Contents

1   Module `Strat` : Interface to strategies

```
module type T =
  sig
```

> **type t**
>
> > The abstract type of strategies.
>
> **val default : t**
>
> > Default strategy of this strategy implementation.
>
> **val grow : t -> int -> int**
>
> > **grow strat new_len**
> > **Returns** the new real length of some contiguous datastructure using strategy **strat**
> > given new virtual length **new_len**. The user should then use this new real length to
> > resize the datastructure.
> > Be careful, the new (real) length **must** be larger than the new virtual length, otherwise
> > your program will crash!
>
> **val shrink : t -> real_len:int -> new_len:int -> int**
>
> > **shrink strat ˜real_len ˜new_len**
> > **Returns** the new real length of a resizable datastructure given its current real length
> > **real_len** and its required new virtual length **new_len** wrt. strategy **strat**. The user
> > should then use this new real length to resize the datastructure. If **-1** is returned, it is
> > not necessary to resize.
> > Be careful, the new (real) length **must** be larger than the new (virtual) length,
> > otherwise your program may crash!

```
  end
```

2 Module `Pres_intf` : Interface to parameterized resizable arrays

```
module type T =
  sig
```

Signatures and types

```
module Strategy :
```

```
Strat.T
```

> Module implementing the reallocation strategy

```
type strategy = Strategy.t
```

> Type of reallocation strategy

```
type 'a t
```

> Type of parameterized resizable arrays

Index and length information

```
val length : 'a t -> int
```

> `length ra`
> **Returns** (virtual) length of resizable array `ra` excluding the reserved space.

```
val lix : 'a t -> int
```

> `lix ra`
> **Returns** (virtual) last index of resizable array `ra` excluding the reserved space.

```
val real_length : 'a t -> int
```

> `real_length ra`
> **Returns** (real) length of resizable array `ra` including the reserved space.

```
val real_lix : 'a t -> int
```

> `real_lix ra`
> **Returns** (real) last index of resizable array `ra` including the reserved space.

Getting and setting

```
val get : 'a t -> int -> 'a
```

> `get ra n`
> **Raises** `Invalid_argument` if index out of bounds.
> **Returns** the nth element of `ra`.

```
val set : 'a t -> int -> 'a -> unit
```

```
set ra n
```
sets the nth element of `ra`.
Raises `Invalid_argument` if index out of bounds.

Creation of resizable arrays

```
val sempty : strategy -> 'a t
```

```
sempty s
```
Returns an empty resizable array using strategy `s`.

```
val empty : unit -> 'a t
```

`empty ()` same as `sempty` but uses default strategy.

```
val screate : strategy -> int -> 'a -> 'a t
```

```
screate s n el
```
Returns a resizable array of length `n` containing element `el` only using strategy `s`.

```
val create : int -> 'a -> 'a t
```

`create n el` same as `screate` but uses default strategy.

```
val smake : strategy -> int -> 'a -> 'a t
```

`smake s n el` same as `screate`.

```
val make : int -> 'a -> 'a t
```

`make n el` same as `create`.

```
val sinit : strategy -> int -> (int -> 'a) -> 'a t
```

```
sinit s n f
```
Returns an array of length `n` containing elements that were created by applying function `f` to the index, using strategy `s`.

```
val init : int -> (int -> 'a) -> 'a t
```

`init n f` sames as `sinit` but uses default strategy.

Strategy handling

```
val get_strategy : 'a t -> strategy
```

```
get_strategy ra
```
Returns the reallocation strategy used by resizable array `ra`.

```
val set_strategy : 'a t -> strategy -> unit
```

`set_strategy ra s` sets the reallocation strategy of resizable array `ra` to `s`, possibly causing an immediate reallocation.

```
val put_strategy : 'a t -> strategy -> unit
```

>   put_strategy ra s sets the reallocation strategy of resizable array ra to s.
>   Reallocation is only done at later changes in size.

```
val enforce_strategy : 'a t -> unit
```

>   enforce_strategy ra forces a reallocation if necessary (e.g. after a put_strategy.

Matrix functions
```
val make_matrix : int -> int -> 'a -> 'a t t
```

>   make_matrix sx sy el creates a (resizable) matrix of dimensions sx and sy containing
>   element el only. Both dimensions are controlled by the default strategy.

