Logical Tools for the Refinement Relations

Deliverable D3-1

ANR project VECOLIB

March 2017, last revision August 2017

Abstract

This report surveys logic based formalisms used to specify the refinement relations between mathematical structures implemented by container libraries and the data structures used in these implementations. We propose a solution based on the combination of logic theories over inductive data types and Separation Logic. For this solution, the report surveys the existing tools (solvers, provers) developed to verify such specifications, and therefore check the refinement relations.

1 Motivation

One of the main goals of the standard libraries for containers is to provide efficient implementations for common mathematical structures such as sets, multisets, sequences, and partial functions. To abstract the details of these implementations, modern programming languages propose generic interfaces to data containers. In general, these interfaces are informally specified in terms of well known operations on the mathematical structures represented.

Few standard libraries provide a formal specification of its containers. Notable exceptions are, e.g., Eiffel [10] and Spark [4] which use first order logic over theories of sets and sequences. Recently, [2] provided a specification of Ada 2012 standard library using first order logic over user defined algebraic data types. Also, an ongoing project tries to provide a fully specified and verified implementation of Ocaml container library in Coq. 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 they exist outside the container, while they are part of the container in Ada.

In order to reason about the correctness of container implementations with respect to their abstract specifications, it is necessary to consider logic-based formalisms that allow to mix the high level logics used to specify the container interface and the low level logics used to specify the implementation. For example, let us consider the simplified case of a mixed specification $\varphi_H \oplus \varphi_L$ of the post-condition of a container operation, where φ_H is the high level specification of the postcondition, φ_L is the low level one, and some logic operation $\oplus$ is used to compose them. To show that the operation implementation is correct, one has to prove that the φ_L is valid in the final state of this operation. To prove the interface specification of the operation, one has to prove that φ_H implies the post-condition of the operation. This solution is employed by approaches for refinement specification where the high level and the low level logics are the same, e.g., second order logic on sets in B [1]. However, such approaches are efficient for programs using simple data structures but are less scalable for pointer manipulating programs.

Because our tools for the lower level are based on Separation Logic, we cannot take a uniform approach as above, because Separation Logic is not well adapted for high level specification. In the VECOLIB proposal, we sketch out a solution based on an explicit refinement relation α linking the high level and low level, as follows. Consider for instance an implementation of sets which uses acyclic singly linked lists. The refinement relation α should specify that for any set variable s, there is a unique acyclic list segment in the heap, i.e., a list segment between some memory location labeled by the head variable S and nil. If the lower level logic is the Separation Logic with an inductive definition of the list segment predicate ls, we may write $\alpha(S, s) \wedge ls(S, nil)$. The refinement relation shall also link the abstract terms to the implementation of abstract operations, e.g. $(\alpha(S, s) \wedge R = add(x, S)) \Rightarrow \alpha(R, Add(x, s))$, where add is the implementation of abstract data type operation Add.

The study of several existing approaches for specifying container libraries [10, 5] turned us to a solution where the refinement relation is not explicit, but included in the inductive definition of the data structure in Separation Logic.

This report is organised as follows. Section 2 surveys the existing approaches for specifying container libraries and for mixing Separation Logic with high level theories. Section 3 defines a preliminary solution we used to verify a container library. Section 4 surveys the tools available or required by the solution we proposed.

2 Specification of Container Libraries

2.1 Eiffel Container Library

The first successful experience in proving a full container library has been reported in 2015 by Polikarpova et al [] for EIFFEL. The choice of EIFFEL is not surprising because the programming language has been designed with the principle of programming by contracts, and therefore it ease the specification process. However, the heap manipulation in EIFFEL is simpler due to its object oriented design principle and voids pointer arithmetics. This simplifies the reasoning about the heap.

The experience does a complete verification of a realistic container library, called EiffelBase2, against full functional specifications. The library, consists of over $8,000$ lines of Eiffel code in 46 classes, and offers arrays, lists, stacks, queues, sets, and dictionaries. The interface specifications are written in first-order logic and characterize the abstract object state using mathematical entities, such as sets and sequences.

