Verified Functional Iterators Using the FoCaLiZe Environment

Catherine Dubois and Renaud Rioboo

CEDRIC-ENSIIE, 1 square de la résistance, 91025 Évry, France
{Catherine.Dubois,Renaud.Rioboo}@ensiie.fr

Abstract. Collections and iterators are widely used in the Object community since they are standards of the Java language. We present a certified functional implementation of collections and iterators addressing the Specification And Verification of Component Based Systems 2006 challenge. More precisely we describe a FoCaLiZe implementation providing these functionalities. Our approach uses inheritance and parameterization to describe functional iterators. Our code can be run in Ocaml and is certified using Coq. We provide general specifications for collections, iterators and removable iterators together with complete implementation for collections using lists as representation and iterators over those.

1 Introduction

Iterators on data structures like lists, sets, vectors, trees, etc. are available in many programming languages, usually as resources of their standard library. In functional languages, iterating facilities are mainly provided as higher order functions (e.g. `fold_left` or `fold_right` for iteration on lists in SML or Ocaml). In object oriented languages like Java, Eiffel or C#, they are provided as objects with methods allowing the enumeration of the data structure elements (e.g. `hasNext` and `next` in the Java `Iterable` interface). Usually the iterable data structure contains a method (e.g. `iterator` in Java collections), each invocation of which creates an iterator. Following the ITERATOR design pattern [7], iterators give a clean way for element-by-element access to a collection without exposing its underlying representation. Following this view, purely functional iterators can also be implemented, such as in [6] or in the Ocaml Reins Data Structure Library[1]. Thus we can consider the type of an iterator as an abstract data-type equipped with 3 functions `start`, `hasnext` and `step`: `start` is applied to a collection and computes an iterator; `hasnext` takes an iterator as an argument and returns a boolean indicating whether the enumeration is finished or not; and `step i`, when `i` is an iterator, returns an element not yet visited and the new iterator. Thus the underlying collection is provided as an argument to `start` and is not used anymore after that.

[1] See `http://ocaml-reins.sourceforge.net/api/Reins.Iterator.S.html` for the interface of the Iterator module.

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 317–331, 2014.

In this paper, we propose a verified implementation of such functional iterators. Here verified means that this implementation has been proved correct with respect to the specification. Specification, implementation and proof are done using the FoCaLiZe[2] environment (which is a successor of FoCaL) [5]. As far as we know, it is the first verified implementation of functional iterators in the flavor of those proposed e.g. by Filliâtre in [6]. In this study we are mainly looking for a way to specify the behavior of an iterator without exposing its internal representation or the representation of the collection it traverses and to evaluate how convenient it is for specifying and proving generic algorithms using such iterators.

This work can also be seen as a contribution to the 2006 SAVCBS (Specification And Verification of Component Based Systems) challenge asking for a specification of the Iterator interface as provided in Java or its equivalent in another language[3]. Different solutions [1] have been proposed, most of them focusing on the verification of non-interference between calls that directly modify the collection and interleaved uses of one or more iterators.

The FoCaLiZe language in which our development is done, is functional. However it borrows some features to the Object world, such as inheritance, redefinition that ease reuse of specifications, code and proofs. Furthermore parameterization facilitates the definition of generic iterators and derived functions.

The rest of this paper is structured as follows. Section 2 presents FoCaLiZe very quickly. Then we introduce in Section 3 the main ingredients to use iterators. Iterators allow the enumeration of values contained in another data structure, often collections. Thus we stick to this view and propose a FoCaLiZe specification of collections and an implementation for sequences as collections in Section 4. Then we present in Section 5 a FoCaLiZe implementation of iterators for collections which are sequences. Some existing approaches are presented and discussed in Section 6. Section 7 concludes and presents some future work.

2 A Quick Presentation of FoCaLiZe

The FoCaLiZe environment provides a set of tools to describe and implement functions and logical statements together with their proof. A FoCaLiZe source program is analyzed and translated into Ocaml sources for execution and Coq sources for certification. The FoCaLiZe language has an object oriented flavor allowing inheritance, late binding and redefinition.

