Verified Implementation of the Bounded List Container with VeriFast

Raphaël Cauderlier and Mihaela Sighireanu

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

Abstract. Containers are ubiquitous in programming but few container libraries have yet been formally verified. Therefore, a trend in software verification is providing fully verified container libraries. We contribute in this paper by presenting a verified implementation of the doubly linked list container which manages the list in fixed size, heap allocated arrays. Our 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. We chose C in order to benefit from the top tools and results in verification of programs manipulating the heap and specified in Separation Logic. We employ the VeriFast tool which provides a high performance deductive verification environment for Separation Logic extended with abstract data types theories. The specifications we obtain entail the contracts of the Ada library and include new features. In particular, we employ a specification method based on a precise model of the data structure which facilitates the verification and captures entirely the library contracts.

1 Introduction

One of the main goals of the standard libraries of programming languages is to provide efficient implementations for common data structures. To abstract the details of these implementations, modern programming languages propose generic interfaces to data containers. These interfaces are informally specified in terms of well understood mathematical structures such as sets, multisets, sequences, and partial functions. The intensive use of these libraries makes important their formal verification.

However, the functional correctness of these libraries is challenging to verify for several reasons. First, their implementation is highly optimized at a very low level, it employs complex data structures and manages the memory directly through pointers/references or ah-hoc allocators. Second, although the container interface is defined based on classic operations on mathematical structures, few standard libraries provide a formal specification of its containers. Notable exceptions are, e.g., Eiffel [10] and SPARK [4]; recently, [2] provided a specification of Ada 2012 standard library. Such formal specifications are very important when the library uses constructs that exceed the scope of the mathematical structure. For example, iterators in Java are generic and can exist outside the container,

while they are part of the container in Ada. Third, 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 tools for deductive verification.

In this work, we focus on the *formal doubly linked lists* container, which is a GNAT implementation of the doubly linked lists container in the standard library of Ada 2012 [1]. The implementation is compatible with SPARK 2014 and "meant to facilitate the formal verification of code using this container" [7]. 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. Due to the introduction of formal contracts in Ada 2012, the library has been specified recently[1] based on a previous specification in Why3 by Dross et al. [4]. However, the formal verification of the library can not be done in SPARK because it employs language features (e.g., low level memory manipulation) not allowed in SPARK. In addition to its industrial use and the availability of its formal specification, the implementation of bounded lists exhibits interesting invariant properties. It combines a memory allocator keeping, in addition to the doubly linked list a singly linked list for the free memory regions.

This implementation choice guided us to use Separation Logic [11] (SL), a formalism that permits a precise modelling of the heap and pointer management. The absence of tools for deductive reasoning in SL on Ada determined us to translate the Ada code into C, for which powerful deductive verification tools exists. We employ VeriFast [8], which allows to specify precisely the organisation of heap fragments using inductive predicates defined by SL formulas. Moreover, VeriFast provides means to combine these inductive predicates with polymorphic algebraic data types, which is a feature playing an important role in this work.

Indeed, to accurately translate the container contracts from Ada, we have to follow the approach of *representation predicates* introduced for SL by O'Hearn et al. [9] and generalised in several works, e.g., by Arthur Charguéraud [3]. It consists of relating heap fragments with detailed algebraic models using inductive predicates. The algebraic models are then constrained in the pure part of SL formulas, i.e., the part not constraining the heap. We show that this approach is expressive enough to obtain a faithful translation of Ada contracts for our library. Moreover, we show that the specifications obtained are more precise with respect to heap organisation and ease the deductive verification process, i.e., writing the lemmas.

To summarise, our contribution is (1) a fully verified implementation in C of a bounded doubly linked list, (2) a contribution to the VeriFast predefined library with predicates and lemmas which are useful for the verification of this implementation in any imperative programming language, (3) a practical demonstration of the power of representation predicates for both expressiveness of specifica-

[1] The annotated version will be published in June 2017.

tions and deductive verification. The sources of this contribution are available at `http://vecolib.imag.fr/index.php/Deliverables`.

