Abstract data types to be used in specifications

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.15 $

Table of Contents

This library provides implementations of abstract data types for linear lists, stacks, sets, and mappings by C modules. The functions exported by these modules can be used in `.lido` specifications or in C modules supplied by users.

This library contains the following modules:

`LidoList` Lists in LIDO Specifications

`List` Linear Lists of Any Type

`BitSet` Bit Sets of Arbitrary Length

`IntSet` Bit Sets of Integer Size

`Stack` Stacks of Any Type

`Map` Mapping of Integers to Any Type

`Table` Mapping Arbitrary Values To Definition Table Keys

`DynSpace` Dynamic Storage Allocation

The module `LidoList` supports construction and access of linear lists in `.lido` specifications. The module `DynSpace` supports efficient dynamic storage allocation. It provides a simplified and restricted interface to the `obstack` module which is recommended to be used for dynamic allocation of arbitrary sized storage entities. The other modules are implementations of abstract data types.

The functions exported by each of these modules may be used in `.lido` computations as well as in C modules. In the latter case the interface file of the module has to be included into the C module.

1 Lists in LIDO Specifications

Several language implementation tasks are solved using linear lists constructed of data elements which are computed at certain tree nodes, e.g. a the list of parameter types that is part of the type of a function. Such lists reflect a left-to-right depth-first order of the corresponding tree nodes. Similarly such lists are decomposed in other contexts, e.g. to check the types of the arguments of a function call. This module provides `.lido` specifications for such patterns of linear list usage. The module uses functions of the linear list module `List`. Any assignable C type may be chosen as type of the list elements.

This module is instantiated by

```
$/Adt/LidoList.gnrc+instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the element type and `HDR.h` is a file that defines the element type, e.g.