FoCaLiZe concrete programming units are *collections* which contain entities in a model akin to classes and objects or types and values. In the following, to avoid confusion with collections as data containers, a FoCaLiZe collection is called an Fcollection. Fcollections have *methods* which can be called using the "`!`" notation as in Code 1. They are derived from *species* which describe and implement methods. In an Fcollection the concrete *representation* of entities is abstracted and a programmer refers to it using the keyword `Self` in FoCaLiZe sources.

[2] `http://focalize.inria.fr`

[3] `http://www.eecs.ucf.edu/~leavens/SAVCBS/2006/challenge.shtml`

Species may inherit from other species and may have parameters which may either be Fcollections or entities providing parametric polymorphism. As shown in Code 4 parameters are declared in sequence and may have dependencies: this excerpt describes a species parameterized by 2 Fcollections named resp. `Elt` and `L`. The first one is expected to derive from the `Setoid` species, it means that it provides at least all the methods appearing in the interface of `Setoid`, i.e the list of methods appearing in `Setoid` or inherited, with their type where the type of entities is made abstract. The second parameter `L` is expected to provide methods in the interface of `Utils(Elt)` where `Utils` is a parameterized species which is applied to the effective Fcollection `Elt`. We can notice the dependency between the first and the second argument.

A species defines a set of entities together with functions and properties applying to them. At the beginning of a development, the representation of these entities is usually abstract, it is precised later in the development. However the type of these entities is referred as `Self` in any species. Species may contain specifications, functions and proofs, all of theses being called *methods*. More precisely species may specify a method (`signature`, `property` keywords as in code 4) or implement it (`let`, `proof of`, `theorem` keywords as in code 1 or 2). A `let` defined function must match its signature and similarly a proof introduced by `proof of` should prove the statement given by the `property` keyword. Statements belong to first order typed logic.

Within FoCaLiZe, proofs are written using the FoCaLiZe proof language and are sent to the Zenon prover which produces Coq proofs. The FoCaLiZe proof language is a declarative language in which the programmer states a property and gives hints (`by`) to achieve its proof which is performed by Zenon. She typically introduces a context with variables (`assume`) and hypothesis (`hypothesis`) and then states a result (`prove`). Elements of the proof are then listed (`by`) and these can be either an hypothesis (`hypothesis`), an already proved statement (`step`), an existing statement (`property`) or a definition (`definition of`, or `type`). When a proof has steps it is ended by `conclude` or `qed by` clauses. Code 9 shows the skeleton of a FoCaLiZe proof tree. The automatic prover Zenon is a first order automatic theorem prover which supports algebraic data types and induction developed by D. Doligez (see for instance [2]).

For more details on FoCaLiZe please refer to the reference manual. More explanations about FoCaLiZe syntax will be given in next sections when necessary.

3 Using Iterators

In this section we present a sample use of our iterators implementation. Our demonstration package `IterTools` proposes a function `copy` (see Code 1) that copies elements from a collection `c` of type `Col`, using an iterator `it` of type `It`. We use a tail recursive function `copy_aux` that uses an iterator (`it`) and a collection (`a`). Since FoCaLiZe is a functional language the state change of an object is implemented using an extra result and the `next` operation returns a pair made of an iterator together with the visited element. The species presented here

is parameterized by four Fcollections `Elt`, `L`, `Col` and `It` specifying operations for elements of the collections, lists of such elements, collections of such elements and iterators on the previous collections. These Fcollections derive from species which will explained in the following sections. The overall hierarchy of species is shown in Figure 1 (without parameters for sake of clarity).

We can notice that the `copy` function provided by the species `IterTools` uses the same implementation for both the source and the target collections. It can be generalized to allow different implementations. In that case, two Fcollections `Col1` (implementation of the source collection) and `Col2` (implementation of the target collection), both having the interface `Collection(Elt, L)`, will be provided as parameters of the species. Furthermore the `It` parameter would have as interface `Iterator(Elt, L, Col1)`.

