A Verified Implementation of the Bounded List Container

Raphaël Cauderlier and Mihaela Sighireanu

IRIF, University Paris Diderot and CNRS, Paris, France
firstname.name@irif.fr

Abstract. This paper contributes to the trend of providing fully verified container libraries. We consider an implementation of the bounded doubly linked list container which manages the list in a fixed size, heap allocated array. The container provides constant time methods to update the list by adding, deleting, and changing elements, as well as cursors for list traversal and access to elements. The library is implemented in C, but we wrote the code and its specification by imitating the ones provided by GNAT for the standard library of Ada 2012. The proof of functional correctness is done using VeriFast, which provides an auto-active verification environment for Separation Logic extended with algebraic data types. The specifications proved entail the contracts of the Ada library and include new features. The verification method we used employs a precise algebraic model of the data structure and we show that it facilitates the verification and captures entirely the library contracts. This case study may be of interest for other verification platforms, thus we highlight the intricate points of its proof.

1 Introduction

Standard libraries of programming languages provide efficient implementations for common data containers. The details of these implementations are abstracted away by generic interfaces which are specified in terms of well understood mathematical structures such as sets, multisets, sequences, and partial functions. The intensive use of container libraries makes important their formal verification.

However, the functional correctness of these libraries is challenging to verify for several reasons. Firstly, their implementation is highly optimized: it employs complex data structures and manages the memory directly through pointers/references or specific memory allocators. Secondly, the specification of containers is rarely formal. Notable exceptions are, e.g., Eiffel [27] and SPARK [10]; recently, [1] provided a specification of the Ada 2012 container library. The formal specifications are very important when the library employs constructs that are out of the scope of the underlying mathematical structure. A typical example of such constructs are iterators. For example, Java iterators are generic and can exist independently of the container; Ada 2012 iterators, called cursors, are part of the container. Thirdly, the specification of the link between the low level implementation and the mathematical specification requires hybrid logics that are

able to capture both low level and high level specifications of the container. For verification purposes, these logics shall be supported by efficient solvers.

This work focuses on the functional verification of the *bounded doubly linked lists* container, which is a GNAT implementation [11] of the doubly linked lists container in the standard library of Ada 2012 [1]. This container is currently used by client programs [2] written in SPARK [21], a subset of the Ada targeted at safety- and security-critical applications. The lists have bounded capacity, fixed at the list creation, and thus avoid dynamic memory allocation during the container use. This feature is required in critical code, where it is necessary to supply formal guarantees on the maximal amount of memory used by the running code.

The container implementation is original compared with other implementations of linked lists inside arrays. It employs an array of fixed size in which it manages (i) the occupied array cells inside a doubly linked list representing the content of the container and (ii) a singly linked list of free array cells. The operations provided are classic for lists. A special mention for the preservation of the *amortized constant time complexity* by the implementation of insert and delete operations. The list elements are designated using (bi-directional) cursors, also used to traverse the list. In conclusion, the code of this container was designed to ensure efficiency of operations and not its verification, and therefore it provides a realistic test for the automated verification.

Thanks to the introduction of formal contracts in Ada 2012, the container has been fully specified recently based on a previous specification in Why3 by Dross et al. [10]. The specification given is "meant to facilitate the formal verification of code using this container" [11], and it is presently used to prove the clients written in SPARK. The container is specified in terms of a model representing a functional implementation of bounded vectors, also written in Ada. This kind of specification is a substitute for the algebraic data types, not supported by Ada. It has the advantage of being executable, which enables the run-time verification of the implementation. An important feature of these contracts is their completeness [26] with respect to the models considered for the container and the cursors. This aspect is a challenge for the state of the art verification tools. The formal verification of these contracts can not be done by GNATprove, the deductive verification environment for SPARK, because the code employs language constructs out of its scope.

The goal of our study is to apply on-the-shelf verification tools to prove the full functional correctness and the memory safety for this implementation, without simplifying the code or the specification. To open the case study to more verification platforms, we choose to write this library in C, because C may capture all the features of the container implementation, except the strong typing and the generic types of Ada. The C implementation mimics the Ada code. The functional specification of the C code translates the container contracts from Ada based on (i) *representation predicates* that relate heap regions with algebraic models using inductively defined predicates, (ii) algebraic lists and maps, and (iii) inductively defined predicates and functions on the algebraic models. The logic including

these features is undecidable in general. Therefore, we have to help the prover to obtain push button verification. The auto-active verification [19] environments are helpful in such tasks.

The invariant properties of the implementation and the features exhibited by the specification guided us towards deductive verification platforms that support Separation Logic [30] (SL) and algebraic data types. Consequently, we choose the VeriFast [14] auto-active verification tool, which provides means for (a) the specification of representation predicates in the style introduced for SL by O'Hearn et al. [24], (b) the definition of polymorphic algebraic data types, predicates and functions, and (c) the definition of user lemmas to help verification. Using these features, we employ a verification methodology based on the refinement of the original specification. The refined specification not only captures accurately the contracts, but also eases the deductive verification process, i.e., the writing of lemmas. For example, we employ a style of writing representation predicates in SL that leads to simpler lemmas for list segments composition.