The paper begins by stating, in Section 2, the principles we adopt for the coding and specification of the bounded lists container. Then, we detail the case study and highlight the main ingredients of its verification in Section 3. We end by presenting the experimental results and the lessons learnt from this work in Section 4.

2 Principles of Coding and Specification

This work belongs to a research project whose aim is to demonstrate that the existing technology in deductive verification based on Separation Logic is qualified to prove functional correctness of libraries of containers designed for critical code. In this paper, we provide evidence for the library provided by GNAT [7] for doubly linked lists (DLL). This library, coded in Ada 2012, implements DLL with bounded capacity, dynamic size, and fixed size of stored data (elements). The functional specification of this library is given by method contracts, written by Emmanuel Briot and Claire Dross from AdaCore [2], using SPARK 2014, the subset of Ada which is designed for programming safety-critical applications. The logic fragment used in these contracts is first order logic with functional abstract data types.

Because the tools for deductive verification in Ada or SPARK do not (yet) include the Separation Logic technology, we turn to C, or more precisely to its subset supported by the verification environment VeriFast [8]. We have coded in a subset of C the linked list container and translated its functional specification in the logic fragment supported by VeriFast. This process was possible because both the C programming language and the VeriFast logic work at a lower level of abstraction than Ada. However, because of differences between programming and specification languages involved, the translation is not a routine as explained in the remainder of this section.

Coding in C: The implementation in Ada employs constructs that can not be directly translated in C, mainly object oriented features (e.g., genericity, type constructor, default initialisation), and the restricted types (e.g., range restrictions on integers). For example, the Ada container is generic on the type of the elements stored in list cells; in C, we have coded the type of elements by a record with one integer field. [How severe are the semantic differences between the Ada and the C? C code complex aliasing patterns such as lists of lists.] The type of the container is parameterized by an integer giving its capacity; we have coded it by an additional field in the record type defining the container. The restricted integer types in Ada are translated to their base type (usually integer) and the range restrictions are imported in the specification as pre-conditions or type invariants. These coding principles can be used to obtain the C implementation of other formal containers available within GNAT, e.g., hashed sets and maps.

The coding principles faced some restrictions of VeriFast. For example, working with arrays of records in VeriFast requires to transform array access to

pointer arithmetics, e.g., `a[i].next` in Ada is translated to the equivalent code `(a+i)->next`. Also, only call by reference is allowed for record parameters.

Specification in VeriFast: The specification in Ada is "model" based: it constrains the *models* on which are mapped the values of the container and of its valid cursors[2]. [How faithful is the encoding of the specification? Pure function in VeriFast, impure only in SPARK.] As expected, the model of the doubly linked list container is the sequence of the values stored; the model of its valid cursors is a mapping from valid cursor values to positions in the sequence of elements. The interesting point is that these models are themselves written in Ada: the GNAT library includes "functional" containers for sequences and maps, i.e., simpler, less efficient implementations. The models are obtained from the values using two functions coded in Ada, `Model` resp. `Positions`. The aim of such functional models is to obtain executable specifications, which is important for testing tools. However, for deductive verification in SPARK, the aforementioned models are mapped to the corresponding algebraic data types in order to call inductive reasoning or SMT solvers.

It is natural to import in VeriFast the model based characteristic of the functional specification, due to the ability to define *ghost* algebraic data types and *representation predicates*, i.e., inductive predicates on the heap which compute abstract data type models. Moreover, VeriFast provides libraries of polymorphic ADTs for lists and association lists together with operations (`fixpoints` in VeriFast), predicates, and lemmas which are useful for inductive reasoning. The use of representation predicates (explained in the next section) introduces an important difference between the two specifications. In VeriFast, the model of the container is more precise, and allows us to obtain the model of cursors from it, without passing by the container. We consider that this is an important design choice, which simplifies the writing of annotations and the verification.

The translation of contract cases requires some adjustment. Indeed, SPARK contracts may define distinct behaviours of methods and the specification of our container uses such contracts intensively, as well as relational contracts that link the input value of a function parameter to its output value. Contract cases and relational contracts are not supported by VeriFast. However, we are able to translate them faithfully thanks to conditional formulas. Another adjustment is done in the import of Ada contracts calling Ada functions which do not belong to the functional models, because VeriFast does not allow such specifications. We import them by inserting their contracts.