```
$/Adt/LidoList.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/LidoList.gnrc+instance=int :inst
```

The module provides two groups of computational roles, `TYPEListRoot`, `PreTYPEListElem`, `PostTYPEListElem`, `TYPEListElem`, for construction of a list from attribute values at tree nodes, and `TYPEDeListRoot`, `PreTYPEDeListElem`, `PostTYPEDeListElem`, `TYPEDeListElem` for distribution of a list over attributes of tree nodes. For each of the two cases there is one role that characterizes the root of the subtree where the list construction or distribution is applied, and three roles for nodes that are related to list elements. These three roles affect the order of list elements differently in cases where the related tree nodes may occur recursively (see below).

For construction of a list `TYPEListRoot` is associated to a grammar symbol that contains all occurrences of the roles for its list elements in its subtree. If `TYPEListRoot` occurs recursively in the tree, its lists are collected separately. The resulting list of type `TYPEList` is obtained by the attribute `TYPEListRoot.TYPEList`.

A user's computation has to provide the list element values of type `TYPE` by attributes named `TYPEElem` at the symbols which have one of the three element roles. One of the three roles should be chosen depending on the desired order of the elements in cases where the list element symbol occurs recursively in the tree:

If the role `PreTYPEListElem` is used the elements are taken in pre-order; i.e. the attribute value of a node occurs in the list prior to those of nodes in its subtrees.

If the role `PostTYPEListElem` is used the elements are taken in post-order; i.e. the attribute value of a node occurs in the list after those of nodes in its subtrees.

If the role `TYPEListElem` is used no elements are taken from subtrees of an element node.

There are situations where not all tree nodes that have the role `TYPEListElem` shall contribute an element to the list. A condition attribute `TYPEListElem.TYPETakeIt` of type `int` can be computed such that it is false (`0`) if this tree node shall not contribute to the list. The value of the attribute `TYPEListElem.TYPEElem` is irrelevant in that case. If the condition attribute `TYPEListElem.TYPETakeIt` is true the value of the attribute `TYPEListElem.TYPEElem` is taken as an element of the list. A default computation sets `TakeIt` to true (`1`). It becomes effective if it is not overridden as described above.

`TYPEFilterListElem` is outdated. Its task should be achieved using the attribute `TYPETakeIt`. `TYPEFilterListElem` may be used instead of `TYPEListElem`. Then the value `TYPEFilterListElem.TYPEElem` will only be inserted into the list if a call of the function `TYPEFilter` yields non-null when given `TYPEFilterListElem.TYPEElem` as argument. The function `TYPEFilter` has to be defined if the role `TYPEFilterListElem` is used.

For decomposition of a list `TYPEDeListRoot` is associated to a grammar symbol, and a computation has to be provided such that the attribute `TYPEDeListRoot.TYPEList` gets a list value. That list value is decomposed such that each occurrence of grammar symbols having one of the element roles for decomposition (see below) get a list element value. The list element values are obtained by attributes named `TYPEElem`. If the list is shorter than the number of the element nodes in the subtree the attributes of the remaining nodes get the value `NoTYPE`. `TYPEDeListRoot.TYPEListTail` is the list of remaining elements which are not associated to element nodes in the subtree, if any.

One of the three element roles for list decomposition should be chosen depending on the desired order of the elements in cases where the list element symbol occurs recursively in the tree:

If the role `PreTYPEDeListElem` is used the list elements are associated to `TYPEElem` attributes of nodes in pre-order, i.e. the attribute of a node gets an element of the list which occurs before those elements in the list that are distributed at the subtrees of the node.

If the role `PostTYPEDeListElem` is used the list elements are associated to `TYPEElem` attributes of nodes in post-order, i.e. the attribute of a node gets an element of the list which occurs after those elements in the list that are distributed at the subtrees of the node.

If the role `TYPEDeListElem` is used no elements are distributed to subtrees of an element node.

A condition attribute `TYPETakeIt` of type `int` is computed to true (`1`) by default. It determines whether an element of the list is taken at this node. That computation may be overridden by a nontrivial computation if such a condition is desired.

If list decomposition is used the name `NoTYPE` has to be defined suitably in a user's specification, e.g. in `HDR.h`.

Both `TYPEListRoot` and `TYPEDeListRoot` may be recursively nested without affecting each other.

An example for the use of this module in type analysis is given in (see Section "Function Types" in *Type analysis tasks*): In the context of a function declaration the list of parameter types is composed and associated as a property of the function type. In the context of a function call that property is accessed, the list is decomposed, and its elements - the formal parameter types - are compared with the types of the arguments.

2 Linear Lists of Any Type

This module implements linear lists whose elements are of an arbitrary type that is specified by a generic instantiation parameter. Any assignable type can be chosen.

Storage for lists is allocated when needed. The module implementation uses efficient dynamic storage allocation of the `obstack` module. The module does not implement automatic garbage collection. Storage used by one instance of this module can be deallocated completely.

One subset of the functions provided by this module is strictly functional, i.e. list values are not modified. Another subset of functions modifies existing list values, e.g. inserts elements into a list. It is explicitly mentioned which functions may cause such side-effects on their arguments.

The module is instantiated by

```
$/Adt/List.gnrc +instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the element type and `HDR.h` is a file that defines the element type, e.g.

```
$/Adt/List.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/List.gnrc+instance=int :inst
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `TYPEList.h` is imported there.

A module `PtrList` is available. It produces list implementations with the same interface as the `List` module does. `PtrList` is only applicable if the element type `TYPE` is a pointer type. An instantiation of `PtrList` implements the lists by lists of `VoidPtr` (`void*`) using an instantiation of the `List` module. Hence, several instances of `PtrList` use one single implementation. The created interface files `TYPEList.h` provide macros for casting the element type to and from `VoidPtr`. The module `PtrList` is instantiated in the same way as the `List` module is:

```
$/Adt/PtrList.gnrc +instance=TYPE +referto=HDR :inst
```

The modules export the following type names and macros:

`TYPEList`
: A pointer type representing lists.

`TYPEListPtr`
: A pointer type pointing to TYPEList objects.

`TYPEMapFct`
: A function type for mapping: `TYPE -> TYPE`.

`TYPECmpFctType`
: A function type for comparison: `TYPE, TYPE -> int` Function of this type have to yield 0 if the two values are equal, 1 if the left argument is greater than the right, -1 if the left argument is less than the right.

`TYPESumFct`
: A function type for combining elements: `TYPE, TYPE -> TYPE`

`NULLTYPEList`
: Denotes the empty list.

`NullTYPEList ()`
: Denotes the empty list.

`SingleTYPEList(e)`
: Creates a list containing only the element `e`.

The following list processing functions are supplied by the module:

`void FinlTYPEList (void)`
: Deallocates all TYPELists (for *all* possible values of TYPE).

`TYPEList ConsTYPEList (TYPE e, TYPEList l)`
: Constructs a TYPEList of an element `e` and a given tail `l`. `e` is the first element of the list.

`TYPE HeadTYPEList (TYPEList l)`
: Returns the first element of the list `l`. The list `l` must not be empty.

`TYPEList TailTYPEList (TYPEList l)`
: Returns the tail of the list `l`. If `l` is empty, an empty list is returned.

`int LengthTYPEList (TYPEList l)`
: Returns the number of elements in the list `l`.

`TYPE IthElemTYPEList (TYPEList l, int i);`
: Returns the `i`-th element of the List `l`. The head of `l` is referred to as 1. If the value of i is greater than the length of the list, an error is reported and the program exits.

`TYPEList CopyTYPEList (TYPEList l, TYPEMapFct cp)`
: Copies the list `l`. Elements are copied by calls of `cp`.

`TYPEList AppTYPEList (TYPEList l1, TYPEList l2)`
: Concatenates two lists `l1` and `l2`. The resulting list contains `l2` at the end of a copy of list `l1`. Hence, no argument is modified.

`TYPEList AppElTYPEList (TYPEList l, TYPE e)`
: Appends an element `e` to the list `l`. The list `l` is not copied, it is modified as a side-effect of this function.

`void InsertAfterTYPEList (TYPEList l, TYPE e)`
: This function requires a non-empty list `l`. The element `e` is inserted just after the first element of `l`. The list `l` is modified as a side-effect of this function.

`TYPEList OrderedInsertTYPEList (TYPEList l, TYPE e, TYPECmpFctType fcmp)`
: Inserts the element `e` into the list `l` maintaining `l` in ascending order with respect to the compare `fcmp`. The updated list is returned. The list `l` may be modified as a side-effect of this function.

`TYPEListPtr RefEndConsTYPEList (TYPEListPtr addr, TYPE e);`
: Appends an element `e` to the end of a list given by its address `addr`. The address where the next element may be appended is returned. The list is modified as a side-effect of this function.

`TYPEListPtr RefEndAppTYPEList (TYPEListPtr addr, TYPEList l);`
: Appends a list `l` to the end of a list given by its address `addr`. The address where the next element may be appended is returned. The list is modified as a side-effect of this function.

`int ElemInTYPEList (TYPE e, TYPEList l, TYPECmpFctType cmpfct);`
: This function returns 1 iff the element `e` is in the list `l`. List elements are compared by the function `cmpfct`.

`TYPEList AddToSetTYPEList (TYPE e, TYPEList l, TYPECmpFctType cmpfct)`
: If `l` contains `e` then `l` is returned. Otherwise a list is returned that contains `e` and the elements of `l`. The comparison function `cmpfct` is used to check whether `l` already contains `e`. The list `l` is not modified.

`TYPEList AddToOrderedSetTYPEList (TYPE e, TYPEList l, TYPECmpFctType cmpfct)`
: If `l` contains `e` then `l` is returned. Otherwise a list is returned that contains `e` and the elements of `l`. The comparison function `cmpfct` is used to check whether `l` already contains `e`. `l` is assumed to be ordered increasingly in the sense of `cmpfct`. The list `l` may be modified as a side-effect of this function.

`TYPEList MapTYPEList (TYPEList l, TYPEMapFct f);`
: Returns a new list obtained by applying `f` to each element of `l`.

`int CompTYPEList (TYPEList l1, TYPEList l2, TYPECmpFctType f);`
: Compares the lists `l1` and `l2` lexicographically by applying `f` to the corresponding elements.

`TYPE SumTYPEList (TYPEList l, TYPESumFct f, TYPE a);`
: Applies the binary function `f` to the elements of the list: `f( f(... f(a, e1), e2, ...), en)` If `l` is empty `a` is returned.

It should be pointed out that the functions `AppElTYPEList`, `InsertAfterTYPEList`, `OrderedInsertTYPEList`, `RefEndConsTYPEList`, `RefEndAppTYPEList`, `AddToOrderedSetTYPEList` modify existing lists and hence cause side-effects. If the non modified original list values are still to be used they have to be copied (`CopyTYPEList`) before they are modified. The other functions can be used in a strictly functional style.

3 Bit Sets of Arbitrary Length

This module implements operations on sets over elements which are nonnegative numbers. The range of each set value is dynamically adapted as required by the operations.

Storage for set values is allocated when needed. The module implementation uses efficient dynamic storage allocation of the `obstack`. The module does not implement automatic garbage collection. Storage used by one instance of this module can be deallocated completely, or for each single set value.

Some set operations are provided in two versions: The functional version allocates a new result value and leaves its operands unchanged. The imperative version modifies one of its operands to represent the result of the operation. Note: In the imperative case, too, the result of the function call rather than the operand has to be used for subsequent accesses to the modified set value, e. g. `s = AddElemToBitSet (x, s);` where s is a `BitSet` variable.

The module does not have generic parameters. It is used by writing