To summarize, we verified the C implementation of bounded doubly linked list container against its functional specification. In addition, we verified the safety of memory accesses due to use of Separation Logic. For this, we annotated the C code and we extended the library for algebraic polymorphic lists of VeriFast with new predicates and lemmas. These logic development may be used in other verification tools based on induction. The annotated code and the sources used by its proof are available at `http://vecolib.imag.fr/index.php/Deliverables`.

The paper begins by presenting the case study in Section 2. Then, we highlight in Section 3 the main ingredients of the verification approach used and the challenges we faced up. Section 4 presents the experimental results. We compare this work with other approaches for verification of containers and complex data structures in Section 5.

2 Dynamic Bounded Doubly-Linked Lists

This section presents the container code and its functional specification.

2.1 Overview

Implementation: The code is written in a very simple fragment of C, which may be easily translated to most imperative programming languages. It uses records and pointers to records, dynamic memory allocation, classic accesses to record fields and array elements, basic integer type and its operations. Like in the original code, the container does not support concurrency and has been written to obtain efficient operations and not to ease the verification. The container elements are designated through *cursors*, which represent valid positions in the list; they may be moved forward and backward in the list. The container interface includes 30 operations including classic operations (creation, copy, size access, clearing and deallocation, equality test, searching) and a rich set of utilities (inserting or deleting bunches of elements at some position, searching from the end, merging lists, swapping elements or links, reversing in place, sorting).

Specification: The functional specification is *model based* [27]. Two mathematical models are used: *algebraic lists* (i.e., finite sequences) to represent the content of the list and *finite partial maps* to model the set of valid cursors (see Section 2.3 for details). The contracts employ operations on these mathematical models that are beyond their classic usage. For example, the test of inclusion between the set of elements of two sequences, or the test that the domain of a partial mapping has been truncated from a given value. For this reason, we enriched the library of mathematical models provided by our prover with such operations and the corresponding axiomatizations (see Section 3.2).

An important feature of our functional specification is the usage of *a refined abstraction* for the list to ease the proof that the operations satisfy their contracts. We introduce a *precise model* for the list, which is an algebraic list of abstract cells, storing container values together with the links between the cells. This precise model is mapped to the abstract model (sequence of values) using a catamorphic mapping [34], called `model`. Moreover, the precise model is used to compute the (abstract) model of cursors, based on a catamorphic mapping, called `positions`. The use of the precise model facilitates the verification effort for proving that implementations of operations satisfy their contracts (see Section 3.1).

The functional specification is complete in the sense given by [26]: the post-condition of each operation uniquely defines its result and the side effect on the model of the container and of its cursors. However, it does not check for memory overflow at the container creation.

For the syntax of specifications, we employ in the following the specification language of VeriFast, which extends the normalized specification language for C, ACSL [3], with shorthand notations and operators for Separation Logic. Therefore, we employ '`?`' to introduce existentially quantified variables, '`&*&`' for both classic conjunction and the separating one, '`|->`' for the points-to operator that defines the content (right operand) of an allocated memory cell (left operand), and `emp` for the empty heap. Algebraic lists of VeriFast have type `list` and are polymorphic; the operations on lists have classic names. The definition of new logic types (and functions) is introduced by the keyword `inductive` (resp. `fixpoint`).

2.2 List container

List elements The data stored in the list container is typed by an abstract type `Element_Type`, defined as an alias to the integer type in our code. This coding is sound for the proof of the functional correctness of the container implementation because the container assumes only that values of `Element_Type` may be compared for equality.