Copying, blitting and range extraction
```
val copy : 'a t -> 'a t
```

>   copy ra
>   **Returns** a copy of resizable array ra. The two arrays share the same strategy!

```
val sub : 'a t -> int -> int -> 'a t
```

>   sub ra ofs len
>   **Raises** Invalid_argument if parameters do not denote a correct subarray.
>   **Returns** a resizable subarray of length len from resizable array ra starting at offset
>   ofs using the default strategy.

```
val fill : 'a t -> int -> int -> 'a -> unit
```

>   fill ra ofs len el fills resizable array ra from offset ofs with len elements el,
>   possibly adding elements at the end. Raises Invalid_argument if offset ofs is larger
>   than the length of the array.

```
val blit : 'a t -> int -> 'a t -> int -> int -> unit
```

>   blit ra1 ofs1 ra2 ofs2 len blits resizable array ra1 onto ra2 reading len elements
>   from offset ofs1 and writing them to ofs2, possibly adding elements at the end of ra2.
>   Raises Invalid_argument if ofs1 and len do not designate a valid subarray of ra1 or if
>   ofs2 is larger than the length of ra2.

Combining resizable arrays
```
val append : 'a t -> 'a t -> 'a t
```

>   append ra1 ra2
>   **Returns** a new resizable array using the default strategy and copying ra1 and ra2 in
>   this order onto it.

```
val concat : 'a t list -> 'a t
```

```
concat l
```
Returns a new resizable array using the default strategy and copying all resizable arrays in `l` in their respective order onto it.

Adding and removing elements
```
val add_one : 'a t -> 'a -> unit
```

`add_one ra el` adds element `el` to resizable array `ra`, possibly causing a reallocation.

```
val remove_one : 'a t -> unit
```

`remove_one ra` removes the last element of resizable array `ra`, possibly causing a reallocation.
Raises `Failure` if the array is empty.

```
val remove_n : 'a t -> int -> unit
```

`remove_n ra n` removes the last n elements of resizable array `ra`, possibly causing a reallocation.
Raises `Invalid_arg` if there are not enough elements or `n < 0`.

```
val remove_range : 'a t -> int -> int -> unit
```

`remove_range ra ofs len` removes `len` elements from resizable array `ra` starting at `ofs` and possibly causing a reallocation.
Raises `Invalid_argument` if range is invalid.

```
val clear : 'a t -> unit
```

`clear ra` removes all elements from resizable array `ra`, possibly causing a reallocation.

Swapping
```
val swap : 'a t -> int -> int -> unit
```

`swap ra n m` swaps elements at indices `n` and `m`.
Raises `Invalid_argument` if any index is out of range.

```
val swap_in_last : 'a t -> int -> unit
```

`swap_in_last ra n` swaps the last element with the one at position `n`.
Raises `Invalid_argument` if index `n` is out of range.

Array conversions
```
val to_array : 'a t -> 'a array
```

`to_array ra` converts a resizable array to a standard one.

```
val sof_array : strategy -> 'a array -> 'a t
```

`sof_array s ar` converts a standard array to a resizable one, using strategy `s`.

`val of_array : 'a array -> 'a t`

    `of_array ar` converts a standard array to a resizable one using the default strategy.

List conversions

`val to_list : 'a t -> 'a list`

    `to_list ra` converts resizable array `ra` to a list.

`val sof_list : strategy -> 'a list -> 'a t`

    `sof_list s l` creates a resizable array using strategy `s` and the elements in list `l`.

`val of_list : 'a list -> 'a t`

    `of_list l` creates a resizable array using the default strategy and the elements in list `l`.

Iterators

`val iter : ('a -> unit) -> 'a t -> unit`

    `iter f ra` applies the unit-function `f` to each element in resizable array `ra`.

`val map : ('a -> 'b) -> 'a t -> 'b t`

    `map f ra`
    Returns a resizable array using the strategy of `ra` and mapping each element in `ra` to its corresponding position in the new array using function `f`.

`val iteri : (int -> 'a -> unit) -> 'a t -> unit`

    `iteri f ra` applies the unit-function `f` to each index and element in resizable array `ra`.

`val mapi : (int -> 'a -> 'b) -> 'a t -> 'b t`

    `mapi f ra`
    Returns a resizable array using the strategy of `ra` and mapping each element in `ra` to its corresponding position in the new array using function `f` and the index position.

`val fold_left : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b`

    `fold_left f a ra` left-folds values in resizable array `ra` using function `f` and start accumulator `a`.

`val fold_right : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

    `fold_right f a ra` right-folds values in resizable array `ra` using function `f` and start accumulator `a`.