```
$/Adt/BitSet.fw
```

in a `.specs` file.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `BitSet.h` is imported there.

The module exports the following type names and macros:

`BitSet` A pointer type representing sets.

`NullBitSet`
: The 0 pointer representing an empty set. Note: The empty set can be represented in different ways. Hence, the function `EmptyBitSet` must be used to check for a set being empty.

The following set processing functions are supplied by the module:

`void FreeBitSet (BitSet s)`
: Deallocates the set `s`.

`void FreeMemBitSet (void)`
: Deallocates all memory allocated for sets.

`BitSet NewBitSet (void)`
: Allocates an empty set.

`int EqualBitSet (BitSet s1, BitSet s2)`
: Yields 1 if `s1` and `s2` contain the same elements; otherwise 0.

`int EmptyBitSet (BitSet s)`
: Yields 1 if `s` is empty; otherwise 0.

`int EmptyIntersectBitSet (BitSet s1, BitSet s2)`
: Yields 1 if the intersection of `s1` and `s2` is empty; otherwise 0.

`int ElemInBitSet (int el, BitSet s)`
: Yields 1 if `el` is an element of `s`; otherwise 0.

`int CardOfBitSet (BitSet s)`
: Yields the number of elements in `s`.

`BitSet AddElemToBitSet (int el, BitSet s)`
: Imperative: Adds element `el` to set `s`.

`BitSet ElemToBitSet (int el)`
: Returns a set consisting only of element `el`. This function can be used as the third (function) parameter in an application of the `CONSTITUENTS` construct.

`BitSet AddRangeToBitSet (int el1, int el2, BitSet s)`
: Imperative: All elements in the range from `el1` to `el2` are added to the set `s`.

`BitSet SubElemFromBitSet (int el, BitSet s)`
: Imperative: Subtracts element `el` from set `s`.

`BitSet UnionToBitSet (BitSet s1, BitSet s2)`
: Imperative: `s1` is set to the union of `s1` and `s2`

`BitSet IntersectToBitSet (BitSet s1, BitSet s2)`
: Imperative: `s1` is set to the intersection of `s1` and `s2`.

`BitSet SubtractFromBitSet (BitSet s1, BitSet s2)`
: Imperative: `s2` is subtracted from `s1`.

`BitSet ComplToBitSet (int upb, BitSet s)`
: Imperative: `s` is complemented with respect to the range 0 .. `upb`; no assumption can be made on elements larger than `upb` in `s`

`BitSet UniteBitSet (BitSet s1, BitSet s2)`
: Functional: Yields the union of `s1` and `s2`

`BitSet IntersectBitSet (BitSet s1, BitSet s2)`
: Functional: Yields the intersection of `s1` and `s2`

`BitSet SubtractBitSet (BitSet s1, BitSet s2)`
: Functional: Yields `s1` minus `s2`.

`BitSet ComplBitSet (int upb, BitSet s)`
: Functional: Yields the complement of `s` with respect to the range 0 .. `upb`; no assumption can be made on elements larger than `upb` in the result.

`int NextElemInBitSet (int elem, BitSet s)`
: Yields the smallest element of `s` that is larger than `elem`, if any; -1 otherwise.

`void ApplyToBitSet (BitSet s, void func(int))`
: Applies the function `func` to each element of `s`

`void PrintBitSet (BitSet s)`
: Prints `s` as a string of 0 and 1 to `stdout`.

`void PrintElemsBitSet (BitSet s)`
: Prints `s` as a comma separated sequence of its elements to `stdout`.

4 Bit Sets of Integer Size

This C module implements sets of small nonnegative integral values by values of an `unsigned` type that is determined on instantiation of the module. The maximal element value depends on the number of bits used for the representation of the chosen `unsigned` type. The operations provided by this module may be used for computations on kind set values (or any other suitable application). In the following description of the set operations we assume that `el` stands for an expression that yields an integer being a set element, and `s, s1, s2` stand for expressions yielding sets.

The module is instantiated by