List cell Also called *node* in the following, the list cell encapsulates the container element together with links to the next and previous cell in the list. A node is also an element of the array allocated for the container.

```
inductive pnode = pnode(int, int, Element_Type);
fixpoint int pprev(pnode pn) {switch(n) {case pnode(pp, pn, pe): return pp;}}
fixpoint int pnext(pnode pn) ...
fixpoint int pelem(pnode pn) ...

predicate node(struct Node_Type* n, int capacity; pnode pn) =
  malloc_block_Node_Type(n) &*&
  n->prev|->?iprev &*& n->next|->?inext &*&
  n->elem|->?pelem &*&
  inext>=0 &*& inext<=capacity &*&
  pn==pnode(iprev, inext, pelem);

fixpoint bool is_free(pnode n) { return pprev(n)==-1; }
predicate free_node(struct Node_Type* n, int capacity; int inext) =
  node(n, capacity, ?pn) &*&
  is_free(pn)==true &*& inext==pnext(pn);

fixpoint bool is_occupied(pnode n) { return pprev(n)>=0; }
predicate occupied_node(struct Node_Type* n, int capacity; pnode pn) =
  node(n, capacity, pn) &*& is_occupied(pn)==true;

predicate bdll(struct Node_Type * tab, int capacity,
               int iprev, int ifrom, int ilast, int ito; list<pnode> m) =
  ifrom==ito ?
    (iprev==ilast &*& values==nil<pnode>)
  : (occupied_node(tab+ifrom, capacity, ?p) &*&
     pprev(p)==iprev &*&
     bdll(tab, capacity, ifrom, pnext(p), ilast, ito, ?mtl) &*&
     m==cons(p, mtl));

predicate uninit_free(struct Node_Type* tab, int capacity,
                      int ifrom, int ito; list<int> model) =
  ifrom==ito ? model==nil
  : (ifrom<ito &*& free_node(tab+ifrom, capacity, 0) &*&
     uninit_free(tab, capacity, ifrom+1, ito, ?mtl) &*&
     model==cons(ifrom, mtl));

predicate init_free(struct Node_Type* tab, int capacity,
                    int ifrom, int ito; list<int> model) =
  ifrom==ito ? model==nil
  : (ifrom>0 &*& free_node(tab+ifrom, capacity, ?inext) &*&
     init_free(tab, capacity, inext, ito, ?mtl) &*&
     model==cons(ifrom, mtl));

predicate free_nodes(struct Node_Type* tab, int cap,
                     int free, int size; list<int> fmodel) =
  free>=0 ? (init_free(tab, cap, free, 0, ?M) &*&
             fmodel==M &*& length(M)+size==cap)
  : (uninit_free(tab, cap, abs(free), cap+1, ?M) &*&
     fmodel==M &*& length(M)+size==cap);

predicate list_inv(struct List* L;
                   struct Node_Type* tab, int cap,
                   int free, list<int> free_model,
                   int head, int last, list<pnode> m) =
  malloc_block_List(L) &*&
  L->nodes|->tab &*& L->capacity|->cap &*& cap > 0 &*&
  malloc_block(tab, sizeof(struct Node_Type)*(cap+1)) &*&
  node(tab, cap, pnode(-1,0,_)) &*&
  L->first|->head &*& head>=0 &*& head<=cap &*&
  L->last|->last &*& last>=0 &*&
  bdll(tab,cap,0,head,last,0,m) &*& L->size|->length(vs) &*&
  L->free|->free &*& free<=cap &*&
  free_nodes(tab,free,length(m),cap,free_model);
```

Fig. 1. Logic definitions for the BDLL container

```
struct Node_Type { int prev; int next; Element_Type elem; };
```

The values of the C type are abstracted by the ghost type `pnode`, defined at line 1 in Figure 1, which records the values of node fields. The logic functions `pprev`, `pnext`, and `pelem` to access first, second, resp. third component of a `purenode` value.

The predicate `node` (line 6 in Figure 1) relates a node `n` allocated in the heap with its model `pn`. The allocation property is expressed by the predefined predicate `malloc_block_Node_Type`. The values of the fields are bound to existentially quantified variables and used to build the model of the node. The predicate `node` constrains the fields `prev` and `nxt` to be indexes in an array starting at index 0 and ending at index `capacity`.

There are two kinds of nodes in the array managed by the container: nodes occupied by list elements and nodes not yet used in the list, i.e., free. Free nodes have the `prev` field at -1 and the `elem` field is irrelevant. They are specified by the predicate `free_node` (line 14 in Figure 1), which also constraints the parameter `inext` to be equal with the value of the field `next`. Occupied nodes have the `prev` field set to a non-negative integer and the `elem` field is relevant. The predicate `occupied_node` (line 19 in Figure 1) relates the node with its abstract model.

Acyclic doubly linked list The container stores the doubly linked list (BDLL) into an array of fixed capacity, which is given at the container creation. The number of elements stored in the list can not exceed the container capacity. The nodes of the BDLL are stored starting from the index 1; index 0 plays the role of the null reference. The type of the list container is given by the following record:

```
struct List {
  int capacity; struct Node_Type* nodes; int size;
  int free; int first; int last;
};
```

The length of the list is given by the field `size`. The first and the last cells of the lists are stored at indexes `first` resp. `last`. Field `free` denotes the start of the list registering the free nodes. The operation creating the container allocates the array `nodes` and sets at free all nodes in the array. The fields denoting the size and the extreme cells of the doubly linked list are set to 0. The initialization of the `free` field is detailed in the next paragraph.

The representation predicate of the BDLL formed by the occupied nodes, `bdll`, is defined at line 22 in Figure 1 as a doubly linked list segment starting by the node at index `ifrom`, ending by the node at index `ilast`; the starting node stores as previous node `iprev`, and the ending node stores as next node `ito`. The predicate definition is classic in Separation Logic [23], except the bound constraint on the node indexes (locations). If the source `ifrom` and target `ilast` indexes are equal, the list is empty; otherwise an occupied node is present at index `ifrom` and it is linked to the previous node and the remained of the list. Notice the use of pointer arithmetics to access the node at index `ifrom`. The predicate `bdll` relates the heap specification with the mathematical model of the list, i.e., the sequence of abstract nodes. We employ the polymorphic algebraic

