Correct Code Containing Containers

Claire Dross[1], Jean-Christophe Filliâtre[2], and Yannick Moy[1]

[1] AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{dross,moy}@adacore.com
[2] CNRS, INRIA Saclay Île-de-France, Université Paris-Sud 11
filliatre@lri.fr

Abstract. Containers like lists, vectors, sets or maps are an attractive alternative to ad-hoc data structures based on pointers for critical software development. Containers offer both a better defense against errors than low-level code manipulating pointers, and a rich high-level API to express properties over data. As standards like DO-178C put formal verification and testing on an equal foot, it is important to give users the ability to apply both to the verification of code using containers. This is the motivation for the definition of containers presented in this paper, whose correct usage and user-provided correctness properties can be checked either by execution during testing or by formal proof with an automatic prover. The key objectives in this work were minimizing changes w.r.t. the existing standard version and maximizing the numbers of proofs performed automatically, subject to the key constraint of keeping it simple to write annotations for users.

Keywords: containers, iterators, verification by contracts, annotations, axiomatization, API usage verification, SMT, automatic provers

1 Introduction

Containers[3] are generic data structures offering a high-level view of collections of objects, while guaranteeing fast access to their content to retrieve or modify it. The most common containers are lists, vectors, sets and maps, which are defined in the standard libraries of most languages, like C++ STL, Ada Standard Libraries or Java JCL. Containers are so profundly useful for expressing program properties that some academic languages included them as language elements [15]. In critical software where verification objectives severely restrict the use of pointers, containers offer an attractive alternative to pointer-intensive user data structures. This is particularly evident when the implementation of containers themselves obeys the coding standards of critical software, with no dynamic allocation and few pointers, like it is the case for the bounded containers defined in Ada 2012 [3].

Standards for critical software development define comprehensive verification objectives to guarantee the high levels of dependability we are expecting

[3] We use the word container as a synonym for a homogeneous collection, not to mean an aggregate containing heterogeneous objects as in component architectures.

of life-critical and mission-critical software. All requirements must be shown to be satisfied by the software, which is a costly activity. In particular, verification of low-level requirements is usually demonstrated by developing unit tests, from which high levels of confidence are only obtained at a high cost. This is the driving force for the adoption of formal verification on an equal foot with testing to satisfy verification objectives. The upcoming DO-178C avionics standard says no less: *Formal methods [..] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.*

Although there are some areas where formal verification can be applied independently [16], most areas where testing is the main source of evidence today would benefit from an integration of formal verification with the existing testing practice. At the simplest, this combination should be provably sound and it should guarantee a coverage of atomic verifications through formal verification and testing. This is the goal of project Hi-Lite, a project aiming at combined unit testing and unit proof of C and Ada programs.

In the context of project Hi-Lite, this work deals with the definition of suitable containers in Ada, based on Ada standard containers, whose properties can be both tested dynamically and proved automatically. Properties over containers offer a high level description of the code, suitable for expressing partial correctness as code contracts. Therefore, we are not only interested in container API usage correction, but also in partial correctness properties of interest to users.

Before they can be used in the context of avionics (or similar) safety-critical software, the new library of *formal* containers will need be certified and the verification tools we present will have to undergo qualification [2]. We present in this paper: (a) a formal proof of correctness of an implementation of the new library in Coq [1], a well-known formal programming language; and (b) a formal proof of the new library properties used in our verification tools, expressed in the Why language [11] for formal verification. Thus, this work can be seen as a contribution to the argument-based approach to certification [14].

In Section 2, we detail the changes that we introduce in formal containers w.r.t. standard containers. In the following sections, we describe in details formal containers for doubly linked lists, and we sketch formal containers for vectors, sets and maps: formal semantics in Section 3, formal specification in the Why language in Section 4, formal proof of correctness in Coq in Section 5. We finally give a survey of related works in Section 6.

2 Formal Containers

We will use the following Ada code as a running example throughout the section. Procedure `Map_F` modifies a list in place, by replacing each element initially contained in the list by its image through function `F`.

```
procedure Map_F (L : in out List) is
   Current : Cursor := First (L);
begin
```

```
   while Current /= No_Element loop
      Replace_Element (L, Current, F (Element (Current)));
      Next (Current);
   end loop;
end Map_F;
```

2.1 Contracts in Ada 2012

The next version of the Ada standard, called Ada 2012, offers a variety of new features to express properties of programs. New checks are defined as *aspects* of program entities, for which the standard defines precisely the various points at which the check is performed during execution. The most prominent of these new checks are the *Pre* and *Post* aspects which define respectively the precondition and postcondition of a subprogram. These are defined as Boolean expressions over program variables and functions. Additionally, the expression in a postcondition can refer to the value returned by a function `F` as `F'Result`, and to the value in the pre-state (before the call) of any variable `V` as `V'Old`. For example, a simple contract on function `Map_F` could specify that its parameter list should not be empty, and that the call does not modify its length:

```
procedure Map_F (L : in out List) with
  Pre  => Length (L) /= 0,
  Post => Length (L) = Length (L'Old);
```

Notice that `L` in the precondition refers to the list in the pre-state while `L` in the postcondition refers to the list in the post-state, hence the need to refer to the special attribute `L'Old` in the postcondition, which designates the value of `L` before the call.[4] When compiled with checks on, any failure from the caller to respect the precondition of `Map_F` or from `Map_F` to respect its postcondition will be detected at run-time and reported as an error. The execution model for these aspects is simply to insert assertions at appropriate locations, which raise exceptions when violated. For each variable `V` whose pre-state value may be read in the postcondition (`V'Old`), the compiler inserts a shallow copy of the variable's value at the beginning of the subprogram body. It is this copy which is read to give the value of `V'Old`.

Expressing properties in contracts is greatly facilitated by the use of if-expressions, case-expressions, quantified-expressions and expression-functions, all defined in Ada 2012. The main objective of these *logic* features is verification by testing, based on their executable semantics. In particular, quantified-expressions are always expressed over finite ranges or (obviously finite) containers, with a loop through the range or container as execution model: `for all J in 0 .. 10 => P (J)` is true if-and-only-if the subprogram `P` returns `True` for every argument, starting from 0 up to 10.

[4] More precisely, attribute `'Old` can be applied to all *names* (as defined in Ada standard), so that we could use `Length (L)'Old` instead of `Length (L'Old)`.

2.2 Ada Standard Containers

Like in many other languages, Ada standard containers define two mutually dependent data structures: containers proper which hold the data, and iterators (or cursors) which provide handles to individual pieces of data. In function `Map_F`, the container has type `List` and the iterator has type `Cursor`, which are both defined by the standard Ada lists. A cursor is implicitly associated to a container (implemented as a pointer field in the cursor type), in which it designates an element. An important feature of Ada standard containers is that cursors remain valid as long as the container exists and the element referenced is not deleted[5], like many iterators in other languages (for example those in C++ STL and Java JCL). This allows modifying a container while iterating over its content with cursors, without risking invalidating these cursors.