Code 1

```
species IterTools (Elt is Setoid,
                   L is Utils(Elt),
                   Col is Collection(Elt, L),
                   It is Iterator(Elt, L, Col)) =

 let rec copy_aux (it, a) =
 if It!has_next(it) then
     let res = It!next (it) in
     copy_aux (snd(res), Col!add(fst(res), a))
 else a;

 let copy (c) = copy_aux (It!start (c), Col!empty);
```

At this step, we have identified some necessary operations of collections and iterators. Collections have to provide 2 operations, `empty` and `add`, these are the methods `Col!empty` and `Col!add` provided by the Fcollection parameter `Col`. For iterators, we need 3 operations, `start`, `has_next` and `next` provided by the Fcollection parameter `It`.

Correctness of `copy` relies on the theorem `copy_spec` whose statement is given in Code 2, establishing that the original collection and its copy have the same elements. The statement uses the `contains` method provided by collections. Inside the proof, we use an invariant property, `copy_invariant` (also in Code 2) which refers to the abstract model of an iterator stated by the logical predicate `model`. For an iterator `i`, a collection `c` and a list of elements `l` the `model(i, c, l)` statement should describe logically the elements of `c` belonging to the list `l` which are not yet visited by the iterator `i`. This informal specification and further constraints on the `model` predicate will be formalized in Section 5. The invariant property `copy_invariant` relates two collections between recursive calls in the `copy_aux` function. An element `x` is either an element of the collection `c` not yet visited by an iterator or contained in an auxiliary collection `a`.

Code 2 *excerpt of species `IterTools`*

```
theorem copy_spec :
```

```
  all e: Elt, all c : Col,
    Col!contains (e, c) <-> Col!contains (e, copy(c))
proof = (* 50 lines *);

theorem copy_invariant:
  all it: It, all a: Col, all c: Col, all l: list(Elt),
    It!model(it, c, l) ->
    (all x: Elt, Col!contains(x, a) -> Col!contains(x, c)) ->
      (all x: Elt, Col!contains(x, copy_aux(it, a)) <->
          ((L!mem(l, x) || Col!contains(x, a))))
proof = (* 150 lines *);
```

The above theorems allow us to state that if the `copy_aux` function terminates it performs the right action. In order to show termination we must prove that the recursive call in `copy_aux` decreases for some well founded order `it_order` defined below (see Code 3).

Code 3 *excerpt of species* `IterTools`

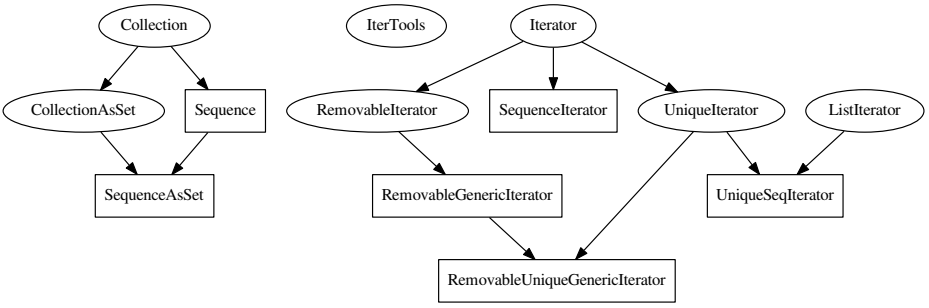
```
let it_order(it1, it2) =
  (0 <= It!measure_it(it2))  &&
  (It!measure_it(it1) < It!measure_it(it2));

theorem well_wrapper_it: well_wrapper(it_order)
proof =(* 30 lines of Coq *);

theorem rec_call_decreases: all it: It, all res: Elt * It,
It!has_next(it) -> (It!next(it) = res) -> it_order(snd(res), it)
  proof = (* 150 lines *);
```

The `well_wrapper` statement is part of FoCaLiZe standard library and states that the ordering it receives as argument is well founded. Since it is not a first order statement we cannot prove it using the Zenon prover and have to do the proof in Coq. Proofs are here omitted but performed by unfolding the different definitions and making use of the `(Zwf_well_founded 0)` Coq property that establishes that the usual order on positive integers is well founded. In order to prove that the recursive call decreases (theorem `rec_call_decreases`) we rely on a property stating that when iterating we decrease some measure of an iterator. We thus have identified two other operations `it_measure_it` and `mea_decreases` which we specify in Code 10.