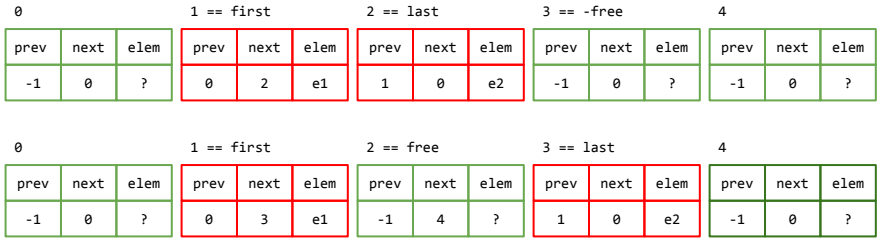

Fig. 2. Two doubly linked lists of capacity 4 and length 2

type `list` available in VeriFast mathematical library and we instantiate it with the logic type `pnode`. This precise model of list content is mapped by the inductively defined ghost function `model` to the abstract model, sequence of values of `Element_type` stored.

Acyclic list of free nodes The free nodes are organized in a singly linked list, called the *free-list*. The start of this list is given by the field `free` of the type `List`. If `free` is negative, the list is built from all nodes stored between `-free` and `capacity` included; this permits a fast initialization of the free-list at the container creation. If `free` is positive, the free-list starts at index `free`, uses as successor relation the `next` field, and ends at index 0. Figure 2 illustrates the two kinds of free lists. The representation predicate `uninit_free` (line 31 in Figure 1) is used when `free` is negative. It collects in the parameter `model` the sequence of indexes of free nodes. For the second case, we define the predicate `init_free` (line 38 in Figure 1). The two kinds of free-list are combined in the predicate `free_nodes` (line 45 of Figure 1) that calls the correct predicate depending on the sign of `free`. Notice that the constraints required by this predicates (the relation between container capacity, BDLL and free-list sizes) are all present in the Ada 2012 specification [11].

Representation predicate The invariants of the container are collected in the predicate `list_inv` (line 52 in Figure 1) which mainly specifies that the container is allocated in the heap (predefined `malloc_block_List` predicate), and its field `tab` is also allocated as a block containing `capacity`+1 records of type `Node_Type`. The first node of this array (at address `tab`) has its `prev` and `next` fields set to -1 resp. 0. The set of remaining nodes is split between the lists specified by the `bdll` and `free_nodes` predicates due to the separating conjunction. The size of the BDLL is exactly the one of its model and stored in the field `size`.

Examples of container contracts We illustrate the usage of representation predicates defined above by presenting some contracts specifying container operations. For example, the contract of the constructor is:

```
struct List* List(int capacity);
//@ requires capacity > 0;
//@ ensures list_inv(result,?t,capacity,-1,_,0,0,?m) &*& length(model(m))=0;
```

It states that the resulting container (denoted by the ghost variable `result`) is a well formed but empty bounded doubly linked list (its abstract model is the empty list) with `capacity` free nodes. As said before, the above contract (like in Ada 2012 specification), does not consider the case of memory shortage.

The contract of `is_equal` illustrates how the catamorphism `model` is used to obtain the abstract contract on the sequence of values from the precise models (given by variables `mL` and `mR` for each list parameter):

```
bool is_equal(struct List* L, struct List* R);
/*@ requires list_inv(L, ?tL, ?cL, ?fL, ?fmL, ?hL, ?lL, ?mL) &*&
             list_inv(R, ?tR, ?cR, ?fR, ?fmR, ?hR, ?lR, ?mR); @*/
/*@ ensures list_inv(L, tL, cL, fL, fmL, hL, lL, mL) &*&
            list_inv(R, tR, cR, fR, fmR, hR, lR, mR) &*&
            result == (model(mL) == model(mR)); @*/
```

The operation `clear` frees all occupied nodes. Its contract only constrains the content of the doubly linked list and leaves unspecified the free list.

```
void clear(struct List* L);
//@ requires list_inv(L,?t,?c,?f,?fm,?h,?l,?m);
//@ ensures list_inv(L,t,c,?f1,?fm1,0,0,?m1) &*& length(model(m1))=0;
```

2.3 Cursors

Following the Ada 2012 semantics [1], "a cursor designates a particular node within a list (...). A cursor keeps designating the same node (...) as long as the node is part of the container, even if the node is moved in the container. [...] If [a cursor] is not otherwise initialized, it is initialized to [...] `No_Element`." Therefore, a cursor is a record storing an *array index*. The special cursor `No_Element` is defined as a global constant storing the index 0, indeed invalid for a list node (recall that valid nodes are stored from index 1).

```
struct Cursor { int current; };
const struct Cursor No_Element = { 0 };
```

The logic type `cursor` abstracts the cursor implementation (line 1 in Figure 3). The representation predicate for cursors, `valid_cursor_or_noelem` (line 3 in Figure 3), checks that the cursor content, `index`, corresponds to an occupied node in the list using the precise model `m` of the BDLL (see line 13). Moreover, the predicate computes from `m` and `first`, the BDLL starting index, the segments `before` and `after`, into which the cursor `pc` splits `m`.