Scanning of resizable arrays

```
val for_all : ('a -> bool) -> 'a t -> bool
```

> for_all p ra
>
> **Returns** true if all elements in resizable array ra satisfy the predicate p, false otherwise.

```
val exists : ('a -> bool) -> 'a t -> bool
```

> exists p ra
>
> **Returns** true if at least one element in resizable array ra satisfies the predicate p, false otherwise.

```
val mem : 'a -> 'a t -> bool
```

> mem el ra
>
> **Returns** true if element el is logically equal to any element in resizable array ra, false otherwise.

```
val memq : 'a -> 'a t -> bool
```

> memq el ra
>
> **Returns** true if element el is physically equal to any element in resizable array ra, false otherwise.

```
val pos : 'a -> 'a t -> int option
```

> pos el ra
>
> **Returns** Some index if el is logically equal to the element at index in ra, None otherwise. index is the index of the first element that matches.

```
val posq : 'a -> 'a t -> int option
```

> posq el ra
>
> **Returns** Some index if el is physically equal to the element at index in ra, None otherwise. index is the index of the first element that matches.

Searching of resizable arrays

```
val find : ('a -> bool) -> 'a t -> 'a
```

> find p ra
>
> **Raises** Not_found if there is no such element.
>
> **Returns** the first element in resizable array ra that satisfies predicate p.

```
val find_index : ('a -> bool) -> 'a t -> int -> int
```

> find_index p ra pos
>
> **Raises**

- **Not_found** if there is no such element or if **pos** is larger than the highest index.
- **Invalid_argument** if **pos** is negative.

Returns the index of the first element that satisfies predicate **p** in resizable array **ra**, starting search at index **pos**.

```
val filter : ('a -> bool) -> 'a t -> 'a t
```

filter p ra
Returns a new resizable array by filtering out all elements in **ra** that satisfy predicate **p** using the same strategy as **ra**.

```
val find_all : ('a -> bool) -> 'a t -> 'a t
```

find_all p ra is the same as **filter**

```
val filter_in_place : ('a -> bool) -> 'a t -> unit
```

filter_in_place p ra as **filter**, but filters in place.

```
val partition : ('a -> bool) -> 'a t -> 'a t * 'a t
```

partition p ra
Returns a pair of resizable arrays, the left part containing only elements of **ra** that satisfy predicate **p**, the right one only those that do not satisfy it. Both returned arrays are created using the strategy of **ra**.

UNSAFE STUFF - USE WITH CAUTION!

```
val unsafe_get : 'a t -> int -> 'a
val unsafe_set : 'a t -> int -> 'a -> unit
val unsafe_sub : 'a t -> int -> int -> 'a t
val unsafe_fill : 'a t -> int -> int -> 'a -> unit
val unsafe_blit : 'a t -> int -> 'a t -> int -> int -> unit
val unsafe_remove_one : 'a t -> unit
val unsafe_remove_n : 'a t -> int -> unit
val unsafe_swap : 'a t -> int -> int -> unit
val unsafe_swap_in_last : 'a t -> int -> unit
```
end

3 Module Nopres_intf : Interfaces to unparameterized resizable arrays and buffers

```
module type T =
  sig
```

Signatures and types

```
module Strategy :
Strat.T
```

> Module implementing the reallocation strategy

```
type strategy = Strategy.t
```

> Type of reallocation strategy

```
type t
```

> Type of resizable arrays

```
type el
```

> Type of the elements in the resizable array

Index and length information

```
val length : t -> int
```

> `length ra`
> **Returns** (virtual) length of resizable array `ra` excluding the reserved space.

```
val lix : t -> int
```

> `lix ra`
> **Returns** (virtual) last index of resizable array `ra` excluding the reserved space.

```
val real_length : t -> int
```

> `real_length ra`
> **Returns** (real) length of resizable array `ra` including the reserved space.

```
val real_lix : t -> int
```

> `real_lix ra`
> **Returns** (real) last index of resizable array `ra` including the reserved space.

Getting and setting

```
val get : t -> int -> el
```

> `get ra n`
> **Raises** `Invalid_argument` if index out of bounds.
> **Returns** the nth element of `ra`.

```
val set : t -> int -> el -> unit
```

> `set ra n` sets the nth element of `ra`.
> **Raises** `Invalid_argument` if index out of bounds.

Creation of resizable arrays

```
val sempty : strategy -> t
```

> sempty s
> **Returns** an empty resizable array using strategy s.

```
val empty : unit -> t
```

> empty () same as sempty but uses default strategy.

```
val screate : strategy -> int -> t
```

> screate s n
> **Returns** a resizable array with strategy s containing n arbitrary elements.
> *Attention: the contents is* **not** *specified!*

```
val create : int -> t
```

> create n same as screate but uses default strategy.

```
val smake : strategy -> int -> el -> t
```

> smake s n el
> **Returns** a resizable array of length n containing element el only using strategy s.

```
val make : int -> el -> t
```

> make n el same as smake but uses default strategy.

```
val sinit : strategy -> int -> (int -> el) -> t
```

> sinit s n f
> **Returns** an array of length n containing elements that were created by applying
> function f to the index, using strategy s.

```
val init : int -> (int -> el) -> t
```

> init n f sames as sinit but uses default strategy.