As seen in the example, a cursor `Cu` can be used for three different purposes: accessing a designated element of the container through `Element (Cu)`; iterating over the content of a container `L`, by starting at the first cursor in the container, given by `First (L)`, and repeatedly reaching to the next cursor, given by `Next (Cu)`; modifying a container like in the call to `Replace_Element` in the example. A designated cursor called `No_Element` is the default invalid cursor (it designates no element).

2.3 API Modification: Independent Cursors

Problem. A useful postcondition for `Map_F` is to state how contained elements are modified by the call. All cursors are preserved through replacement of an element in a list. Thus, for every cursor `Cu` that designates an element `E` in `L` before the call, `Cu` designates `F (E)` in `L` after the call. It seems like we could express it with a quantified-expression:

```
procedure Map_F (L : in out List) with
  Post => (for all Cu in L => Element (Cu) = ???);
```

The expression denoted `???` should designate the value obtained by calling `F (Element (Cu))` in the pre-state, which could be intuitively written as `F (Element (Cu))'Old`. Unfortunately, this expression is not valid because it refers to the value in the pre-state of `Cu`, which is not defined in the pre-state. As a side note, notice that, even for a cursor `Cu` defined outside of the function, `Cu'Old` would be the same as `Cu` in our example, because the semantics of attribute `'Old` is to perform a shallow copy, so it does not copy the implicit container in a cursor.

Idea. To solve the previous two problem, we separate cursors from their container, so that a cursor can be used both in the pre-state and in the post-state. Then, the previous postcondition can be expressed easily:

[5] This is true for lists, sets and maps. Cursors of vectors behave slightly differently, but they are much less used, as vectors are best accessed through indexes.

```
procedure Map_F (L : in out List) with
  Post =>
   (for all Cu in L => Element (L, Cu) = F (Element (L'Old, Cu)));
```

Notice that we passed an additional argument to function `Element`, to indicate the container in which the element at cursor `Cu` should be retrieved. This is true for every function in the API of containers which previously accessed implicitly the container through a cursor, such as `Next` in the example.

This is the only modification to the API of containers that we introduce. The alternative of using existing containers both greatly complicates the execution model and the formal verification.

Using existing containers requires a different semantics for the `'Old` attribute, which would reach to the complete pre-state including the stack and heap, similar to the work by Kosiuczenko on Java programs [13] that builds a complete history of updates alongside execution. Our solution has the benefits of sticking to the standard semantics for `'Old` in Ada 2012, leading to a simple and efficient execution model implemented in the GNAT compiler.

The semantics of standard cursors also leads to more complex verification conditions to check the correct use of containers' API: each access through cursor `Cu` to container `Co` is valid only if 1) `Co` is alive, which can be hard to know if `Co` is implicit, and 2) `Cu` is associated to container `Co`, which amounts to deciding whether `Co` is the same as the implicit container in `Cu`. With the semantics of formal containers, both verification conditions above disappear.

As seen from the example, the impact on user code is minimal, easy to automate, leading to more readable programs. Assuming the constraint that a cursor is never used to index two different containers, which is almost always true in code (but not in annotations, as shown in our example), it will be possible to translate automatically programs that use formal containers into programs that use standard containers, should the need arise. This translation would require computing the container associated to each cursor by static analysis.

2.4 API Addition: Parts of Containers

Problem. In order to prove the postcondition of `Map_F` stated above, we need to annotate the loop in `Map_F` with a loop invariant, which states the accumulated effect of N iterations through the loop. We would like to state that *list elements already scanned* have been modified by `F` (like in the postcondition) and that *list elements not yet scanned* are unchanged. This pattern of loop invariant, consisting of two parts for the elements scanned and the elements not yet scanned, is typical of loops that iterate over a container.

Idea. We introduce two new functions, called `Left` and `Right`, which return the containers containing respectively the elements preceding (exclusively) or following (inclusively) a given cursor in the container. With these functions, the effect of the loop on the elements already scanned resembles the postcondition:

```
pragma Assert
  (for all Cu in Left (L, Current) =>
    Element (L, Cu) = F (Element (L'Old, Cu)));
```

The effect of the loop on the elements not yet scanned is not simply the equality of right containers, which would be expressed as:

```
pragma Assert (Right (L, Current) = Right (L'Old, Current));
```

Indeed, the equality operator of the container API defines structural equality, which is not enough here. We need to know that the right containers contain the same elements *and the same cursors*. This is expressed with a new function, called `Strict_Equal`. The invariant above becomes:

```
pragma Assert
  (Strict_Equal (Right (L, Current), Right (L'Old, Current)));
```

3 Formal Semantics

In this section, we present a pen-and-paper formal semantics for lists. We show briefly how the formal semantics of other containers compares to the one of lists in the last subsection.

3.1 Syntax of Lists

A program is a sequence of variable declarations for lists (in `Lvar`) and cursors (in `Cvar`) followed by a sequence of instructions (in `Instr`). Procedures *Insert*, *Delete* and *Replace_Element* modify their argument list, which must be therefore a variable, and have no return value. The remaining instructions are assignments. Notice that list assignment makes an explicit copy of its argument, which prevents aliasing between lists. `LExpr` is the set of list expressions. *Empty* is the empty list constant. Functions *Left* and *Right* return parts of containers as defined in Section 2.4. `CExpr` is the set of cursor expressions. *No_Element* is the constant invalid cursor. Functions *First*, *Last*, *Next* and *Previous* are used for iterating over lists. `EExpr` is the set of element expressions. Function *Element* accesses the element designated by a cursor in a list. `BExpr` is the set of Boolean expressions. *Has_Element* checks for the validity of a cursor in a given container, = is the structural equality and *Strict_Equal* is the more constraining equality described in Section 2.4. Finally, `IExpr` is the set of integer expressions. Function *Length* returns the length of a list.

