
A3 = array:fix(A2).
```

A fixed-size array does not grow automatically and does not allow accesses beyond the last set entry:

```
{'EXIT',{badarg,_}} = (catch array:set(18, true, A3)).
{'EXIT',{badarg,_}} = (catch array:get(18, A3)).
```

DATA TYPES

array(Type)

A functional, extendible array. The representation is not documented and is subject to change without notice. Notice that arrays cannot be directly compared for equality.

array() = array(term())

array_indx() = integer() >= 0

array_opts() = array_opt() | [array_opt()]

array_opt() =
{fixed, boolean()} |

fixed |
{default, Type :: term()} |
{size, N :: integer() >= 0} |
(N :: integer() >= 0)

indx_pairs(Type) = [indx_pair(Type)]

indx_pair(Type) = {Index :: array_indx(), Type}

EXPORTS

default(Array :: array(Type)) -> Value :: Type

Gets the value used for uninitialized entries.

See also *new/2*.

fix(Array :: array(Type)) -> array(Type)

Fixes the array size. This prevents it from growing automatically upon insertion.

See also *set/3* and *relax/1*.

foldl(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:

Function =
fun((Index :: array_indx(), Value :: Type, Acc :: A) -> B)

Folds the array elements using the specified function and initial accumulator value. The elements are visited in order from the lowest index to the highest. If *Function* is not a function, the call fails with reason *badarg*.

See also *foldr/3*, *map/2*, *sparse_foldl/3*.

foldr(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:

```
Function =
  fun((Index :: array_indx(), Value :: Type, Acc :: A) -> B)
```

Folds the array elements right-to-left using the specified function and initial accumulator value. The elements are visited in order from the highest index to the lowest. If *Function* is not a function, the call fails with reason *badarg*.

See also *foldl/3*, *map/2*.

from_list(List :: [Value :: Type]) -> array(Type)

Equivalent to *from_list(List, undefined)*.

from_list(List :: [Value :: Type], Default :: term()) ->
array(Type)

Converts a list to an extendible array. *Default* is used as the value for uninitialized entries of the array. If *List* is not a proper list, the call fails with reason *badarg*.

See also *new/2*, *to_list/1*.

from_orddict(Orddict :: indx_pairs(Value :: Type)) -> array(Type)

Equivalent to *from_orddict(Orddict, undefined)*.

from_orddict(Orddict :: indx_pairs(Value :: Type),
Default :: Type) ->
array(Type)

Converts an ordered list of pairs *{Index, Value}* to a corresponding extendible array. *Default* is used as the value for uninitialized entries of the array. If *Orddict* is not a proper, ordered list of pairs whose first elements are non-negative integers, the call fails with reason *badarg*.

See also *new/2*, *to_orddict/1*.

get(I :: array_indx(), Array :: array(Type)) -> Value :: Type

Gets the value of entry *I*. If *I* is not a non-negative integer, or if the array has fixed size and *I* is larger than the maximum index, the call fails with reason *badarg*.

If the array does not have fixed size, the default value for any index *I* greater than *size(Array)-1* is returned.

See also *set/3*.

is_array(X :: term()) -> boolean()

Returns *true* if *X* is an array, otherwise *false*. Notice that the check is only shallow, as there is no guarantee that *X* is a well-formed array representation even if this function returns *true*.

is_fix(Array :: array()) -> boolean()

Checks if the array has fixed size. Returns *true* if the array is fixed, otherwise *false*.

See also *fix/1*.

map(Function, Array :: array(Type1)) -> array(Type2)

Types:

Function = fun((Index :: array_indx(), Type1) -> Type2)

Maps the specified function onto each array element. The elements are visited in order from the lowest index to the highest. If *Function* is not a function, the call fails with reason *badarg*.

See also *foldl/3*, *foldr/3*, *sparse_map/2*.

new() -> array()

Creates a new, extendible array with initial size zero.

See also *new/1*, *new/2*.

new(Options :: array_opts()) -> array()

Creates a new array according to the specified options. By default, the array is extendible and has initial size zero. Array indices start at *0*.

Options is a single term or a list of terms, selected from the following:

N::integer() >= 0 or *{size, N::integer() >= 0}*:
Specifies the initial array size; this also implies *{fixed, true}*. If *N* is not a non-negative integer, the call fails with reason *badarg*.

fixed or *{fixed, true}*:
Creates a fixed-size array. See also *fix/1*.

{fixed, false}:
Creates an extendible (non-fixed-size) array.

{default, Value}:
Sets the default value for the array to *Value*.

Options are processed in the order they occur in the list, that is, later options have higher precedence.

The default value is used as the value of uninitialized entries, and cannot be changed once the array has been created.