Strategy handling

```
val get_strategy : t -> strategy
```

> get_strategy ra
> **Returns** the reallocation strategy used by resizable array ra.

```
val set_strategy : t -> strategy -> unit
```

> set_strategy ra s sets the reallocation strategy of resizable array ra to s, possibly
> causing an immediate reallocation.

```
val put_strategy : t -> strategy -> unit
```

> `put_strategy ra s` sets the reallocation strategy of resizable array `ra` to `s`.
> Reallocation is only done at later changes in size.

```
val enforce_strategy : t -> unit
```

> `enforce_strategy ra` forces a reallocation if necessary (e.g. after a `put_strategy`.

Copying, blitting and range extraction

```
val copy : t -> t
```

> `copy ra`
> **Returns** a copy of resizable array `ra`. The two arrays share the same strategy!

```
val sub : t -> int -> int -> t
```

> `sub ra ofs len`
> **Raises** `Invalid_argument` if parameters do not denote a correct subarray.
> **Returns** a resizable subarray of length `len` from resizable array `ra` starting at offset
> `ofs` using the default strategy.

```
val fill : t -> int -> int -> el -> unit
```

> `fill ra ofs len el` fills resizable array `ra` from offset `ofs` with `len` elements `el`,
> possibly adding elements at the end. Raises `Invalid_argument` if offset `ofs` is larger
> than the length of the array.

```
val blit : t -> int -> t -> int -> int -> unit
```

> `blit ra1 ofs1 ra2 ofs2 len` blits resizable array `ra1` onto `ra2` reading `len` elements
> from offset `ofs1` and writing them to `ofs2`, possibly adding elements at the end of ra2.
> Raises `Invalid_argument` if `ofs1` and `len` do not designate a valid subarray of `ra1` or if
> `ofs2` is larger than the length of `ra2`.

Combining resizable arrays

```
val append : t -> t -> t
```

> `append ra1 ra2`
> **Returns** a new resizable array using the default strategy and copying `ra1` and `ra2` in
> this order onto it.

```
val concat : t list -> t
```

> `concat l`
> **Returns** a new resizable array using the default strategy and copying all resizable
> arrays in `l` in their respective order onto it.

Adding and removing elements

```
val add_one : t -> el -> unit
```

`add_one ra el` adds element `el` to resizable array `ra`, possibly causing a reallocation.

`val remove_one : t -> unit`

    `remove_one ra` removes the last element of resizable array `ra`, possibly causing a reallocation.

    Raises `Failure` if the array is empty.

`val remove_n : t -> int -> unit`

    `remove_n ra n` removes the last n elements of resizable array `ra`, possibly causing a reallocation.

    Raises `Invalid_arg` if there are not enough elements or `n < 0`.

`val remove_range : t -> int -> int -> unit`

    `remove_range ra ofs len` removes `len` elements from resizable array `ra` starting at `ofs` and possibly causing a reallocation.

    Raises `Invalid_argument` if range is invalid.

`val clear : t -> unit`

    `clear ra` removes all elements from resizable array `ra`, possibly causing a reallocation.

Swapping

`val swap : t -> int -> int -> unit`

    `swap ra n m` swaps elements at indices `n` and `m`.

    Raises `Invalid_argument` if any index is out of range.

`val swap_in_last : t -> int -> unit`

    `swap_in_last ra n` swaps the last element with the one at position `n`.

    Raises `Invalid_argument` if index `n` is out of range.

Array conversions

`val to_array : t -> el array`

    `to_array ra` converts a resizable array to a standard one.

`val sof_array : strategy -> el array -> t`

    `sof_array s ar` converts a standard array to a resizable one, using strategy `s`.

`val of_array : el array -> t`

    `of_array ar` converts a standard array to a resizable one using the default strategy.

List conversions

`val to_list : t -> el list`

`to_list ra` converts resizable array `ra` to a list.

`val sof_list : strategy -> el list -> t`

   `sof_list s l` creates a resizable array using strategy `s` and the elements in list `l`.

`val of_list : el list -> t`

   `of_list l` creates a resizable array using the default strategy and the elements in list `l`.

Iterators

`val iter : (el -> unit) -> t -> unit`

   `iter f ra` applies the unit-function `f` to each element in resizable array `ra`.

`val map : (el -> el) -> t -> t`

   `map f ra`
   Returns a resizable array using the strategy of `ra` and mapping each element in `ra` to its corresponding position in the new array using function `f`.