To specify the refinement relation, each class in EIFFEL declares its abstract state through a set of *model* attributes. For example, the model of the class `LINKED_LIST` is a sequence of list elements (of generic type `E`), denoted by an attribute (field) of the class as follows:

```
class LINKED_LIST[E]
  model sequence
  feature [public]
    ghost sequence: MML_SEQUENCE[E]
  feature [private]
    first_cell: LINKABLE [G]
    last_cell: LINKABLE [G]
    ghost cells: MML_SEQUENCE [LINKABLE [G]]
...
```

The sequence type `MML_SEQUENCE` is provided by a specific EIFFEL library, called the Mathematical Model Library (MML). The instances of MML model classes are mathematical values that are mapped to the corresponding logical representations in the underlying prover. The interface view of the class is a sequence of elements, denoted by the public attribute `sequence`. The link between the data structure used, i.e., a linked list starting from `first_cell` and ending in `end_cell`, and the model `sequence` is defined by the private attribute `cells` that stores the sequence of references (interface `LINKABLE`) to cells. The update of the model is preceded by the update of

the internal model representation, like below for the `push_back` operation for a value `v`:

```
cells := cells + <cell>
sequence := sequence + <v>
```

where `cell` is the new cell allocated to store `v`.

The verification conditions generated from the EIFFEL contracts of the container implementation are checked by a prover, AutoProof, that is designed to deal with the objects in MML. The prover has been significantly extended to deal with this case study, mainly with new theories and lemmas that ease the interactive proof.

2.2 Ada Container Library

The contract-based programming style has been added in ADA 2012 standard. This make possible the translation in ADA of the specifications for the standard container libraries, which where only written for SPARK (the formal, non standard version of ADA) as described in [4]. The translation, done at AdaCore during the VECOLIB project, lead to a new formal specified library of containers published in July 2017 [2].

For each container, the ADA specification is given as:

- a mapping M from a container type to a *model type*, which is a functional implementation of the container; model types are provided for sequences, sets and maps;
- for each container operation o, a contract $(pre_o, post_o)$ in the form of a pre/post condition using predicates and operations over the model type and the container type.

The specification also includes the specification of container "cursors", i.e., a data structure that allows to iterate over the elements of the container.

For example, for the container of bounded doubly linked lists, the model mapping M maps the container type into the sequence of its elements. Since the container is implemented using an array, the model of its cursors maps the positions in the list to indexes in the array.

An important feature of this specification is its *executability*. This feature provides a mean to check by testing the implementation and the specification but also to simplify the specification and the verification of the code using the library.

The model mapping M can be seen as a refinement relation. To verify its soundness, one has to check that all contract of operations provided over the container are satisfied. This task is not trivial because it requires to verify the implementation of the container. The specification of invariants for the data structure used to implement the container and the loop invariants in the operations are not provided.

During the VECOLIB project, we consider the verification of the new library of containers as it is described in Section 3. This experience leads us to the conclusion that, for the verification process, it is simpler to decompose the refinement relation M used in the ADA specification into two relations: (i) a first relation mapping the container type to a mathematical model that includes all the details of the implementation, also called the concrete mode, and (ii) a mapping from the previous model to the container model used in ADA specifications. For example, for the doubly linked lists container, we use as concrete model a sequence of records storing the cells of the list in the order given by the list (and not by the array storing the list). The mapping from this concrete model to the sequence of elements is inductively defined over sequences. This decomposition facilitates the proof of correctness because it reduces the task of the low level solver and uses the power of solvers dealing with mathematical theories, here the theory of sequences of data.

2.3 Specifying Refinement with VeriFast in C

For C, the VERIFAST [8] tool includes its ow specification language which allows to specify precisely the organisation of heap fragments using inductive predicates defined by Separation Logic formulas. In addition, VERIFAST provides means to combine these inductive predicates with polymorphic algebraic data types, which is a feature very important for this work because it can be used to specify the refinement relation.

Indeed, the invariants and the contracts of a container implementation can be captured container contracts and infrom 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., [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.