Examples:

array:new(100)

creates a fixed-size array of size 100.

array:new({default,0})

creates an empty, extendible array whose default value is *0*.

array:new([{size,10},{fixed,false},{default,-1}])

creates an extendible array with initial size 10 whose default value is *-1*.

See also *fix/1*, *from_list/2*, *get/2*, *new/0*, *new/2*, *set/3*.

new(Size :: integer() >= 0, Options :: array_opts()) -> array()

Creates a new array according to the specified size and options. If *Size* is not a non-negative integer, the call fails with reason *badarg*. By default, the array has fixed size. Notice that any size specifications in *Options* override parameter *Size*.

If *Options* is a list, this is equivalent to *new([{size, Size} | Options])*, otherwise it is equivalent to *new([{size, Size} | [Options]])*. However, using this function directly is more efficient.

Example:

array:new(100, {default,0})

creates a fixed-size array of size 100, whose default value is *0*.

See also *new/1*.

relax(Array :: array(Type)) -> array(Type)

Makes the array resizable. (Reverses the effects of *fix/1*.)

See also *fix/1*.

reset(I :: array_indx(), Array :: array(Type)) -> array(Type)

Resets entry *I* to the default value for the array. If the value of entry *I* is the default value, the array is returned unchanged. Reset never changes the array size. Shrinking can be done explicitly by calling *resize/2*.

If *I* is not a non-negative integer, or if the array has fixed size and *I* is larger than the maximum

index, the call fails with reason *badarg*; compare *set/3*

See also *new/2*, *set/3*.

resize(Array :: array(Type)) -> array(Type)

Changes the array size to that reported by *sparse_size/1*. If the specified array has fixed size, also the resulting array has fixed size.

See also *resize/2*, *sparse_size/1*.

resize(Size :: integer() >= 0, Array :: array(Type)) ->
array(Type)

Change the array size. If *Size* is not a non-negative integer, the call fails with reason *badarg*. If the specified array has fixed size, also the resulting array has fixed size.

set(I :: array_indx(), Value :: Type, Array :: array(Type)) ->
array(Type)

Sets entry *I* of the array to *Value*. If *I* is not a non-negative integer, or if the array has fixed size and *I* is larger than the maximum index, the call fails with reason *badarg*.

If the array does not have fixed size, and *I* is greater than *size(Array)-1*, the array grows to size *I+1*.

See also *get/2*, *reset/2*.

size(Array :: array()) -> integer() >= 0

Gets the number of entries in the array. Entries are numbered from *0* to *size(Array)-1*. Hence, this is also the index of the first entry that is guaranteed to not have been previously set.

See also *set/3*, *sparse_size/1*.

sparse_foldl(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:

Function =
fun((Index :: array_indx(), Value :: Type, Acc :: A) -> B)

Folds the array elements using the specified function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the lowest index to the highest. If *Function* is not a function, the call fails with reason *badarg*.

See also *foldl/3*, *sparse_foldr/3*.

sparse_foldr(Function, InitialAcc :: A, Array :: array(Type)) -> B

Types:

Function =
fun((Index :: array_indx(), Value :: Type, Acc :: A) -> B)

Folds the array elements right-to-left using the specified function and initial accumulator value, skipping default-valued entries. The elements are visited in order from the highest index to the lowest. If *Function* is not a function, the call fails with reason *badarg*.

See also *foldr/3*, *sparse_foldl/3*.

sparse_map(Function, Array :: array(Type1)) -> array(Type2)

Types:

Function = fun((Index :: array_indx(), Type1) -> Type2)

Maps the specified function onto each array element, skipping default-valued entries. The elements are visited in order from the lowest index to the highest. If *Function* is not a function, the call fails with reason *badarg*.

See also *map/2*.

sparse_size(Array :: array()) -> integer() >= 0

Gets the number of entries in the array up until the last non-default-valued entry. That is, returns *I+1* if *I* is the last non-default-valued entry in the array, or zero if no such entry exists.

See also *resize/1*, *size/1*.

sparse_to_list(Array :: array(Type)) -> [Value :: Type]

Converts the array to a list, skipping default-valued entries.

See also *to_list/1*.

sparse_to_orddict(Array :: array(Type)) ->
indx_pairs(Value :: Type)

Converts the array to an ordered list of pairs *{Index, Value}*, skipping default-valued entries.

See also *to_orddict/1*.

to_list(Array :: array(Type)) -> [Value :: Type]

Converts the array to a list.

See also *from_list/2*, *sparse_to_list/1*.

to_orddict(Array :: array(Type)) -> indx_pairs(Value :: Type)

Converts the array to an ordered list of pairs *{Index, Value}*.

See also *from_orddict/2*, *sparse_to_orddict/1*.