$$
\begin{array}{lll}
\texttt{Instr} := & \mathit{Insert}(\texttt{LVar}, \texttt{CExpr}, \texttt{EExpr}) \\
\mid & \mathit{Delete}(\texttt{LVar}, \texttt{CVar}) \\
\mid & \mathit{Replace\_Element} \\
 & \quad (\texttt{LVar}, \texttt{CExpr}, \texttt{EExpr}) \\
\mid & \texttt{Cvar} := \texttt{CExpr} \\
\mid & \texttt{Lvar} := \mathit{Copy}(\texttt{LExpr}) \\
\mid & \ldots \\
\texttt{LExpr} := & \texttt{Lvar} \\
\mid & \mathit{Empty} \\
\mid & \mathit{Left}(\texttt{LExpr}, \texttt{CExpr}) \\
\mid & \mathit{Right}(\texttt{LExpr}, \texttt{CExpr}) \\
\texttt{IExpr} := & \mathit{Length}(\texttt{LExpr}) \\
\mid & \ldots
\end{array}
\qquad
\begin{array}{lll}
\texttt{CExpr} := & \texttt{Cvar} \\
\mid & \mathit{No\_Element} \\
\mid & \mathit{First}(\texttt{LExpr}) \\
\mid & \mathit{Last}(\texttt{LExpr}) \\
\mid & \mathit{Next}(\texttt{LExpr}, \texttt{CExpr}) \\
\mid & \mathit{Previous}(\texttt{LExpr}, \texttt{CExpr}) \\
\texttt{EExpr} := & \mathit{Element}(\texttt{LExpr}, \texttt{CExpr}) \\
\mid & \ldots \\
\texttt{BExpr} := & \mathit{Has\_Element}(\texttt{LExpr}, \texttt{CExpr}) \\
\mid & \texttt{LExpr} = \texttt{LExpr} \\
\mid & \texttt{CExpr} = \texttt{CExpr} \\
\mid & \mathit{Strict\_Equal}(\texttt{LExpr}, \texttt{LExpr}) \\
\mid & \ldots
\end{array}
$$

For the sake of simplicity, we only list instructions and expressions that are specific to containers. For example, we have not included loops or branching statements in the set of instruction, or arithmetic operations in the set of integer expressions.

3.2 Operational Semantics of Lists

Since lists are generic in the kind of element they contain, the type $\mathbb{E}$ of elements is unspecified. The type $\mathbb{D}$ of cursors can be any countably infinite type. We add a specific element $\perp$ to this set, $\mathbb{D} \cup \perp$ is written $\mathbb{D}^{\perp}$. There is an environment for lists Γ_L, and an environment for cursors Γ_C:

$$
\begin{array}{l}
\Gamma_L : \texttt{Lvar} \to \mathbb{L} = \{\mathit{Len} : \mathbb{N}, \mathit{Fc} : [1..\mathit{Len}] \hookrightarrow \mathbb{D}, \mathit{Fe} : \mathit{Im}(\mathit{Fc}) \to \mathbb{E}\} \\
\Gamma_C : \texttt{Cvar} \to \mathbb{D}^{\perp}
\end{array}
$$

Intuitively, lists can be seen as an array Fc of cursors with a mapping Fe from cursors to elements. Fc is injective, so Fc restrained to $Im(Fc)$ is bijective, and $Fc^{-1} : Im(Fc) \to [1..Len]$ is its inverse. We extend Fc^{-1} to $Fc_{+}^{-1} : Im(Fc) \cup \{\perp\} \to [1..Len+1]$ with $Fc_{+}^{-1}(\perp) = Len + 1$. Given an instruction I in `Instr`, a list l in `LExpr`, a cursor c in `CExpr`, an expression e in `EExpr`, a Boolean expression b in `BExpr` and an integer expression i in `IExpr`, judgments take the following form:

$$
\begin{array}{c}
\Gamma_L, \Gamma_C \vdash I \Rightarrow \Gamma_L', \Gamma_C' \qquad \Gamma_L, \Gamma_C \vdash l \Rightarrow \mathbb{L} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow \mathbb{D}^{\perp} \\
\Gamma_L, \Gamma_C \vdash e \Rightarrow \mathbb{E} \qquad \Gamma_L, \Gamma_C \vdash b \Rightarrow \mathbb{B} \qquad \Gamma_L, \Gamma_C \vdash i \Rightarrow \mathbb{Z}
\end{array}
$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\}}{\Gamma_L, \Gamma_C \vdash Length(l) \Rightarrow Len}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d \qquad d \in Im(Fc)}{\Gamma_L, \Gamma_C \vdash Element(l, c) \Rightarrow Fe(d)}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d}{\Gamma_L, \Gamma_C \vdash Has\_Element(l, c) \Rightarrow d \in Im(Fc)}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l1 \Rightarrow \{Len1, Fc1, Fe1\} \qquad \Gamma_L, \Gamma_C \vdash l2 \Rightarrow \{Len2, Fc2, Fe2\}}{\Gamma_L, \Gamma_C \vdash l1 = l2 \Rightarrow Len1 = Len2 \ \& \ Fe1 \circ Fc1 = Fe2 \circ Fc2}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l1 \Rightarrow \{Len1, Fc1, Fe1\} \qquad \Gamma_L, \Gamma_C \vdash l2 \Rightarrow \{Len2, Fc2, Fe2\}}{\Gamma_L, \Gamma_C \vdash Strict\_Equal(l1, l2) \Rightarrow Len1 = Len2 \ \& \ Fc1 = Fc2 \ \& \ Fe1 = Fe2}$$

$$\frac{\Gamma_L, \Gamma_C \vdash c1 \Rightarrow d1 \qquad \Gamma_L, \Gamma_C \vdash c2 \Rightarrow d2}{\Gamma_L, \Gamma_C \vdash c1 = c2 \Rightarrow d1 = d2}$$

$$\frac{}{\Gamma_L, \Gamma_C \vdash No\_Element \Rightarrow \bot}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d1 \qquad d1 \in Im(Fc) \cup \{\bot\}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Next(l, c) \Rightarrow d2 \\ where\ d2 = \{Len \neq 0 \ \& \ d1 \in Im(Fc) \backslash \{Fc(Len)\} \rightarrow Fc(Fc^{-1}(d1) + 1), else \rightarrow \bot\}\end{array}}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d1 \qquad d1 \in Im(Fc) \cup \{\bot\}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Previous(l, c) \Rightarrow d2 \\ where\ d2 = \{Len \neq 0 \ \& \ d1 \in Im(Fc) \backslash \{Fc(1)\} \rightarrow Fc(Fc^{-1}(d1) - 1), else \rightarrow \bot\}\end{array}}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\}}{\Gamma_L, \Gamma_C \vdash First(l) \Rightarrow d \qquad where\ d = \{Len = 0 \rightarrow \bot, Len > 0 \rightarrow Fc(1)\}}$$

$$\frac{\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\}}{\Gamma_L, \Gamma_C \vdash Last(l) \Rightarrow d \qquad where\ d = \{Len = 0 \rightarrow \bot, Len > 0 \rightarrow Fc(Len)\}}$$

$$\frac{}{\Gamma_L, \Gamma_C \vdash Empty \Rightarrow \{0, F_\varnothing, F_\varnothing\}}$$

$$\frac{\begin{array}{c}\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \\ \Gamma_L, \Gamma_C \vdash c \Rightarrow d \qquad d \in Im(Fc) \cup \{\bot\} \qquad n = Fc_+^{-1}(d) - 1\end{array}}{\Gamma_L, \Gamma_C \vdash Left(l, c) \Rightarrow \{n, Fc', Fe'\} \qquad where\ Fc' = Fc|_{[1..n]} \qquad Fe' = Fe|_{Im(Fc')}}$$

$$\frac{\begin{array}{c}\Gamma_L, \Gamma_C \vdash l \Rightarrow \{Len, Fc, Fe\} \\ \Gamma_L, \Gamma_C \vdash c \Rightarrow d \qquad d \in Im(Fc) \cup \{\bot\} \qquad n = Fc_+^{-1}(d) - 1\end{array}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Right(l, c) \Rightarrow \{Len - n, Fc', Fe'\} \\ where\ Fc' = \lambda i : [1..Len - n].Fc(n + i) \qquad Fe' = Fe|_{Im(Fc')}\end{array}}$$