We can see that though the 6 lines of effective code in Code 1 use a simple accumulator they must be completed by 10 lines of specification statements in Code 2 which are not obvious to guess. Furthermore their proofs are quite tedious and the overall species is globally 350 lines of mixed FoCaLiZe and Coq code.

4 Collections

Though the word collection is a keyword of the FoCaLiZe language we use it in the normal UML/Java sense and we present the functionalities we implemented.



Ellipses correspond to specification species. Rectangles correspond to implementation species (complete species)

Fig. 1. The Overall Hierarchy

4.1 Specification Hierarchy

Basic collections contain a finite number of values and we have methods to add and remove values in a collection, check if a value is in a collection and transform a collection into a list of values as in Code 4. We can also compute the size of a collection, check if a collection is empty etc.

Code 4

```
species Collection (Elt is Setoid, L is Utils(Elt))  =
  signature contains: Elt -> Self -> bool;

  signature add: Elt -> Self -> Self;
  property add_contains: all c: Self, all e x: Elt,
    contains(x, add(e, c)) <-> ( (x = e) || contains(x, c));

  signature remove: Elt -> Self -> Self;
  property remove_contains:
    all c: Self, all e : Elt, all x : Elt, not (x = e)) ->
       (contains (x, (remove (e, c))) <-> contains (x, c));

  signature tolist : Self -> list(Elt);
  property tolist_contains :
    all c: Self, all e : Elt, contains (e, c)
        <-> L!mem (tolist(c), e);
```

We then distinguish between collections which are sets. Thus we define, using inheritance, a new species (`CollectionAsSet`) describing these collections as sets. A new property, `unique_contains`, is added, it explains that such collections have no redundant element (Code 5)

Code 5

```
species CollectionAsSet (Elt is Setoid, L is Utils(Elt))  =
  inherit Collection(Elt, L) ;

  property unique_contains : all c : Self, all e : Elt,
    not(contains (e, remove (e, c))) ;
end;;
```

4.2 Implementations

In the previous subsection, a specification of basic collections is presented. We could go further into specifications by providing specifications for sequences, maps and then provide one or more implementations for each category. Here we simply provide an implementation of sequences using a simple implementation based on lists (seen as the elements of an inductive type providing 2 constructors: cons and empty). In FoCaLiZe a species is complete when all its signatures have an implementation and all its statements have received a proof. Thus the species given in Code 6 contains the definition of every function and the proof of every property specified in `Collection`. It also exports the functions `head` and `tail` with their specifications which have their obvious meaning.

In this case all functions have a simple termination proof (`structural` keyword as in Code 7) since their code performs simple pattern matching on lists and recursive calls on the tail of their initial parameter. All properties can be proved by induction over lists which is supported by the Zenon prover as shown in the proof of `unique_contains` (Code 9). Statements `<2>1` and `<2>1` are the base and inductive steps, statement `<2>3` is the property we prove inductively and the `<2>f` step enables to abstract the list representation.

Code 6

```
species Sequence (Elt is Setoid, L is Utils(Elt)) =
inherit Collection(Elt, L);

representation = list(Elt);

let contains (e: Elt, l: Self) = L!mem (l, e);
```

We also provide a `SequenceAsSet` complete species where we use the invariant that lists have no doubles as in Code 7.

Code 7

```
species SequenceAsSet(Elt is Setoid, L is Utils(Elt)) =
inherit Sequence(Elt, L), CollectionAsSet(Elt, L);

let torep (l : Self ) : list(Elt) = l;

let rec nodouble (l) =
      match l with
```

```
        | [] ->  true
        | h :: q ->  nodouble (q) && not(L!mem (q, h))
termination proof = structural l;
```

For correctness of the invariant we follow [17] and need to prove that all functions returning an element of `Self` preserve the invariant property as shown in Code 8. These proofs use induction on lists and the overall code is 170 lines of FoCaLiZe code.