Given a BDLL container, the model of valid cursors for this container is defined (following Ada 2012 specification) as the finite bijection between the set of abstract cursors and the positions (from 1 to the size) in the list. We encode the mathematical type map by an association list, using the polymorphic type provided in the libraries of VeriFast. The cursor model is computed by the logic function `positions` (line 19 in Figure 3) from the container model, the index of the first node in the BDLL, and the first position in the list.

```
inductive cursor = NoElem | Valid(int);

predicate valid_cursor_or_noelem(struct Cursor* C,
            int index, int first, list<pnode> m;
            cursor pc,
            list<pnode> before, list<pnode> after) =
  C->current|->index &*&
  pre_valid_cursor_or_noelem(index,first,vs,pc,before,after);

predicate pre_valid_cursor_or_noelem(int index, int first, list<pnode> vs;
            cursor pc, list<pnode> before, list<pnode> after) =
  pc == (index == 0 ? NoElem : Valid(index)) &*&
  index == first ?
    (before == nil &*& after == vs) :
    (vs != nil &*&
     pre_valid_cursor_or_noelem(index,pnext(head(vs)),tail(vs),pc,?bef,after) &*&
     before == cons(head(vs),bef));

fixpoint list<pair<cursor, int> > positions (list<purenode> values,
                                             int index, int position) { ... }
```

Fig. 3. Logic definitions for cursors

Notice that this manner of specifying cursor model is coherent with the sequence model of the container, because the access to the elements of a sequence is based on positions. However, this specification choice does not combine well with inductive reasoning and induces additional work for the proof (see Section 3). We have to enrich the inductive list model with operations using positions. For example, we define the operation `M_Element(m,p)` which returns the `p`th element of the list `m`. We also defined operations `P_Has_Key` and `P_Get` on association lists to test if an abstract cursor is in the domain of the map resp. to obtain the value to which it is bound.

An example of contract using cursors is the operation `element`, which returns the value stored at the position in the list given by the cursor `C`:

```
Element_Type element(struct List* L, struct Cursor* C)
/*@ requires list_inv(L,?tab,?capacity,?free,?fm,?head,?last,?m) &*&
             valid_cursor_or_noelem(C,?index,head,m,?pc,?bef,?aft) &*&
             P_Has_Key(positions(m,head,1),pc)==true; @*/
/*@ ensures list_inv(L,tab,capacity,free,fm,head,last,m) &*&
            valid_cursor_or_noelem(C,index,head,m,pc,bef,aft) &*&
            result==M_Element(model(m),P_Get(positions(m,head,1),pc)); @*/
```

Contracts of functions changing the positions in the list (e.g., insert or delete) are complete with respect to the model of cursors. For example, consider the post-condition of operation `delete_first`, which deletes first `Count` elements of the list. It uses a conditional expression (syntax like in C) to specify two contract cases. The first case corresponds to an input container with size less than `Count`. In the second case, the container preserved its content after position `Count` (predicate `M_Range_Shifted`) and the positions of valid cursors in the new container (of model `nvs`) are shifted by `Count` (predicate `P_Positions_Shifted`) with respect to the old container.

```
void delete_first(struct List* L, int Count);
/*@ requires list_inv(L,?tab,?cap,?free,?fm,?head,?last,?m) &*&
             Count >= 0; @*/
/*@ ensures list_inv(L,tab,cap,?nfree,?nfm,?nhead,?nlast,?nm) &*&
            length(m) <= Count ? length(nm) == 0
            : length(nm) == length(m) - Count &*&
              M_Range_Shifted(model(nm),model(m),1,length(nm),Count) &*&
              P_Positions_Shifted(positions(nm,nhead,1),
                                  positions(m,head,1),1,Count); @*/
```

3 Verification Approach

We employ an auto-active verification approach [19], supported by the tool VeriFast [14]. The auto-active approach provides more automation of the verification process based on the ability given to the user to help the prover by adding annotations and lemmas and the efficient use of back-end solvers. This section highlights the methodology applied to conduct auto-active verification for this case study. This methodology is independent of the specific tool used. We also comment on the advantages and difficulties encountered with the tool used. Notice that we do not have prior experience with VeriFast.

3.1 Model-based Specification for Verification

The contracts provided for our container are in a first order logic over sequences and maps, which employs recursive logic functions. This theory is undecidable so we have to provide lemma to help the prover to tackle verification conditions.

Usage of a precise model is the solution we found to ease the writing of lemmas. It consists in refining the abstract model used for the container specification into a model that captures more details on the container organization. The abstract model is obtained from the refined one using a catamorphic mapping. This methodology is required by the gap between the abstract model and the lower level implementation of the container.