Above are the description of the semantics of expressions. The result of function *Length* on a list evaluating to $\{Len, Fc, Fe\}$ is *Len*. Similarly, function *Element* returns the value of *Fe* on d, where cursor argument c evaluates to d. Notice that $Element(l, c)$ is defined only when $d \in Im(Fc)$, which is expressed in the informal semantics as *c designates an element in l*. Indeed, the associated Ada function will raise a run-time error otherwise. $Has\_Element(l, c)$ checks if

c effectively designates an element in l. Equality over lists (=) is the structural equality. It only implies that the elements in its two list arguments appear in the same order, *i.e.,* the equality of $Fe \circ Fc : [1..Len] \rightarrow \mathbb{E}$. *Strict_Equal* is stronger than =, as expected, since it also implies the equality of Fc and Fe. Equality of cursors is simply equality of their evaluations.

The five following rules describe the semantics of cursor expressions. The special invalid cursor *No_Element* evaluates to $\perp$. This is possible because $\perp$ cannot appear in $Im(Fc)$, as $Fc : [1..Len] \rightarrow \mathbb{D}$. Therefore $\perp$ is not a *valid* cursor, *i.e.,* it designates no element, in any list. Function *Next* is defined for both valid cursors and *No_Element*. It returns *No_Element* when applied to a cursor which has no valid successor (*i.e.,* for *No_Element* and the last cursor). *Previous* is similar. Function *First* is defined on every list. It returns *No_Element* when called on an empty list. *Last* is similar.

The last three rules describe the semantics of list expressions. The empty list, returned by *Empty*, is the only list whose length is null ($F_\varnothing$ is the only function that is defined on the empty set $\varnothing$). *Left* is defined for both valid cursors and *No_Element*. Its evaluation yields a list whose valid cursors are the valid cursors of the list argument which precede cursor argument c (when c is *No_Element*, that means all cursors). *Right* is similar.

The rules bellow describe the semantics of instructions. Rules concerning reads or assignment of variables are omitted (they are the usual ones). *Insert* modifies the environment so that its list variable argument designates, after the call, a list where a cursor and an element have been inserted at the proper place. The cursor argument, which can be either a valid cursor or *No_Element*, encodes the place the new element is inserted. The newly created cursor is not specified. It should be different from *No_Element* and from every valid cursor in the argument list. *Delete* modifies the environment so that its cursor variable argument (which must reference a valid cursor before the call) is deleted from the list referenced by its list variable argument. The cursor variable references the special invalid cursor *No_Element* after the call. *Replace_Element* modifies the environment so that, after the call, its cursor argument (which must be valid) designates its element argument in the list referenced by its list variable argument.

$$\frac{\begin{array}{c}\Gamma_L(l) = \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d1 \\ d1 \in Im(Fc) \cup \{\bot\} \qquad n = Fc_{+}^{-1}(d1) \qquad d2 \notin Im(Fc) \cup \{\bot\} \qquad \Gamma_L, \Gamma_C \vdash e \Rightarrow Elt\end{array}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Insert(l, c, e) \Rightarrow \Gamma_L[l \mapsto \{Len + 1, Fc', Fe'\}], \Gamma_C \\ \textit{where } Fc' = \lambda i : [1..Len + 1]. \\ \{i \in [1..n-1] \rightarrow Fc(i), i = n \rightarrow d2, i \in [n+1..Len+1] \rightarrow Fc(i-1)\} \\ Fe' = \lambda d : Im(Fc').\{d \in Im(Fc) \rightarrow Fe(d), d = d2 \rightarrow Elt\}\end{array}}$$

$$\frac{\Gamma_L(l) = \{Len, Fc, Fe\} \qquad \Gamma_C(c) = d \qquad d \in Im(Fc) \qquad n = Fc^{-1}(d)}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Delete(l, c) \Rightarrow \Gamma_L[l \mapsto \{Len - 1, Fc', Fe'\}], \Gamma_C[c \mapsto \bot] \\ \textit{where } Fc' = \lambda i : [1..Len - 1].\{i \in [1..n-1] \rightarrow Fc(i), i \in [n..Len-1] \rightarrow Fc(i+1)\} \\ Fe' = Fe|_{Im(Fc')}\end{array}}$$

$$\frac{\Gamma_L(l) = \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash c \Rightarrow d1 \qquad d1 \in Im(Fc) \qquad \Gamma_L, \Gamma_C \vdash e \Rightarrow Elt}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Replace\_Element(l, c, e) \Rightarrow \Gamma_L[l \mapsto \{Len, Fc, Fe'\}], \Gamma_C \\ \textit{where } Fe' = \lambda d : Im(Fc).\{d = d1 \rightarrow Elt, d \neq d1 \rightarrow Fe(d)\}\end{array}}$$

3.3 Vectors, Sets and Maps

Sets. Sets do not allow duplication of elements, the order of iteration in a set is not user-defined and the link between cursors and elements is preserved in most cases. In the semantics, sets can be modeled as the same tuples as lists where Fe is injective: $\{Len : \mathbb{N}, Fc : [1..Len] \hookrightarrow \mathbb{D}, Fe : Im(Fc) \hookrightarrow \mathbb{E}\}$. For non ordered sets, the order of iteration is not specified.

$$\frac{\begin{array}{c}\Gamma_L(l) = \{Len, Fc, Fe\} \qquad d1 \notin Im(Fc) \cup \{0\} \\ \Gamma_L, \Gamma_C \vdash e \Rightarrow Elt \qquad Elt \notin Im(Fe) \qquad Fc' : [1..Len + 1] \hookrightarrow Im(Fc) \cup \{d1\}\end{array}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Insert(l, e) \Rightarrow \Gamma_L[l \mapsto \{Len + 1, Fc', Fe'\}], \Gamma_C \\ \textit{where } Fe' = \lambda d : Im(Fc').\{d \in Im(Fc) \rightarrow Fe(d), d = d1 \rightarrow Elt\}\end{array}}$$