```
$/Adt/IntSet.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `IntSet` instance and `TYPE` specifies the type used for representing set values. If the `referto` parameter is specified to be `TYPE`, then `unsigned TYPE` has to be a valid integral C type. Examples for the `referto` parameter are `short`, `int`, or `'long int'`. If the `referto` parameter is omitted `int` is assumed. The `instance` parameter may be omitted if there is only one instance of this module.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `NAMEIntSet.h` is imported there.

The module exports the following type names and macros (all names are prefixed by the value `NAME` of the `instance` parameter):

`NAMEIntSet`
: the type representing sets.

`NAMENULLIS`
: the unique value for an empty set.

`NAMENullIS()`
: a macro with no arguments for the unique value for an empty set, to be used where a functional notation is required as in `WITH` clauses of `CONSTITUENTS`.

The following operations for set processing are supplied by the module. It is indicated whether an operation is implemented by a function or by a macro. The macros do not evaluate any argument repeatedly.

`NAMEIntSet NAMESingleIS (int el)`
: Yields a singleton set containing `el`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEAddElIS (int el, NAMEIntSet s)`
: Yields the union of `NAMESingleIS(el)` and `s`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEConsIS (int el, NAMEIntSet s)`
: Yields the union of `NAMESingleIS(el)` and `s`; validity of `el` is not checked. (macro)

`int NAMEInIS (int el, NAMEIntSet s)`
: Yields 1 iff `el` is in `s`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEUniteIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields the union of `s1` and `s2`. (macro)

`NAMEIntSet NAMESubIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields `s1` minus `s2`. (macro)

`NAMEIntSet NAMEInterIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields the intersection of `s1` and `s2`. (macro)

`int NAMEDisjIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields 1 iff `s1` and `s2` are disjoint. (macro)

`int NAMEInclIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields 1 iff `s1` is a subset of `s2`. (function)