Notice that the Ada implementation contains ghost code, mainly assertions, and code which is not specified. For example, the internal invariants of the container type are dynamically checked using ghost code. All these invariants are statically checked in our C implementation by the VeriFast annotations. We

[2] In Ada 2012, "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`." [1]

provide contracts for all methods not specified in the Ada code as required by the modular analysis of VeriFast.

3 Dynamic Bounded Doubly-Linked Lists

3.1 List container

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. We assume that elements are integers. In Ada, the type of the element is an abstract type, parameter of the library.

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

The predicate `node` specifies a node allocated on heap and its model as follows:

```
inductive purenode = purenode(int, int, Element_Type);

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

The inductive type `purenode` record the values of node fields. To specify that the node at location `n` is allocated on the heap, we use the VeriFast predicate `malloc_block_Node_Type` generated automatically at the definition of the record `Node_Type`. The values stored by the record fields are introduced (using question marks) by *points-to* atoms, e.g., `n->elem|->?pelem`. The value stored in the `next` link is bounded by the parameter `capacity`. The parameter `pn` is a *computed* parameter, i.e., deducible from the other parameters of the predicate. The constraints on the heap (also called *spatial*) and on values (also called *pure*) are composed by the conjunction operator `&*&` which represents both the separating conjunction and the logic conjunction.

There are two kinds of nodes: 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. Occupied nodes have the `prev` field set to a non-negative integer and the `elem` field is relevant. We define model functions (introduced by `fixpoint`) and derived predicates using them to identify the two kinds of nodes.

```
fixpoint bool is_free(purenode 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(purenode n) { return pprev(n)>=0; }
```

```
predicate occupied_node(struct Node_Type* n, int capacity;
                        purenode pn) =
  node(n, capacity, pn) &*& is_occupied(pn)==true;
```

Acyclic doubly linked list The container stores the doubly linked list (DLL) in 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 DLL are stored starting from the index 1; index 0 plays the role of the null reference. Therefore, 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;
};
```

Cells at the extremity of the lists are stored at indexes `first` resp. `last`. In Ada, the fields of the container are all initialized, the array of nodes allocated, and the types used for indexes are constrained. We shift these constraints in the C function playing the role of the container constructor and in the representation predicate of the container which is used to specify function preconditions.

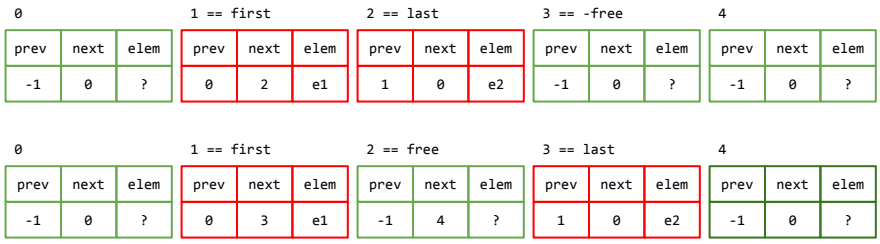

Fig. 1. Two doubly linked lists of capacity 4 and length 2

The representation predicate of the list of occupied nodes is defined classically as a list segment starting at node `ifrom`, ending at node `ilast` and linked with the previous node `iprev` and the following node `ito`:

```
predicate dll(struct Node_Type * tab, int capacity,
              int iprev, int ifrom, int ilast, int ito;
              list<purenode> values) =
  ifrom==ito ?
    (iprev==ilast &*& values==nil<purenode>)
  : (occupied_node(tab+ifrom, capacity, ?p) &*&
     pprev(p)==iprev &*&
     dll(tab, capacity, ifrom, pnext(p), ilast, ito, ?vtl) &*&
     values==cons(p, vtl));
```

where `pprev`, `pnext`, and `pelem` return the previous, next, resp. element component of a `purenode` value. This form of the predicate for acyclic DLLs has good

properties and eases the generation of lemmas for composition of list segments required to prove code doing list traversal [6,3].