If the set is ordered, the order of iteration is constrained by the order over elements. As a consequence, function $Fe \circ Fc$ has to preserve order.

$$\frac{\begin{array}{c}\Gamma_L(l) = \{Len, Fc, Fe\} \qquad d1 \notin Im(Fc) \cup \{\bot\} \qquad \Gamma_L, \Gamma_C \vdash e \Rightarrow Elt \\ Elt \notin Im(Fe) \qquad n = (max(\{i \in [1..Len] | Fe(Fc(i)) < Elt\} \cup \{0\})) + 1\end{array}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Insert(l, e) \Rightarrow \Gamma_L[l \mapsto \{Len + 1, Fc', Fe'\}], \Gamma_C \\ \textit{where } Fc' = \lambda i : [1..Len + 1]. \\ \{i \in [1..n-1] \rightarrow Fc(i), i = n \rightarrow d1, i \in [n+1..Len+1] \rightarrow Fc(i-1)\} \\ Fe' = \lambda d : Im(Fc').\{d \in Im(Fc) \rightarrow Fe(d), d = d1 \rightarrow Elt\}\end{array}}$$

Maps. Maps behave just like sets of pairs key/element except that they only constrain keys: $(k1, e1) < (k2, e2) \leftrightarrow k1 < k2$ and $(k1, e1) = (k2, e2) \leftrightarrow k1 = k2$.

Vectors. Vectors do not expect cursors to keep designating the same element in every case. Instead, like in arrays, elements can be accessed through their position (an index). As a consequence, we model vector as tuples $\{Len : \mathbb{N}, Fc : [1..Len] \hookrightarrow \mathbb{D}, Fe : [1..Len] \rightarrow \mathbb{E}\}$ where Fc is injective. When an element is inserted or deleted from a vector, nothing can be said for the cursors that follow the place of insertion/deletion.

$$\frac{\begin{array}{c}\Gamma_L(l) = \{Len, Fc, Fe\} \qquad \Gamma_L, \Gamma_C \vdash i \Rightarrow n \\ n \in [0..Len] \qquad \Gamma_L, \Gamma_C \vdash e \Rightarrow Elt \qquad Fc_S \Rightarrow [n+1..Len+1] \hookrightarrow \mathbb{D} \backslash Im(Fc|_{[1..n]})\end{array}}{\begin{array}{c}\Gamma_L, \Gamma_C \vdash Insert(l, i, e) \Rightarrow \Gamma_L[l \mapsto \{Len+1, Fc', Fe'\}], \Gamma_C \\ \text{where } Fc' = \lambda j : [1..Len+1]. \\ \{j \in [1..n] \rightarrow Fc(j), j \in [n+1..Len+1] \rightarrow Fc_S(j)\} \qquad Fe' = \lambda j : [1..Len+1]. \\ \{j \in [1..n] \rightarrow Fe(j), j = n+1 \rightarrow Elt, j \in [n+2..Len+1] \rightarrow Fe(j-1)\}\end{array}}$$

4 Axiomatization

In this section, we present an axiomatization of lists in the language Why, targeted at automatic provers. We show that this axiomatization is correct w.r.t. the pen-and-paper formal semantics we gave in Section 3. We will formally prove it is correct w.r.t. formal semantics in Coq in Section 5.

4.1 Presentation of Why

The Why platform [11] is a set of tools for deductive program verification. The first feature of Why is to provide a common frontend to a wide set of automated and interactive theorem provers. Why implements a total, polymorphic, first-order logic, in which the user can declare types, symbols, axioms and goals. These goals are then translated to the native input languages of the various supported provers. In our case, we are using the backends for the Coq proof assistant [1] and the three SMT solvers Alt-Ergo [7], Z3 [8] and Simplify [9]. Here is for instance some Why input syntax to declare a modulo operation over integers, together with some possible axiomatization:

```
logic mod_ : int, int -> int
axiom mod__ : forall a, b : int. 0 < b ->
  exists q : int. a = b * q + mod_(a,b) and 0 <= mod_(a,b) < b
goal test : mod_(7,2) = 1
```

The second feature of Why we are using here is to provide a verification condition generator for an idealized, alias-free, Hoare-logic-like programming language. The user declares and implements programs, which are annotated with pre- and postconditions, and local assertions such as loop invariants. Verification conditions are then generated and transmitted to the theorem provers. Here is for instance the declaration of a program function

```
parameter mod : a:int -> b:int -> {0<b} int {result=mod_(a,b)}
```

which takes two integers `a` and `b` as arguments, has precondition `0<b`, returns a result of type `int` and has postcondition `result=mod_(a,b)`. When used inside programs, this function will trigger verification conditions (namely, that its second argument is positive). This is one way to express that modulo is a partial operation.

4.2 Axiomatization of Lists

Note that this axiomatic is not meant to be the exact translation of our semantics, but rather to facilitate the verification of programs with the automatic prover Alt-Ergo.

Types. The type of elements is indifferent, so it is defined as an abstract type `element_t`. Cursors and lists are described respectively with abstract types `cursor` and `list`, further axiomatized in the following.

Properties. To encode the semantics defined in Section 3, we introduce three logic functions:

```
logic length_   : list -> int
logic position_ : list, cursor -> int
logic element_  : list, cursor -> element_t
```

Logic functions `length_` and `element_` define accessors to the fields Len and Fe of a list. The encoding is more complex for the field Fc, due to the fact that Fc is used in three different ways in the specification: 1) directly, 2) through its inverse Fc^{-1}, and 3) through its domain $Im(Fc)$. Function `position_` is the extension of Fc^{-1} to cursors not in $Im(Fc)$, whose image is set to 0. It gives access to both $Im(Fc)$ ($c \in Im(Fc) \Leftrightarrow$ `position_(l, c) > 0`) and Fc^{-1}. We rewrite almost all rules of the semantics to remove occurrences of Fc. For example, in the rule for $Next$, $d2 = Fc(Fc^{-1}(d1)+1)$ can be rewritten as $Fc^{-1}(d2) = Fc^{-1}(d1)+1$. The only rule that cannot be translated that way is =. For this one rule, we can use an existential quantification, $\forall\ d1 : Im(Fc1).\exists\ d2 : Im(Fc2).Fc1^{-1}(d1) = Fc2^{-1}(d2)\ \ \&\ \ Fe1(d1) = Fe2(d2)$. Why functions `length_`, `element_` and `position_` are related to the semantics as follows:

$$\forall\ \texttt{l},\ \texttt{i},\ \texttt{length\_(l) = i} \Leftrightarrow \exists\ Len,\ Fc,\ Fe,\ \Gamma_L, \Gamma_C \vdash \texttt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ Len = \texttt{i}$$