For example, let us consider the VERIFAST specification of a doubly linked list of integers stored in the heap and related with the model of sequences which is denoted by `list` in VERIFAST:

```
predicate dll(struct Node_Type* iprev, struct Node_Type* ifrom,
      struct Node_Type* ilast, struct Node_Type* ito;
              list<int> values) =
  ifrom==ito ?
    (iprev==ilast &*& values==nil<int>)
  : (malloc_block_Node_Type(ifrom) &*&
     ifrom->prev|->iprev &*& ifrom->next|->?inext &*&
     ifrom->elem|->?pelem &*&
     dll(ifrom, inext, ilast, ito, ?vtl) &*&
     values==cons(pelem, vtl));s
```

The type of the list is a record `struct Node_Type` with fields `next`, `prev`, and `elem` for the links to next and previous elements of the list and the integer element. The parameter `values` representing the algebraic model is after the semi-colon, which means that it is computed (i.e., completely fixed) by the inductive predicate from the first parameters. The constructors of the algebraic model `list` are `nil` and `cons`. The variables quantified existentially are prefixed by a query sign at their first occurrence in the formula. To specify an acyclic doubly linked list ending in null with all elements strictly positive, the formula includes the predicate above and constrains the algebraic model by the predicate `all_pos` as follows:

```
dll(null, S, ?L, null; ?seq) &*& all_pos(seq)
```

where `S` and `L` are pointers to the start resp. last element of the list, and the predicate `all_pos` is defined inductively by:

```
predicate all_post(list<int> s) =
  switch(s) {
    case nil: true;
    case cons(?x, ?s0): x > 0 &*& all_pos(s0);
  }
```

Another approach is for defining the refinement relation is to use the two step approach: define a sequence of records `Node_Type` using the predicate `dll` and then mapping this sequence to the sequence of integers representing its elements using a inductive defined function, called *fixpoint* in VeriFast specifications:

```
fixpoint list<int> values(list<Node_Type> s) =
  switch(s) {
    case nil: return nil<int>;
```

```
    case cons(?r, ?s0): return cons((&r)->elem, values(s0));
  }
```

This approach allows to relate the heap specification in SL with a very precise algebraic model over C types. This model is used to extract several abstractions of the list: the sequence of its values, the sequence of addresses of cells stored in the list, etc.

From the experience we conducted with the verification of the refinement relation for the container of doubly linked list (see next section), we conclude that this last approach is the most promising one for the expressive power of specifications and for the capabilities of logical tools, i.e., solvers or provers.

3 Verification of a Container Library

This section reports on the experience we conducted for the verification of the refinement relation for a container library of bounded doubly linked lists using VeriFast. A detailed report (paper submitted to VSTTE 2017) and the sources of the library are available as appendixes of this report.

The library we consider is inspired by the GNAT [7] library for doubly linked lists (DLL). It implements DLL with bounded capacity, dynamic size, and fixed size of stored data (elements). The functional specification of the Ada 2012 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 algebraic 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 (see the detailed report).

The specification is, like in Ada 2012 container, model based. This is possible due to the features provided by VeriFast to define ghost algebraic data types and representation predicates, i.e., inductive predicates on the heap which compute abstract data type models.

In the following, we sketch up the container implementation and specification. Its implementation is very interesting because it combines an allocator with a classic data structure for linked lists.

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`.

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].

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 **??** 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`:

```
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;
```

Experimental results: 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 [11].

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.

4 Logical Tools

The conclusion of this study is that a logic specification for the refinement relation suitable for container libraries implemented in heap-manipulating languages should satisfy the following requirements:

- include ghost specifications allowing to mix C types and algebraic data types,
- include means to specify lemmas and additional functions or predicates used to map concrete models to abstract models,
- include the Separation Logic theory with the ability of defining inductive predicates mixing SL with algebraic data types.

The specification language of VeriFast satisfies these requirements.

The expressive power of the logic specification is balanced by the undecidability of the logic theories involved as discussed in our previous reports. To obtain automatic tools for the verification of the refinement relation, we are working on sound procedures for the satisfiability and entailment problems in the theory of SL combined with algebraic or abstract data types. An example of such work is the procedure proposed in [6] for the theory of SL combined with multi-set contraints.

References

[1] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[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. Abstract software specifications and automatic proof of refinement. In *Proceedings of RSSRail*, volume 9707 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2016.

[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] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.