`val iteri : (int -> el -> unit) -> t -> unit`

   `iteri f ra` applies the unit-function `f` to each index and element in resizable array `ra`.

`val mapi : (int -> el -> el) ->`
  `t -> t`

   `mapi f ra`
   Returns a resizable array using the strategy of `ra` and mapping each element in `ra` to its corresponding position in the new array using function `f` and the index position.

`val fold_left : ('a -> el -> 'a) -> 'a -> t -> 'a`

   `fold_left f a ra` left-folds values in resizable array `ra` using function `f` and start accumulator `a`.

`val fold_right : (el -> 'a -> 'a) -> t -> 'a -> 'a`

   `fold_right f a ra` right-folds values in resizable array `ra` using function `f` and start accumulator `a`.

Scanning of resizable arrays

`val for_all : (el -> bool) -> t -> bool`

   `for_all p ra`
   Returns `true` if all elements in resizable array `ra` satisfy the predicate `p`, `false` otherwise.

`val exists : (el -> bool) -> t -> bool`

```
exists p ra
```

Returns `true` if at least one element in resizable array `ra` satisfies the predicate `p`, `false` otherwise.

```
val mem : el -> t -> bool
```

```
mem el ra
```

Returns `true` if element `el` is logically equal to any element in resizable array `ra`, `false` otherwise.

```
val memq : el -> t -> bool
```

```
memq el ra
```

Returns `true` if element `el` is physically equal to any element in resizable array `ra`, `false` otherwise.

```
val pos : el -> t -> int option
```

```
pos el ra
```

Returns `Some index` if `el` is logically equal to the element at `index` in `ra`, `None` otherwise. `index` is the index of the first element that matches.

```
val posq : el -> t -> int option
```

```
posq el ra
```

Returns `Some index` if `el` is physically equal to the element at `index` in `ra`, `None` otherwise. `index` is the index of the first element that matches.

Searching of resizable arrays

```
val find : (el -> bool) -> t -> el
```

```
find p ra
```

Raises `Not_found` if there is no such element.

Returns the first element in resizable array `ra` that satisfies predicate `p`.

```
val find_index : (el -> bool) -> t -> int -> int
```

```
find_index p ra pos
```

Raises

- `Not_found` if there is no such element or if `pos` is larger than the highest index.
- `Invalid_argument` if `pos` is negative.

Returns the index of the first element that satisfies predicate `p` in resizable array `ra`, starting search at index `pos`.

```
val filter : (el -> bool) -> t -> t
```

```
filter p ra
```

Returns a new resizable array by filtering out all elements in `ra` that satisfy predicate `p` using the same strategy as `ra`.

```
val find_all : (el -> bool) -> t -> t
```

    `find_all p ra` is the same as `filter`

```
val filter_in_place : (el -> bool) -> t -> unit
```

    `filter_in_place p ra` as `filter`, but filters in place.

```
val partition : (el -> bool) ->
  t -> t * t
```

    `partition p ra`

    Returns a pair of resizable arrays, the left part containing only elements of `ra` that satisfy predicate `p`, the right one only those that do not satisfy it. Both returned arrays are created using the strategy of `ra`.

UNSAFE STUFF - USE WITH CAUTION!

```
val unsafe_get : t -> int -> el
val unsafe_set : t -> int -> el -> unit
val unsafe_sub : t -> int -> int -> t
val unsafe_fill : t -> int -> int -> el -> unit
val unsafe_blit : t -> int -> t -> int -> int -> unit
val unsafe_remove_one : t -> unit
val unsafe_remove_n : t -> int -> unit
val unsafe_swap : t -> int -> int -> unit
val unsafe_swap_in_last : t -> int -> unit
```

  `end`

    Interface to unparameterized resizable arrays

```
module type Buffer =
  sig
```

    `include Nopres_intf.T`

    Includes all functions that exist in non-parameterized arrays.

    String conversions

```
val sof_string : strategy -> string -> t
```

    `sof_string s ar` converts a string to a resizable buffer using strategy `s`.

```
val of_string : string -> t
```

    `of_string ar` converts a string to a resizable buffer using the default strategy.

    Functions found in the standard `Buffer`-module

    Note that the function `create n` ignores the parameter `n` and uses the default strategy instead. You can supply a different strategy with `creates s n` as described above.

```
val contents : t -> string
```

```
        contents b
```
Returns a copy of the current contents of the buffer b.

```
val reset : t -> unit
```

reset b just clears the buffer, possibly resizing it.

```
val add_char : t -> char -> unit
```

add_char b c appends the character c at the end of the buffer b.

```
val add_string : t -> string -> unit
```

add_string b s appends the string s at the end of the buffer b.

```
val add_substring : t -> string -> int -> int -> unit
```

add_substring b s ofs len takes len characters from offset ofs in string s and appends them at the end of the buffer b.

```
val add_buffer : t -> t -> unit
```

add_buffer b1 b2 appends the current contents of buffer b2 at the end of buffer b1. b2 is not modified.

```
val add_channel : t -> Pervasives.in_channel -> int -> unit
```

add_channel b ic n reads exactly n character from the input channel ic and stores them at the end of buffer b.