Let us explain why this methodology leads to efficient verification in our case. Consider the specification where (i) the model for the container is the sequence of the value stored and (ii) the model for the cursors is the mapping of occupied nodes to list positions. To capture these models with the representation predicate for the heap, i.e., the predicate `bdll` defined at line 22 in Figure 1, we have to replace the model `m` by the sequence of values `vs` and the map of cursors `mc`. The verification of iterative operations on the list requires to provide a lemma that allows to compose "well linked" list segments into a new list segment, i.e.,

```
lemma void bdll_concat(struct Node_Type * t,
                int p, int f, int l1, int n1, int l, int n)
requires bdll(t,?c,p,f,l1,n1,?vs1,?mc1) &*&
         bdll(t,c,l1,n1,l,n,?vs2,?mc2) &*& node(t+n, c, ?pn);
```

```
ensures bdll(t,c,p,f,l,n,append(vs1,vs2),
             append_maps(mc1,mc2)) &*& node(t+n, c, pn);
{ ... }
```

This lemma employs an operation `append_maps`, that concatenates two models of counters `mc1` and `mc2` such that the positions associated with counters in `mc1` are shifted by the size of the domaine of `mc1`. This operation is more difficult to axiomatize that list concatenation. Moreover, all invariant proofs require to keep together the two loosely related models (sequence and map) which leads to less modular proofs. Our solution to this problem is to employ the precise model of the list segment represented by the `bdll` predicate, as has been presented in Section 2.2. The composition lemma for `bdll` predicate is simpler because it avoids the reasoning on the model of cursors.

The catamorphism mappings used to obtain the abstract model of the container and the model of valid cursors have good inductive definitions and enable efficient decision procedures [34]. However, these decision procedures are not available in our verification tool; this work may be a motivation to add them.

Specification of user types by representation predicates mapping them to inductive types is classical in Separation Logic. We encode the invariant of the BDLL data structure in the predicate `list_inv`. The adoption of C for the implementation keeps us away from the problems of verifying object-related properties, for example. However, this choice leads to an overburden in annotations because we have to specify that parameters of type '`struct List*`' satisfy the invariant.

Additional annotation have been supplied to axiomatize global constants (like the `No_Element` record in Figure 1) and arrays of user-defined structures (like `nodes` in `List`).

Contract cases are intensively used in the considered GNAT library. We got around the absence of contract cases in VeriFast using conditional expressions and logic predicate and functions that relate two models (old and new). We do not observe any expressivity or performance problems with this method of encoding contracts.

3.2 Support for Specification Types

Specification of model types is done based on the mathematical models sequence (or inductive polymorphic list) and map (or inductive polymorphic association list). The VeriFast libraries including these models (mainly `list.*`) contain 9 predicates and 20 lemmas, and are not enough for the operations on models required in our specifications. We added tens of lemma and predicates. They are useful not only for the container proof but also for the verification of client program with inductive back-end solvers. (Nowadays, these proofs are done by GNATprove by calling SMT solvers with quantifiers support.)

More problematic was the lack of support for finite maps and automation of inductive reasoning. VeriFast does not provide sets and finite maps as primitives.

The encoding of cursors model by association lists renders lemmas needed on cursor models more complex. For example, map inclusion is defined as follows:

```
predicate P_le<t>(list<pair<t,int> > Left, list<pair<t,int> > Right) =
    switch(Left) {
        case nil: return true;
        case cons(p, m):
            return P_Has_Key(Right, fst(p)) == true &*&
                   snd(p) == P_Get(Right, fst(p)) &*&
                   P_le(m, Right);
    };
```

This definition is not as easy to reason about as we might expect. In particular, some properties of this definition of inclusion such as reflexivity are only provable under the additional assumption that the keys are distinct.

We proved that the models of cursors fulfill the constraint `distinct_keys` (defined also in VeriFast libraries) because keys are index positions in the array used to denote separated cells.

```
lemma void positions_distinct_keys(int index, list<purenode> m, int pos)
requires bdll(?tab, ?cap, ?fst, index, ?last, ?z, m) &*& node(tab+z, cap, ?tab0);
ensures bdll(tab, cap, fst, index, last, z, m) &*& node(tab+z, cap, tab0) &*&
        distinct_keys(positions(m, index, pos)) == true;
```

Notice that these proofs are not necessary for provers with support for finite maps and sets. Although VeriFast supports as back-end solver Z3 [8], it does not use it for such theories. The inductive theories are supported by other back-end solvers, e.g., CVC4 [29] that are not connected to VeriFast.

3.3 Annotations Load

We include all the required (contracts) and auxiliary annotations (loop invariants, open/close of predicates, definitions and calls of lemma) in the source of the container proved by the solver. The prover (VeriFast with redux solver) includes all this annotation burden, since we can not direct the prover in the usage of these annotations. VeriFast provides two mechanisms to limit the burden of required annotations: (i) lemmas can be marked as automated which means they will be given to the backend solver on all problems, (ii) inductive predicate definitions can be automatically folded and unfolded when used with computed parameters.

We introduce few automated lemma and the introduced lemma are called in annotation to lighten the prover load. We don't observe performance problems by including all these annotations and despite the absence of modular proofs. The frame reasoning rule of Separation Logic seems to play an important role in this good behavior.