$$\begin{aligned}\forall\ \texttt{l},\ \texttt{c},\ \texttt{i},\ \texttt{position\_(l, c) = i} \Leftrightarrow\ & \exists\ Len,\ Fc,\ Fe,\ d,\\ & \Gamma_L, \Gamma_C \vdash \texttt{l} \Rightarrow \{Len, Fc, Fe\}\ \&\ \Gamma_L, \Gamma_C \vdash \texttt{c} \Rightarrow d\\ & \&\ \texttt{i} \geq 0\ \&\ (\texttt{i} = 0 \rightarrow d \notin Im(Fc))\\ & \&\ (\texttt{i} > 0 \rightarrow d \in Im(Fc)\ \&\ Fc^{-1}(d) = \texttt{i})\end{aligned}$$

$$\begin{aligned}\forall\ \texttt{l},\ \texttt{c},\ \texttt{e},\ \texttt{position\_(l, c) > 0} \rightarrow\ & \\ \texttt{element\_(l, c) = e} \Leftrightarrow\ & \exists\ Len,\ Fc,\ Fe,\ d,\ Elt,\\ & \Gamma_L, \Gamma_C \vdash \texttt{l} \Rightarrow \{Len, Fc, Fe\}\ \&\ \Gamma_L, \Gamma_C \vdash \texttt{c} \Rightarrow d\\ & \&\ \Gamma_L, \Gamma_C \vdash \texttt{e} \Rightarrow Elt\ \&\ Fe(d) = Elt\end{aligned}$$

Axioms. We encode the semantic properties of functions `length_`, `element_` and `position_` into axioms, while ensuring that the axiomatic is not unnecessarily restrictive (all semantic lists should be also axiomatic lists). We have four axioms:

1. ∀ `l, length_(l) >= 0`
2. ∀ `l, c, length_(l) >= position_(l, c) >= 0`
3. ∀ `l, position_(l, no_element) = 0`
4. ∀ `l, c1, c2, position_(l,c1) = position_(l,c2) > 0` → `c1 = c2`

It is rather straightforward to check that these axioms are implied by the semantics. We show this translation for the second axiom.

We start with rewriting the axiom to introduce a fresh variable for each application:

∀ `l, c, il, ip, length_(l) = il &  position_(l, c) = ip` → `il >= ip >= 0`.

We can then translate the two equalities thanks to the equivalences:

$$\begin{array}{l}\forall\ \mathtt{l},\ \mathtt{c},\ \mathtt{il},\ \mathtt{ip},\\ \quad (\exists\ Len,\ Fc,\ Fe,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ Len = \mathtt{il})\ \ \&\\ \quad (\exists\ Len,\ Fc,\ Fe,\ d,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L, \Gamma_C \vdash \mathtt{c} \Rightarrow d\\ \quad\quad \&\ \ \mathtt{ip} \geq 0\ \ \&\ \ (\mathtt{ip} = 0 \rightarrow d \notin Im(Fc))\\ \quad\quad \&\ \ (\mathtt{ip} > 0 \rightarrow d \in Im(Fc)\ \ \&\ \ Fc^{-1}(d) = \mathtt{ip}))\quad \rightarrow\\ \quad\quad\quad \mathtt{il} \geq \mathtt{ip} \geq 0\end{array}$$

This can be simplified, by splitting $\mathtt{ip} \geq 0$ into $\mathtt{ip} = 0$ and $\mathtt{ip} > 0$:

$$\begin{array}{l}\forall\ \mathtt{l},\ \mathtt{c},\\ \quad (\exists\ Len,\ Fc,\ Fe,\ d,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L, \Gamma_C \vdash \mathtt{c} \Rightarrow d\ \ \&\\ \quad\quad Len \geq 0 \geq 0\ \ \&\ \ (d \in Im(Fc) \rightarrow Len \geq Fc^{-1}(d) \geq 0)\end{array}$$

The three other axioms may be translated in the same way. Here are the results:

1. $\forall\ \mathtt{l},\ \exists\ Len,\ Fc,\ Fe,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ Len \geq 0$
2. $\forall\ \mathtt{l},\ \mathtt{c},\ \exists\ Len,\ Fc,\ Fe,\ d,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L, \Gamma_C \vdash \mathtt{c} \Rightarrow d\ \ \&\ \ (d \in Im(Fc) \rightarrow Len \geq Fc^{-1}(d) \geq 0)$
3. $\forall\ \mathtt{l},\ \exists\ Len,\ Fc,\ Fe,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ 0 \notin Im(Fc)$
4. $\forall\ \mathtt{l},\ \mathtt{c1},\ \mathtt{c2},\ \exists\ Len,\ Fc,\ Fe,\ d1,\ d2,\quad \Gamma_L, \Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L, \Gamma_C \vdash \mathtt{c1} \Rightarrow d1\ \ \&\ \ \Gamma_L, \Gamma_C \vdash \mathtt{c2} \Rightarrow d2\ \ \&\ \ d1 \in Im(Fc)\ \ \&\ \ d2 \in Im(Fc)\ \ \&\ \ Fc^{-1}(d1) = Fc^{-1}(d2) \rightarrow d1 = d2$

Semantic Rules. Each Ada function represented in the semantics is translated into a Why program. Given the semantic rule $\frac{Pre_S}{Post_S}$ defining this function, we define a precondition Pre_A and a postcondition $Post_A$ on the Why program, so that $Pre_A \Rightarrow Pre_S$ and $Post_S \Rightarrow Post_A$. We illustrate the general pattern that we applied with function *Next*, which we translate into program `next` in Why:

```
parameter next :
  l:list -> c:cursor ->
  { c = no_element or position_(l, c) > 0 }
  cursor
  { result = next_(l, c) }
```

The precondition of this function states that either the argument cursor `c` is valid in the argument list `l` (because `position_(l, c) > 0` $\Leftrightarrow c \in Im(Fc)$) or the argument cursor is equal to *No_Element*. This precondition is exactly the condition for the application of the semantic rule for *Next*.