The model built for the DLL is more precise than the one built for the container in Ada, since it also includes the links of cells. This design choice is very important for the verification process because this precise model allows to obtain both models used in Ada (i.e., the sequence of values and the map of cursors) while keeping the predicate compositional. We have also tried a solution, close to the Ada specification, where the predicate computes the two models (sequence and map). This solution generates a lot of annotations and requires more lemmas.

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` as follows: if `free` is negative, the list is built from all nodes stored between $-$`free` and `capacity`; otherwise, the list starts at index `free` and uses as successor relation the `next` field of nodes. Figure 1 illustrates the two kinds of a free list. The first kind of free list is used to obtain a fast method to initialize the nodes in the free-list (mainly `next` fields, all set to 0) at the DLL initialization.

The representation predicates of the two kinds of free-lists define the model of free-list as the sequence of indexes of free nodes:

```
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));
```

We choose a uniform shape for these predicates as list segments in order to exploit the lemmas satisfied by such predicates [6,3] for (de)composition of list segments. The two kinds of free-list are combined in a predicate that instantiates the correct predicate depending on the sign of `free`.

```
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);
```

Representation predicate The invariants of the container are specified by the following predicate:

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

The predicate specifies that the container is allocated on the heap, 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 link fields at -1 and 0. The remaining nodes are split between the `dll` and `free_nodes` predicates due to the separating conjunction. The size of the DLL is exactly the one of its model and stored in the field `size`.

In VeriFast, allocated blocks are implicitly translated as arrays of bytes (predicate `chars`). To access the i-th node of `nodes` inside the block allocated for it, we introduce an axiom that intuitively says that if an array of chars at address `tab+i` is not empty, we can split it such that we identify a node at its start:

```
lemma void open_malloc_block(struct Node_Type* tab,
                             int i, int len)
  requires len > 0 &*&
    chars((void*)(tab+i),len*sizeof(struct Node_Type),_) ;
  ensures  malloc_block_Node_Type(?n) &*& n == tab+i &*&
    n->prev|-> _ &*& n->next|-> _ &*& n->elem|-> _ &*&
    chars((void*)(tab+i+1),(len-1)*sizeof(struct Node_Type),_);
{ assume(false); }
```

Examples of function contracts The contract of the constructor of the list provides the default values for the fields of the container:

```
struct List* List(int capacity);
//@ requires capacity > 0;
//@ ensures list(result, ?tab, capacity, -1, _, 0, 0, nil);
```

The contract for the list equality test illustrates how the precise model (variables `mL` and `mR`) is used to extract the sequence of values by the logic function `model`:[Use fractions for aliasing and read-sharing]

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

For some methods, the contract we specify is more precise than in Ada due to the lack of encapsulation of container fields. For example, for the method `clear`, the Ada contract specifies only that, at the end of the method, the list has an empty model. In our contract, the values of fields for list head and tail are constrained.

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

3.2 Cursors

Cursors are used to mark positions in the list. We implement them as a record storing an array index. The special cursor `No_Element` is defined as a global constant storing index 0.

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

Following this implementation, the model of a cursor is given by a value of inductive type `cursor`.

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

The predicate `pre_valid_cursor_or_noelem` captures the semantics of a cursor storing an integer `index`. It computes from `m`, the precise model of the DLL, and `first`, its starting index, the model of the cursor and the two list segments `before` and `after`, into which `C` splits `m`. Notice that it is a pure predicate and may be duplicated in a specification for the need of verification.

Given a DLL container, there is a bijection between valid cursor models and the positions in the list. This bijection is modeled in our specification by an association list. We specify a library of logic functions, predicates, and lemmas required by the annotations of this model of cursors. One of these functions is `positions`, which builds the map model from the pure model of the list and the index of the first element of the list. It is used to specify the contracts of methods using cursors. For example, the function `element` returning the value stored at the position in the list given by the cursor `C` has the following contract:

```
Element_Type element(struct List* L, struct Cursor* C)
/*@ requires list(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(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)); @*/
```

where the logic function `P_Has_Key` tests if the model of `C`, i.e., `pc`, is a key in the map of valid cursors, and `P_Get` returns the exact position of this cursor. The function `M_Element` returns the element, in the sequence of values, `model(m)`, at the position of this cursor.