Raises End_of_file if the channel contains fewer than n characters.

```
val output_buffer : Pervasives.out_channel -> t -> unit
```

output_buffer oc b writes the current contents of buffer b on the output channel oc.

Additional buffer functions

```
val add_full_channel : t -> Pervasives.in_channel -> unit
val add_full_channel_f :
  t -> Pervasives.in_channel -> int -> (int -> int) -> unit
```

```
end
```

Extended interface to buffers (resizable strings)

4 Module `Weak_intf` : Interface to weak resizable arrays

```
module type T =
  sig
```

Signatures and types

`module Strategy :`

`Strat.T`

> Module implementing the reallocation strategy

`type strategy = Strategy.t`

> Type of reallocation strategy

`type 'a t`

> Type of parameterized resizable arrays

Index and length information

`val length : 'a t -> int`

> `length ra`
> **Returns** (virtual) length of resizable array `ra` excluding the reserved space.

`val lix : 'a t -> int`

> `lix ra`
> **Returns** (virtual) last index of resizable array `ra` excluding the reserved space.

`val real_length : 'a t -> int`

> `real_length ra`
> **Returns** (real) length of resizable array `ra` including the reserved space.

`val real_lix : 'a t -> int`

> `real_lix ra`
> **Returns** (real) last index of resizable array `ra` including the reserved space.

Getting, setting and checking

`val get : 'a t -> int -> 'a option`

> `get ra n`
> **Raises** `Invalid_argument` if index out of bounds.
> **Returns** the nth element of `ra`.

`val get_copy : 'a t -> int -> 'a option`

```
get_copy ra n
```
see documentation of module `Weak` in the standard distribution.

```
val check : 'a t -> int -> bool
```

```
check ra n
```
Returns `true` if the nth cell of `ra` is full, `false` if it is empty. Note that even if `check` `ar n` returns `true`, a subsequent `Weak_intf.T.get`[4] `ar n` can return `None`.

```
val set : 'a t -> int -> 'a option -> unit
```

```
set ra n
```
sets the nth element of `ra`.
Raises `Invalid_argument` if index out of bounds.

Creation of resizable arrays

```
val sempty : strategy -> 'a t
```

```
sempty s
```
Returns an empty resizable array using strategy `s`.

```
val empty : unit -> 'a t
```

```
empty ()
```
same as `sempty` but uses default strategy.

```
val screate : strategy -> int -> 'a t
```

```
screate s n el
```
Returns a resizable array of length `n` using strategy `s`.

```
val create : int -> 'a t
```

```
create n
```
same as `screate` but uses default strategy.

```
val sinit : strategy -> int -> (int -> 'a option) -> 'a t
```

```
sinit s n f
```
Returns an array of length `n` containing elements that were created by applying function `f` to the index, using strategy `s`.

```
val init : int -> (int -> 'a option) -> 'a t
```

```
init n f
```
sames as `sinit` but uses default strategy.

Strategy handling

```
val get_strategy : 'a t -> strategy
```

```
get_strategy ra
```
Returns the reallocation strategy used by resizable array `ra`.

```
val set_strategy : 'a t -> strategy -> unit
```

`set_strategy ra s` sets the reallocation strategy of resizable array `ra` to `s`, possibly causing an immediate reallocation.

`val put_strategy : 'a t -> strategy -> unit`

`put_strategy ra s` sets the reallocation strategy of resizable array `ra` to `s`. Reallocation is only done at later changes in size.

`val enforce_strategy : 'a t -> unit`

`enforce_strategy ra` forces a reallocation if necessary (e.g. after a `put_strategy`.

Copying, blitting and range extraction

`val copy : 'a t -> 'a t`

`copy ra`
Returns a copy of resizable array `ra`. The two arrays share the same strategy!

`val sub : 'a t -> int -> int -> 'a t`

`sub ra ofs len`
Raises `Invalid_argument` if parameters do not denote a correct subarray.
Returns a resizable subarray of length `len` from resizable array `ra` starting at offset `ofs` using the default strategy.

`val fill : 'a t -> int -> int -> 'a option -> unit`

`fill ra ofs len el` fills resizable array `ra` from offset `ofs` with `len` elements `el`, possibly adding elements at the end. Raises `Invalid_argument` if offset `ofs` is larger than the length of the array.