An axiom `next__` defines the behaviour of logic function `next_` over the allowed cases only, leaving the value of other applications of `next_` unspecified, as seen in Section 4.1:

```
axiom next__ :
  forall l:list. forall c:cursor. forall nxt:cursor.
    (length_(l) > position_(l,c) > 0 ->
      position_(l, nxt) = position_(l, c) + 1)
    and (length_(l) > 0 and position_(l, c) = length_(l)
      or c = no_element ->
        nxt = no_element)
```

Axiom `next__` is defined as two implications: in the first case, the next cursor of the cursor argument is valid and we define its position; in the second case, the result is *No_Element*. Intuitively, this is the same as the semantic rule for *Next*. Using the same equivalences as above, we can rewrite axiom `next__` into a logic formula that is exactly the semantics of *Next*, rewritten to use only Fc^{-1}. Like in the previous paragraph, we start by introducing one fresh variable per function application and we then use the traductions of each construct to obtain :

$$
\begin{array}{l}
\forall\ \mathtt{il},\ \mathtt{ic},\ \mathtt{in}, \\
\quad (\exists\ Len,\ Fc,\ Fe,\quad \Gamma_L,\Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ Len = \mathtt{il})\ \ \& \\
\quad (\exists\ Len,\ Fc,\ Fe,\ dc,\quad \Gamma_L,\Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L,\Gamma_C \vdash \mathtt{c} \Rightarrow dc \\
\qquad \&\ \ \mathtt{ic} \geq 0\ \ \&\ \ (\mathtt{ic} = 0 \rightarrow dc \notin Im(Fc)) \\
\qquad \&\ \ (\mathtt{ic} > 0 \rightarrow dc \in Im(Fc)\ \ \&\ \ Fc^{-1}(dc) = \mathtt{ic}))\ \ \& \\
\quad (\exists\ Len,\ Fc,\ Fe,\ dn,\quad \Gamma_L,\Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\ \ \&\ \ \Gamma_L,\Gamma_C \vdash \mathtt{nxt} \Rightarrow dn \\
\qquad \&\ \ \mathtt{in} \geq 0\ \ \&\ \ (\mathtt{in} = 0 \rightarrow dn \notin Im(Fc)) \\
\qquad \&\ \ (\mathtt{in} > 0 \rightarrow dn \in Im(Fc)\ \ \&\ \ Fc^{-1}(dn) = \mathtt{in})) \rightarrow \\
\qquad\quad \exists\ Len,\ Fc,\ Fe,\ dc,\ dn\quad \Gamma_L,\Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\} \\
\qquad\qquad \&\ \ \Gamma_L,\Gamma_C \vdash \mathtt{c} \Rightarrow dc\ \ \&\ \ \Gamma_L,\Gamma_C \vdash \mathtt{nxt} \Rightarrow dn\ \ \& \\
\qquad\qquad\quad (\mathtt{il} > \mathtt{ic} > 0 \rightarrow \mathtt{in} = \mathtt{ic} + 1)\ \ \& \\
\qquad\qquad\quad (\mathtt{il} > 0\ \ \&\ \ \mathtt{ic} = \mathtt{il}\ ||\ dc = 0 \rightarrow dn = 0)
\end{array}
$$

It can be rewritten as:

$$\begin{aligned}&\exists\ Len,\ Fc,\ Fe,\ dc,\ dn,\quad \Gamma_L,\Gamma_C \vdash \mathtt{l} \Rightarrow \{Len, Fc, Fe\}\\ &\&\ \Gamma_L,\Gamma_C \vdash \mathtt{c} \Rightarrow dc\ \ \&\ \ \Gamma_L,\Gamma_C \vdash \mathtt{nxt} \Rightarrow dn\ \ \&\\ &\qquad (dc \in Im(Fc)\ \ \&\ \ Len > Fc^{-1}(dc) \rightarrow\\ &\qquad\qquad dn \in Im(Fc)\ \ \&\ \ Fc^{-1}(dn) = Fc^{-1}(dc) + 1)\ \ \&\\ &\qquad (dc \in Im(Fc)\ \ \&\ \ Fc^{-1}(dc) = Len\ ||\ dc = 0 \rightarrow dn = 0)\end{aligned}$$

4.3 Effectiveness

It is worth noting that the main difficulty we faced, when developing the axiomatization presented above, was to match the somewhat fuzzy expectations of SMT automatic provers: some work best with large predicates and few axioms, some work best with many smaller axioms, *etc.* As a point of comparison, developing the axiomatization took us three times more efforts than the formal proof of correctness in Coq presented in Section 5.

We wrote a number of tests (30) to convince ourselves that the axiomatization of lists presented is effective. With a combination of provers we managed to prove all the verification conditions generated. Our running example is proved rather quickly (less than 1s per VC). To facilitate proofs, which impacts both provability and speed, we defined 15 lemmas. These are not a burden in the maintainability of the axiomatic and cannot introduce inconsistencies since they are also automatically proved.

4.4 Vectors, Sets and Maps

Here is a table referencing the work done for each container. It contains the size of its Why file (its number of lines), the number of lemmas given and the number of tests passed. The code and tests are available online [4].

Container	Lines	Lemmas	Tests	Container	Lines	Lemmas	Tests
List	352	15	30	Ordered Sets	506	30	38
Vectors	298	0	22	Hashed Maps	394	27	22
Hashed Sets	429	24	35	Ordered Maps	476	35	25

5 Validation of the Axiomatization

We have presented a formal semantics for containers in Section 3 and its axiomatization in Why in Section 4. We have shown a pen-and-paper proof that the axiomatization presented is correct w.r.t. formal semantics. Given the size of the axiomatization, such a manual proof may easily contain errors. In this section, we describe an implementation in Coq of the formal semantics of containers, and a formal proof of correctness of the Why axiomatization of lists w.r.t. the Coq implementation.

5.1 Coq Implementation and Proof for Lists

Types. Our proofs are generic in the type of element. For the representation of cursors, we use positive natural numbers to which we add 0 to model the special cursor $\bot$. For lists, we model the tuple with a functional list of pairs cursor-element. The field *Len* is the length of the functional list, the field *Fc* is the function that, for an integer $i \in [1..Len]$, returns the cursor of the pair at the i^{th} position in the list and *Fe* returns, for each cursor in $Im(Fc)$, its first association in the list. Therefore, a list of this kind always defines one and only one tuple $\{Len, Fc, Fe\}$.

```
Definition cursor : Set := nat.
Definition Rlist : Set := List.list (cursor*element_t).
```

To keep only tuples where *Fc* is injective and has value in $\mathbb{D}$, we constrain the functional lists with a predicate. This predicate states that every cursor that appears in a list is different from $\bot$ (positive) and does not appear again in the same list.

```
Fixpoint well_formed (l : Rlist) : Prop :=
   match l with
        nil => True
      | a :: ls =>    fst a > 0
                   /\ has_element ls (fst a) = false
                   /\ well_formed (ls)
   end.
Record list := {this :> Rlist; wf : well_formed this}.
```

The property of well formedness has to be preserved through every modification of the list. With the Coq lists restricted that way, there is one and only one list per tuple $\{Len : \mathbb{N}, Fc : [1..Len] \hookrightarrow \mathbb{D}, Fe : Im(Fc) \rightarrow \mathbb{E}\}$. The two representations are equivalent.

Axioms. Thanks to the `-coq` option of Why, we translate automatically our Why axioms into Coq. Then they can then be proved valid formally.