[How useful are the specifications? Helpful to verify a client program. Notice the abstract definition of `list` predicate in .h.]

4 Experiments and Lessons Learnt

We have coded, specified, and verified 22 main methods from the 30 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. The size of our development is given below. To obtain a specification close to the Ada one, we wrote two files of logic definitions for models (`vfseq.gh` and `vfmap.gh`) extending the 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 6; in Ada, the rate between source code and contracts is already of about 1 for 3.

Table 1. Statistics on the proof

File	*#pred*	*#fix-points*	*#lemma*	*lines*	
				annot	*code*
`vflist.gh`	2	7	8	177	–
`vfseq.gh`	14	10	17	355	–
`vfmap.gh`	12	3	2	185	–
`cfdlli.h`	0	4	0	319	45
`cfdlli.c`	16	4	40	1473	385
Total	42	28	67	2524	430

When specifying the private functions for node (de)allocation (specifications not provided in Ada), we obtain very big contracts of nearly 20 atoms, because these methods break the invariant of the container. However, they are both used with code such that the sequence of these calls restore the container invariant. Collapsing of these methods may reduce the size of annotations.

The choice of a precise model for the specification of the container has two main advantages: it facilitated the writing of lemma for composition of list segments and it allowed to manipulate several abstractions of the container, including the sequence of values stored, the map of valid cursors, and the container size. All these abstractions are catamorphisms on the precise model, so easy to be defined with VeriFast fix-points and enabling efficient decision procedures [12].

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 computed predicates. In conclusion, VeriFast provided us all the tools required to specify and verify this library. We encountered some troubles with the use of arrays of records, solved by pointer arithmetics and the axiom discussed in the previous section. We found very pleasant the application of automatic lemmas, which discharged most of the inductive reasoning on lists and arithmetics. We got

around the absence of contract cases by using conditional expressions and by expressing the relation between old and new values through the precise model.

[Related work: GrassHopper, Viper or Dafny, Why3.] [Related work: discuss Polikarpova et al.] This work is the first deductive verification we are aware of for this implementation of the container. VeriFast provides an example of an array list storing pointers, but which includes only simple functions (append and remove). Static analysers like Astrée have been applied to similar implementations but the capacity of the containeris fixed to a constant. Recently, a more complex container has been verified with SPARK, the bounded sets implemented with RBT [5]. However, this experiment did not consider the full implementation and omits cursors.

Acknowledgements: We thank Claire Dross 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. E. Briot and C. Dross. Generic Ada library for algorithms and containers. Github project, 2017. Available online at `https://github.com/AdaCore/ada-traits-containers`.
3. A. Charguéraud. Higher-order representation predicates in separation logic. In *Proceedings of CPP*, pages 3–14. ACM, 2016.
4. C. Dross, J. Filliâtre, and Y. Moy. Correct code containing containers. In *TAP*, volume 6706 of *LNCS*, pages 102–118. Springer, 2011.
5. C. Dross and Y. Moy. Auto-active proof of red-black trees in spark. In *NFM*, volume to appear of *LNCS*. Springer, 2017.
6. C. Enea, M. Sighireanu, and Z. Wu. On automated lemma generation for separation logic with inductive definitions. In *ATVA*, volume 9364 of *LNCS*, pages 80–96. Springer, 2015.
7. 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*`.
8. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In K. Ueda, editor, *Proceedings of APLAS*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.
9. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, pages 268–280. ACM, 2004.
10. N. Polikarpova, J. Tschannen, and C. A. Furia. A fully verified container library. In *Proceedings of FM*, volume 9109 of *Lecture Notes in Computer Science*, pages 414–434. Springer, 2015.
11. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74. IEEE, 2002.
12. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.