We found useful the two ways of specifying inductive predicates in VeriFast: by case on the model or by case over the aliasing of heap locations. We started with the first style, but finally chose the second to bring advantages of automatic folding and unfolding of computed predicates.

3.4 Challenges dealt

To resume, we faced the following challenges during the verification process:

- We considered a functional specification which is already in use in client code. Therefore, we can not adapt this specification to ease the verification. Instead, we propose a method based on a refined specification based on a precise model of the container that eases the verification and allows to obtain the initial specification with minimal cost.
- The specification we received is complete with respect to the model of containers and cursors. This requires to specify logic functions and predicate that are more complex than the usual ones.
- The code has been designed to obtain efficient container implementation and does not focus on verification. Therefore, the verification task has been more difficult compared with previous work verifying functional specification of container libraries [27,38] designed with verification in mind.
- Only specifications of contracts for public operations on the container were provided. We had to annotate the code and the internal operations. This implied an additional cost in annotations because some internal operations break the data structure invariants.
- Having in mind the extension of this verification effort to other bounded container libraries (for sets or maps), we propose reusable logic libraries and suggest some improvements for the auto-active verification tool in use.

4 Verification Results

Bugs found We don't find spectacular bugs in code, which is normal for a library that has been used for years. We only detect a potential arithmetic overflow in the computation of the memory to be allocated and a potential memory shortage. The last problem is in fact dealt for the SPARK clients using tools that measure the memory allocated by the program.

Complete specifications We also fix some minor completeness problems with the original specifications. Our verification effort leads to a complete functional specifications for all operations, including non public operations.

Specification load We have coded, specified, and verified 27 functions out of the 39 provided by the container library including equality and emptiness tests, clear, assign and copy, getting and setting one element, manipulating the cursors, inserting and deleting at some cursor, finding an element before and after a cursor. Most of the 12 remaining functions deal with sorting. The size of our development is given in Table 1. To obtain a specification close to the Ada 2012 one, we wrote two files of logic definitions for models (`vfseq.gh` and `vfmap.gh`) extending the

Table 1. Statistics on the proof

File	*#pred*	*#fix-points*	*#lemma*	*lines*	
				annot	*code*
`vflist.gh`	2	8	9	234	–
`vfseq.gh`	14	10	34	486	–
`vfmap.gh`	12	9	57	1133	–
`cfdlli.h`	4	10	0	321	37
`cfdlli.c`	14	5	65	2328	396
Total	46	42	165	4502	433

VeriFast libraries. Additional fixpoint functions and lemmas required on VeriFast lists are written in file `vflist.gh`. The rate between source code and annotations is about 1 for 8. The required annotations (i.e., data structure invariant, pre/post conditions, and logic predicate and function used directly in them) represent a quarter of all annotations (including also loop invariants and lemmas). In Ada 2012 container, the rate between source code and contracts is already of about 1 for 3.

Verification performance We run VeriFast on a machine with 16GB RAM, Intel core i5, and 2.70 GHz, installed with Linux. The back-end solver of VeriFast was redux. The verification takes 1.3 seconds for the full container.

5 Related Work

The verification of individual data structure has received special attention. General safety properties (i.e., absence of out of array bounds accesses, null dereferences, division by zero, arithmetic overflow) may be verified automatically with low load of annotations using *static analysis methods*, e.g. [12,18,20,16]. More complex properties like reachability of locations in the heap and shape of the data structures could also be proved with static analysis methods based on shape analysis, e.g., [31,5,4,9]. These automatic techniques have been applied to linked lists coded in arrays [33]. These methods concern limited properties and may be used in the early stages of the library development to infer internal invariant properties. Extension of fully automatic techniques to cover functional specification abstractions like sets or bags are based either on *shape analysis*, e.g., [6,13] or on logic fragments supported by SMT decision procedures [17,15,36,37]. These functional specifications capture essential mathematical properties of the data structure but do not deal with properties of iterators over them.

At the opposite on the spectrum of verification techniques, *interactive provers* have been used to obtain detailed specifications about data structures based on powerful theories, e.g., [7,22,28], but they require expertise and great amount of proof scripting.

At the intermediate level of automation, functional verification tools have been used to tackle the verification of specific data structures (e.g., Dafny [32], GRASShoper [25], VeriFast [14], or Why3 [35]) but we are not aware of any experiment on bounded lists.

The full functional correctness of container libraries has been considered in [38,27]. They consider complex data structures in imperative and object oriented languages that require to verify special properties and may benefit from modular verification thanks to inheritance. In both cited works, a special effort has been deployed to improve the prover to call solvers for different theories or to generate verification conditions that may be dealt efficiently. This efforts lead to a low annotation overhead, especially in [27]. We use an on-the-shelf auto-active verification tool but improve its performances by employing a refinement method which leads to more automation but a more important annotation overhead. None of these works consider the container of bounded list.