`val blit : 'a t -> int -> 'a t -> int -> int -> unit`

`blit ra1 ofs1 ra2 ofs2 len` blits resizable array `ra1` onto `ra2` reading `len` elements from offset `ofs1` and writing them to `ofs2`, possibly adding elements at the end of ra2. Raises `Invalid_argument` if `ofs1` and `len` do not designate a valid subarray of `ra1` or if `ofs2` is larger than the length of `ra2`.

Combining resizable arrays

`val append : 'a t -> 'a t -> 'a t`

`append ra1 ra2`
Returns a new resizable array using the default strategy and copying `ra1` and `ra2` in this order onto it.

`val concat : 'a t list -> 'a t`

`concat l`
Returns a new resizable array using the default strategy and copying all resizable arrays in `l` in their respective order onto it.

Adding and removing elements

```
val add_one : 'a t -> 'a option -> unit
```

    `add_one ra el` adds element `el` to resizable array `ra`, possibly causing a reallocation.

```
val remove_one : 'a t -> unit
```

    `remove_one ra` removes the last element of resizable array `ra`, possibly causing a reallocation.
    Raises `Failure` if the array is empty.

```
val remove_n : 'a t -> int -> unit
```

    `remove_n ra n` removes the last n elements of resizable array `ra`, possibly causing a reallocation.
    Raises `Invalid_arg` if there are not enough elements or `n < 0`.

```
val remove_range : 'a t -> int -> int -> unit
```

    `remove_range ra ofs len` removes `len` elements from resizable array `ra` starting at `ofs` and possibly causing a reallocation.
    Raises `Invalid_argument` if range is invalid.

```
val clear : 'a t -> unit
```

    `clear ra` removes all elements from resizable array `ra`, possibly causing a reallocation.

Swapping

```
val swap : 'a t -> int -> int -> unit
```

    `swap ra n m` swaps elements at indices `n` and `m`.
    Raises `Invalid_argument` if any index is out of range.

```
val swap_in_last : 'a t -> int -> unit
```

    `swap_in_last ra n` swaps the last element with the one at position `n`.
    Raises `Invalid_argument` if index `n` is out of range.

Standard conversions

```
val to_std : 'a t -> 'a Weak.t
```

    `to_std ra` converts a resizable weak array to a standard one.

```
val sof_std : strategy -> 'a Weak.t -> 'a t
```

    `sof_std s ar` converts a standard weak array to a resizable one, using strategy `s`.

```
val of_std : 'a Weak.t -> 'a t
```

    `of_std ar` converts a standard weak array to a resizable one using the default strategy.

List conversions

```
val to_list : 'a t -> 'a option list
```

    `to_list ra` converts resizable array `ra` to a list.

```
val of_list : 'a option list -> 'a t
```

    `of_list l` creates a resizable array using the default strategy and the elements in list `l`.

Iterators

```
val iter : ('a option -> unit) -> 'a t -> unit
```

    `iter f ra` applies the unit-function `f` to each element in resizable array `ra`.

```
val iteri : (int -> 'a option -> unit) -> 'a t -> unit
```

    `iteri f ra` applies the unit-function `f` to each index and element in resizable array `ra`.

```
val fold_left : ('b -> 'a option -> 'b) -> 'b -> 'a t -> 'b
```

    `fold_left f a ra` left-folds values in resizable array `ra` using function `f` and start accumulator `a`.

```
val fold_right : ('a option -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

    `fold_right f a ra` right-folds values in resizable array `ra` using function `f` and start accumulator `a`.

Scanning of resizable arrays

```
val for_all : ('a option -> bool) -> 'a t -> bool
```

    `for_all p ra`
    Returns `true` if all elements in resizable array `ra` satisfy the predicate `p`, `false` otherwise.

```
val exists : ('a option -> bool) -> 'a t -> bool
```

    `exists p ra`
    Returns `true` if at least one element in resizable array `ra` satisfies the predicate `p`, `false` otherwise.

```
val mem : 'a option -> 'a t -> bool
```

    `mem el ra`
    Returns `true` if element `el` is logically equal to any element in resizable array `ra`, `false` otherwise.

```
val memq : 'a option -> 'a t -> bool
```

```
memq el ra
```
Returns `true` if element `el` is physically equal to any element in resizable array `ra`, `false` otherwise.

```
val pos : 'a option -> 'a t -> int option
```

```
pos el ra
```
Returns `Some index` if `el` is logically equal to the element at `index` in `ra`, `None` otherwise. `index` is the index of the first element that matches.