`int NAMEEqualIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields 1 iff `s1` is equal to `s2`. (macro)

`int NAMEEmptyIS (NAMEIntSet s)`
: Yields 1 iff `s` is empty. (macro)

`int NAMECardIS (NAMEIntSet s)`
: Yields the number of elements in `s`. (function)

5 Stacks of Any Type

This module implements a stack named `NAMEStack`, whose elements are of type `TYPE`. Values of this type can be pushed onto the stack and popped off of it in the usual way, and in addition each element of the stack can be indexed directly and its value obtained. All of the operations exported by this module are implemented as macros, using the facilities of the obstack module (see Section "Memory Object Management" in *Library Reference Manual*).

The module is instantiated by

```
$/Adt/Stack.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `Stack` instance and `TYPE` is the element type.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `NAMEStack.h` is imported there.

The following macros are supplied by the module:

`int NAMEStackEmpty`
: Yields the value 1 if the stack has no elements, 0 otherwise.

  This operation must appear in an expression context.

`size_t NAMEStackSize`
: Yields the number of elements in the stack.

  This operation must appear in an expression context.

`void NAMEStackPush(TYPE v)`
: Push an element onto the stack. The parameter `v` must be an expression that yields a value of type `TYPE`. That value becomes the new top element of the stack. The previous top element becomes the new second element, and so on.

  This operation must appear in a statement context.

`TYPE NAMEStackPop`
: Remove the top element of the stack. The previous second element becomes the new top element, and so on.

  This operation must appear in an expression context, and yields the value (of type `TYPE`).

`TYPE NAMEStackTop`
: Obtain the contents (of type `TYPE`) of the top element of the stack without changing the state of the stack.

  This operation must appear in an expression context.

`TYPE NAMEStackElement (i)`
: Obtain the contents (of type `TYPE`) of a specific element of the stack without changing the stack. The argument gives the distance of the desired element from the top of the stack (0 for the newest element, 1 for the next newest, and so on). There is no check on the validity of the value of `i`.

  This operation must appear in an expression context.

`TYPE NAMEStackArray (i)`

Obtain the contents (of type `TYPE`) of a specific element of the stack without changing the state of the stack. For the purpose of this operation, the stack is considered to be an array. Element 0 is the oldest value on the stack, element 1 is the next oldest, and so on. There is no check on the validity of the value of `i`.

This operation must appear in an expression context.

`void ForEachNAMEStackElementDown (i)`

Cycle through the elements of the stack, from the most recent to the oldest. The parameter `i` must be declared as an lvalue of type `TYPE*` and will point, in turn, to each element of the stack.

This operation must appear in a context where `for (i=...; i>=...;i--)` is allowed.

`void ForEachNAMEStackElementUp (i)`

Cycle through the elements of the stack, from the oldest to the most recent. The parameter `i` must be declared as an lvalue of type `TYPE*` and will point, in turn, to each element of the stack.

This operation must appear in a context where `for (i=...; i<...;i++)` is allowed.

6 Mapping Integral Values To Other Types

This module implements mappings from non-negative integers to an arbitrary type that is specified by a generic instantiation parameter. Any assignable type can be chosen.

The map is implemented by a dynamically allocated array using the `obstack` module via the `DynSpace` interface. The mapping storage can be deallocated.

The size of the mapping has to to be stated on initialization.

The module is instantiated by

```
$/Adt/Map.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `Map` instance and `TYPE` is the target type of the mapping.

Note: The target type of the mapping `TYPE` must be either a type that is predefined in C, or its definition must be made available by some `.head` specification.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `NAMEMap.h` is imported there.

The following functions are supplied by the module:

`void NAMEInitMap (int max_n)`
: Initializes the mapping for value between 0 and `max_n`.

`void NAMEFinlMap (void)`
: Deallocates the mapping.

`void NAMEInitMapValues (TYPE v)`
: Initializes all mappings in the range 0 to `max_n` to the value `v`.