6 Conclusion

We apply auto-active verification provided by the VeriFast tool to prove the functional specifications of the bounded doubly linked list container. The implementation we consider is in C, but it mimics the GNAT library [11], which is used in SPARK client programs. The functional specification is model-based and uses sequence and map mathematical models in a specific way to model the content of the list and its valid cursors. Our main contributions are (i) the improvement of the logic libraries of VeriFast to deal with such specific models and (ii) the use of a refinement based methodology to ease the proof automation.

This case study provides a motivation for the development of inductive solvers and their connection with auto-active provers like VeriFast. This experiment is another demonstration of the known fact (see [27]) that proving functional specifications of real world containers is more difficult than proving functional specification of data structures. The support for automation of these proofs is of an utmost importance to scale the verification to a full library of containers.

Acknowledgements: We thank Claire Dross and Yannick Moy from AdaCore for guiding us through the Ada standard library and for supplying the last version of its specification. We thank Samantha Dihn for the first C version of the Ada containers.

References

1. Ada Europe. Ada Reference Manual - Language and Standard Libraries, Chapter A.18.3 The Generic Package `Containers.Doubly_Linked_Lists` Norm ISO/IEC 8652:2012(E), 2012. Available online at `http://www.adaic.org/resources/add_content/standards/12rm/html/RM-TTL.html`.
2. AdaCore. SPARK verification gallery. Available at `https://github.com/AdaCore/spark2014/tree/master/testsuite/gnatprove/tests`.
3. P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.
4. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, Dec. 2011.
5. B. E. Chang and X. Rival. Relational inductive shape analysis. In *Proceedings of POPL*, pages 247–260. ACM, 2008.
6. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006 – 1036, 2012. The Programming Languages track at the 24th ACM Symposium on Applied Computing (SAC'09).
7. A. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceeding of ICFP*, pages 79–90. ACM, 2009.
8. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedigns of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

9. C. Dragoi, C. Enea, and M. Sighireanu. Local shape analysis for overlaid data structures. In *Proceedings of SAS*, volume 7935 of *LNCS*, pages 150–171. Springer, 2013.
10. C. Dross, J. Filliâtre, and Y. Moy. Correct code containing containers. In *Proceedings of TAP*, volume 6706 of *LNCS*, pages 102–118. Springer, 2011.
11. GNU Fundation. GNAT library components in gcc 7.1. Available at `https://sourceware.org/svn/gcc/tags/gcc_7_1_0_release/gcc/ada/` files `a-cfdlli.ad*`.
12. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of PLDI*, pages 339–348. ACM, 2008.
13. S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. *Property-Directed Shape Analysis*, pages 35–51. Springer, Cham, 2014.
14. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
15. S. Jacobs and V. Kuncak. *Towards Complete Reasoning about Axiomatic Specifications*, pages 278–293. Springer, Berlin, Heidelberg, 2011.
16. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *FAC*, 27(3):573–609, 2015.
17. V. Kuncak, R. Piskac, P. Suter, and T. Wies. *Building a Calculus of Data Structures*, pages 26–44. Springer, Berlin, Heidelberg, 2010.
18. V. Laviron and F. Logozzo. Subpolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. *STTT*, 13(6):585–601, Nov 2011.
19. K. R. M. Leino and M. Moskal. Usable auto-active verification. Available at `fm.csl.sri.com`, 2010.
20. J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *Proceedings of VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.
21. J. W. McCormick and P. C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.
22. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *Proceeding of ICFP*, pages 229–240. ACM, 2008.
23. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
24. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, pages 268–280. ACM, 2004.
25. R. Piskac, T. Wies, and D. Zufferey. *Automating Separation Logic with Trees and Data*, pages 711–728. Springer, Cham, 2014.
26. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *Proceedings of VSTTE*, LNCS, pages 127–141, Berlin, Heidelberg, 2010. Springer.
27. N. Polikarpova, J. Tschannen, and C. A. Furia. A fully verified container library. In *Proceedings of FM*, volume 9109 of *LNCS*, pages 414–434. Springer, 2015.
28. F. Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of CPP*, pages 3–16. ACM, 2017.
29. A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Proceedings of VMCAI*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.
30. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74. IEEE, 2002.

31. N. Rinetzky, S. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of SAS*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005.
32. K. Rustan and M. Leino. Main microsoft research dafny web page. Available at `http://research.microsoft.com/en-us/projects/dafny`.
33. P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *Proceedings of APLAS*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012.
34. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *Proceedings of POPL*, pages 199–210. ACM, 2010.
35. Why3-Team. Why3 verification gallery. Available at `http://toccata.lri.fr/gallery/`.
36. T. Wies, M. Muñiz, and V. Kuncak. *An Efficient Decision Procedure for Imperative Tree Data Structures*, pages 476–491. Springer, Berlin, Heidelberg, 2011.
37. T. Wies, M. Muñiz, and V. Kuncak. *Deciding Functional Lists with Sublist Sets*, pages 66–81. Springer, Berlin, Heidelberg, 2012.
38. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Proceedings of PLDI*, pages 349–361. ACM, 2008.