```
val posq : 'a option -> 'a t -> int option
```

```
posq el ra
```
Returns `Some index` if `el` is physically equal to the element at `index` in `ra`, `None` otherwise. `index` is the index of the first element that matches.

Searching of resizable arrays

```
val find : ('a option -> bool) -> 'a t -> 'a option
```

```
find p ra
```
Raises `Not_found` if there is no such element.
Returns the first element in resizable array `ra` that satisfies predicate `p`.

```
val find_index : ('a option -> bool) -> 'a t -> int -> int
```

```
find_index p ra pos
```
Raises

- `Not_found` if there is no such element or if `pos` is larger than the highest index.
- `Invalid_argument` if `pos` is negative.

Returns the index of the first element that satisfies predicate `p` in resizable array `ra`, starting search at index `pos`.

```
val filter : ('a option -> bool) -> 'a t -> 'a t
```

```
filter p ra
```
Returns a new resizable array by filtering out all elements in `ra` that satisfy predicate `p` using the same strategy as `ra`.

```
val find_all : ('a option -> bool) -> 'a t -> 'a t
```

```
find_all p ra
```
is the same as `filter`

```
val filter_in_place : ('a option -> bool) -> 'a t -> unit
```

```
filter_in_place p ra
```
as `filter`, but filters in place.

```
val partition : ('a option -> bool) ->
  'a t -> 'a t * 'a t
```

```
partition p ra
```
Returns a pair of resizable arrays, the left part containing only elements of `ra` that satisfy predicate `p`, the right one only those that do not satisfy it. Both returned arrays are created using the strategy of `ra`.

UNSAFE STUFF - USE WITH CAUTION!

```
val unsafe_get : 'a t -> int -> 'a option

val unsafe_set : 'a t -> int -> 'a option -> unit

val unsafe_sub : 'a t -> int -> int -> 'a t

val unsafe_fill : 'a t -> int -> int -> 'a option -> unit

val unsafe_blit : 'a t -> int -> 'a t -> int -> int -> unit

val unsafe_remove_one : 'a t -> unit

val unsafe_remove_n : 'a t -> int -> unit

val unsafe_swap : 'a t -> int -> int -> unit

val unsafe_swap_in_last : 'a t -> int -> unit
```

```
end
```

5 Module `Res` : Global module for resizable datastructures and default implementations

Default strategies
```
module DefStrat :
    Strat.T  with type t = float * float * int
```
Default strategy for resizable datastructures

`type t` is a triple `waste, shrink_trig, min_size`, where `waste` (default: 1.5) indicates by how much the array should be grown in excess when reallocation is triggered, `shrink_trig` (default: 0.5) at which percentage of excess elements it should be shrinked and `min_size` (default: 16 elements) is the minimum size of the resizable array.

```
module BitDefStrat :
    Strat.T  with type t = float * float * int
```
Same as `DefStrat`, but the minimum size is 1024 elements (bits).

Default instantiation of standard resizable datastructures
```
module Array :
    Pres_intf.T  with module Strategy = DefStrat
```
Resizable parameterized array using the default reallocation strategy.

```
module Floats :
    Nopres_intf.T  with module Strategy = DefStrat and type el = float
```

Resizable float array using the default reallocation strategy.

```
module Bits :
    Nopres_intf.T  with module Strategy = BitDefStrat and type el = bool
```
Resizable bit vector using the default reallocation strategy.

```
module Weak :
    Weak_intf.T  with module Strategy = DefStrat
```
Resizable weak array using the default reallocation strategy.

```
module Buffer :
    Nopres_intf.Buffer  with module Strategy = DefStrat and type el = char
```
Resizable buffer using the default reallocation strategy.

Functors for creating standard resizable datastructures from strategies

```
module MakeArray :
    functor (S : Strat.T) -> Pres_intf.T  with module Strategy = S
```
Functor that creates resizable parameterized arrays from reallocation strategies.

```
module MakeFloats :
    functor (S : Strat.T) -> Nopres_intf.T  with module Strategy = S and type el = float
```
Functor that creates resizable float arrays from reallocation strategies.

```
module MakeBits :
    functor (S : Strat.T) -> Nopres_intf.T  with module Strategy = S and type el = bool
```
Functor that creates resizable bit vectors from reallocation strategies.

```
module MakeWeak :
    functor (S : Strat.T) -> Weak_intf.T  with module Strategy = S
```
Functor that creates resizable weak arrays from reallocation strategies.

```
module MakeBuffer :
    functor (S : Strat.T) -> Nopres_intf.Buffer  with module Strategy = S and type el
= char
```
Functor that creates resizable buffers (=string arrays) from reallocation strategies.