`void NAMESetMap (int n, TYPE v)`
: Maps `n` to the value `v`. It may override a previously set value.

`TYPE NAMEGetMap (int n)`
: Yields the `TYPE` value set for `n` by the most recent call of `NAMESetMap` or `InitMapValues`, if any; otherwise the result is undefined.

7 Mapping Arbitrary Values To Definition Table Keys

This module implements mappings from values of an arbitrary type (specified by a generic instantiation parameter) to unique definition table keys. It is instantiated by

```
$/Adt/Table.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the table instance and `TYPE` is the type of the values being mapped to definition table keys.

Note: if `TYPE` is not a type that is predefined in C then its definition must be made available by some `.head` specification.

Each table is implemented by a dynamically allocated memory with a 32-bit address space. The user must supply a hash function to compute a 32-bit value that characterizes a given `TYPE` value. If `TYPE` is representable in 32 bits, then this function is simply an identity function that casts the given value to type `ub4` (representing an unsigned 32-bit value). Otherwise the general hash function available in the Eli library can compute an appropriate value (see Section "Computing a Hash Value" in *Solutions of common problems*).

Each element of the memory may hold more than one table entry; the user must supply a comparison function to verify that a particular table entry is the one sought. If `TYPE` values are ordered, this function should return an integer less than, equal to, or greater than zero when its first `TYPE` argument is less than, equal to, or greater than its second. Otherwise, the function must return zero if its `TYPE` arguments are equal, and an integer greater than zero if they are unequal.

The following functions are supplied by the module:

`void NAMEInitTable (ub4 (*hash)(TYPE), int (*cmp)(TYPE, TYPE))`
: Initializes the table. `hash` is the hash function on `TYPE` values, and `cmp` is the comparison function on `TYPE` values.

`DefTableKey NAMEKeyInTable (TYPE v)`
: Yields the definition table key associated with the value `v`. If there is no definition table key associated with the value `v` then `NAMEKeyInTable` yields `NoKey`.

`DefTableKey NAMEDefInTable (TYPE v)`
: Yields the definition table key associated with the value `v`. If there is no definition table key associated with the value `v` then `NAMEDefInTable` associates the value `v` with a new definition table key and yields that key.

These functions can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `NAMETable.h` is imported there.

8 Dynamic Storage Allocation

This module provides functions for dynamic storage allocation. Its operations are significantly faster than the C function `malloc`, since storage is allocated within larger chunks which are less frequently requested from the system. The module is implemented on top of a more general storage allocation module `obstack`, which may be used directly if more elaborate allocation mechanisms are needed.

The use of the module requires an initializing call of the function `InitDynSpace`.

`void* InitDynSpace ()`
: initializes a storage area for subsequent allocations. The result should be assigned to a variable used in subsequent allocation calls.

The module does not have generic parameters. It is used by writing

```
$/Adt/DynSpace.fw
```

in a `.specs` file.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in C modules if the interface file `DynSpace.h` is imported there.

The module exports the following functions:

`void* InitDynSpace (void)`
: Initializes a dynamic storage area and returns a pointer to it. it has to be used in any subsequent call of `DynAlloc`.

`void* DynAlloc (void *space, int size)`
: Allocates storage of the given `size` in the area pointed to by `space`.

`void DynClear (void *space)`
: Deallocates the complete storage area pointed to by `space`.

The functions can be used in a C module in the following way:

```
#include "DynSpace.h"
void *MySpace;
/* ... */
MySpace = InitDynSpace ();
/* ... */
p = (TypeX*) DynAlloc (MySpace, sizeof (TypeX));
/* ... */
DynClear (MySpace);
/* ... */
```

Note: Neither the cast nor the call of sizeof can be part of a LIDO computation, since type identifiers are not allowed in computations. Hence the calls of this module must reside in a C module (that implements the type of the allocated objects).

Index