Semantic Rules. We have written an implementation for each construct of our language. Since Coq is a pure functional language, the instructions that modify their list argument, such as *Insert*, are modeled by a function that returns a new list. The implementations are as close as possible to the semantic of their corresponding construct. Since they are Coq functions, these implementations have to be total, so we complete them for the cases that are not described in the semantics. For example, the semantics of *Next* is only defined when the cursor that designates the element to be replaced is valid in the list or equal to *No_Element*. In the Coq implementation below, it returns *No_Element* if we are not in that case (we could have chosen any other return value).

```
Fixpoint next (l : Rlist) (cu : cursor) :=
   match l with
        nil => no_element
      | a :: ls =>
          if beq_nat (fst a) cu then first ls else next ls cu
   end.
```

The result of the semantic rule for *Next* is completely defined. It is easy to get convinced that, when the cursor given as a parameter is indeed valid in the list or equal to *No_Element*, the Coq function `next` indeed returns the appropriate cursor. We use this representation to prove formally that the contracts in our axiomatic are indeed implied by the semantics.

For nearly every rule of the semantics of lists, the result of the modification is completely defined in terms of the value of the arguments. The only rule that is not completely determined is *Insert*, since the value of the new cursor is not given. For the implementation, we define a function `new` that returns a possible cursor. To keep our proofs as general as possible, we took care to use only the properties of `new` that were defined in the semantic (*i.e.,* that the result of `new` $\notin Im(Fc) \cup \bot$) by enforcing it thanks to Coq's module system.

5.2 Vectors, Sets and Maps

All containers have the same Coq representation. Therefore, some parts of the proofs are shared. Sets and maps (ordered or not) share the same lemmas and heavily rely on those of lists. Vectors also rely on the lemmas of lists but less heavily (they are quite different). Here are the size of the files (the number of lemmas in each file) and the architecture.

<table>
<tr><td colspan="6">Raw Lists: 154 lemmas</td></tr>
<tr><td rowspan="2">Lists</td><td>Raw Vectors: 108 lemmas</td><td colspan="4">Raw Sets: 139 lemmas</td></tr>
<tr><td>Vectors</td><td>Hashed Sets</td><td>Ordered Sets</td><td>Hashed Maps</td><td>Ordered Maps</td></tr>
</table>

Like for the *Insert* rule for lists, every unspecified part of the semantic of every container is kept as general as possible thanks to a sealed Coq module that only allows proofs to use the specified parts. The whole proof is 16,000+ lines of Coq. All of it plus commented excerpts can be found online [4].

6 Related Work

Formal proof over containers is an active area of research. There are two important, complementary areas in this domain: certifying user code that uses containers while assuming that their implementation comply with their specification (what we are doing) and certifying that an implementation of containers indeed complies with its specifications.

One the one hand, Bouillaguet *et al.* [6] focus on verifying that a container's implementation indeed complies with its specifications. They use resolution based first-order theorem provers to verify that the invariants of data structures such as sets and maps are preserved when considering operations on their encodings as arrays and trees. Zee *et al.* [17] even presented the first verification of full functional correctness for some linked data structure implementations. Unlike Bouillaget *et al.*, they use interactive theorem provers as well to discharge their verification conditions. Since they aim at certifying an implementation once, it does not seem to be too heavy a burden.

On the other hand, Gregor and Schupp [12] focus, like we do, on the certification of user programs. They present an unsound static analysis of C++ programs using STL containers. They generate partially executable assertions in C++ to express the constraints over containers' usage, in particular a non-executable `foreach` quantifier to iterate over all objects of a given type in the current memory state. Blanc *et al.* [5] also work on certifying user code using the C++ STL containers. Just as we did in this work, they axiomatize the containers and then construct some preconditions (resp. postconditions) that are more (resp. less) constraining than those of the semantics. Their work is still substantially different from what we did since they only check that the containers are properly used and they have no annotation language to allow the user to express other properties. Dillig *et al.* [10] present a static analysis for reasoning precisely over the content of containers. In our industrial context, we cannot assume that we can analyze the code that fills the containers, like they do, so we need to rely instead on user annotations to provide constraints over the containers' content. This in turn leaves us the possibility to analyze user properties that the static analysis would not target naturally, like the sum of values in a container.

7 Conclusion

We have presented a library of *formal containers*, a slightly modified version of the standard Ada containers. The aim was to make them usable in programs annotated with properties of interest to the user, that can be both tested and formally proved automatically. Although we have limited experience with using this library, our experiments so far indicate that users can easily add executable annotations to their programs, leading to automatic proofs of rich properties with SMT prover Alt-Ergo. We are now looking forward to working with our industrial partners in project Hi-Lite to develop large use-cases with formal containers.

We have given a formal semantics for these containers, and we have proved that this semantics is consistent by implementing it in Coq. We have developed an axiomatization of these containers in the language Why, targeted at automatic proofs with SMT provers, and we have proved in Coq that this axiomatization is correct w.r.t. the formal semantics of containers. On the one hand, this formalization is an essential step towards an argument-based certification of the library of formal containers, for their use in safety-critical software development. On the other hand, the proof of correctness of the axiomatization used in automatic provers is a very strong assurance against inconsistencies in proofs, which are a sour point of formal methods in industry.

Formal containers have been included in the standard library of GNAT Ada compiler. This implementation in Ada should be correct w.r.t. the formal semantics in Coq, but we have not proved it formally. This is an interesting (but difficult) problem for the future.

References

1. The Coq Proof Assistant. `http://coq.inria.fr`.
2. DO-178B: Software considerations in airborne systems and equipment certification, 1982.
3. `http://www.ada-auth.org/standards/12rm/html/RM-TTL.html`.
4. `http://www.open-do.org/projects/hi-lite/formal-containers/`.
5. Nicolas Blanc, Alex Groce, and Daniel Kroening. Verifying C++ with STL containers via predicate abstraction. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 521–524, New York, NY, USA, 2007. ACM.
6. Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*, VMCAI '07, pages 74–88, Berlin, Heidelberg, 2007. Springer-Verlag.
7. Sylvain Conchon and Evelyne Contejean. The Alt-Ergo automatic theorem prover, 2008. `http://alt-ergo.lri.fr/`.
8. Leonardo de Moura and Nikolaj Bjørner. Z3, An Efficient SMT Solver. `http://research.microsoft.com/projects/z3/`.
9. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
10. Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 11th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, January 2011.
11. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
12. Douglas Gregor and Sibylle Schupp. STLlint: lifting static checking from languages to libraries. *Softw. Pract. Exper.*, 36:225–254, March 2006.
13. Piotr Kosiuczenko. An abstract machine for the old value retrieval. In *Proceedings of the 10th International Conference on Mathematics of Program Construction*, MPC '10, pages 229–247. Springer, 2010.
14. John Rushby. Formalism in safety cases. In Chris Dale and Tom Anderson, editors, *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, pages 3–17. Springer, February 2010.
15. Jacob T Schwarz. Programming with sets: an introduction to setl. Lavoisier, October 1986.
16. Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.
17. Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 349–361, New York, NY, USA, 2008. ACM.