Code 8 *excerpt of species* `SequenceAsSet`

```
theorem remove_preserves_inv : all l : Self, all e : Elt,
  nodouble (torep(l)) -> nodouble (torep(remove (e, l)))
proof =
<1>1 prove all l: list(Elt), all e: Elt,
     nodouble(l) -> nodouble(remove(e, l))
     (* 40 lines proof *)
<1>2 qed by step <1>1 definition of torep;
```

We can now prove uniqueness of an element in a sequence implemented as a list in Code 9 which statement is in Code 5

Code 9 *excerpt of species* `SequenceAsSet`

```
proof of unique_contains =
<1>1 assume c : Self,
     assume e : Elt,
     prove  not( L!mem (remove(e, c), e))
     <2>1 prove not(L!mem (remove(e, []), e)) (* 2lines *)
     <2>2 prove all l : list(Elt),  not(L!mem (remove(e, l), e))
          -> all x : Elt,  not(L!mem (remove(e, x::l), e))
          (* 30 lines *)
     <2>3 prove all l : list(Elt), not(L!mem (remove(e, l), e))
          by step <2>1, <2>2
     <2>f qed by step <2>3
<1>f qed by step <1>1
            property tolist_contains
            definition of tolist;
```

5 Iterators

Once collections have been specified, we can use them to specify iterators. We were inspired functional iterators of [6] for the interface and by JML specifications of [8] for the logical description.

5.1 Specification Hierarchy

In this paper we mainly describe finite linear iterations but many other may be considered. In order to provide a library which can easily be reused we heavily use inheritance and parameterization. We begin with basic iterator functionalities as in Code 10. An Fcollection implementing collections `Col` is a parameter of the specification species of iterators `Iterator`.

Code 10

```
species Iterator (Elt is Setoid,
                  L is Utils(Elt),
                  Col is Collection(Elt, L)) =

signature start : Col -> Self;

signature has_next : Self -> bool;

signature step_it : Self -> Elt * Self;
property step_it_empty :
  all c : Col, Col!is_empty (c) -> not(has_next (start (c)));
property step_it_nonempty :
  all c : Col, not(Col!is_empty (c)) -> has_next (start (c));

signature measure_it : Self -> int;
property mea_positive : all a : Self, 0 <=  measure_it (a) ;
property mea_decreases :
  all i1: Self, all res: Elt * Self, has_next(i1) ->
    step_it(i1) = res -> measure_it (snd(res)) < measure_it (i1);
```

The `mea_decreases` property expresses that when stepping an iterator we decrease a measure and thus enables us to prove the termination of the iteration.

As outlined in Section 3 and following [3] and [8], we rely on the user of our hierarchy for writing a `model` logical statement relating an iterator, a collection and a list of values. An implementation of the `model` signature should be a statement describing the list of elements of collection which have not been visited by the iterator.

Code 11 *excerpt of species* `Iterator`

```
signature model : Self -> Col -> list(Elt) ->  prop;

(** elements of l are in c *)
property model_includes: all it: Self, all c: Col, all l : list(Elt),
   model(it, c, l) -> all e: Elt, L!mem(l, e) -> Col!contains(e, c);

(** should start with full collection *)
property model_start : all c : Col, model (start (c), c, Col!tolist(c)) ;

(** when has_next is true l should not be empty *)
property model_has_next_true :
   all it : Self, all l : list(Elt), all c : Col,
   model (it, c, l) -> has_next(it) -> not(l = []);

(** when has_next is false there should remain no element to treat *)
property model_has_next_false :
   all it : Self, all l : list(Elt), all c : Col,
   model (it, c, l) -> not (has_next(it)) -> l = [];
```

```
[* we should return an element among those to be treated *)
property model_step :
   all it it2 : Self, all e : Elt, all l : list(Elt), all c : Col,
    model (it, c, l) -> has_next(it) -> step_it (it) = (e, it2)  ->
      L!mem (l, e);

property model_step_exists: all it it2: Self, all c: Col, all e: Elt,
   all l: list(Elt), has_next(it) ->
     model(it, c, l) -> step_it(it) = (e, it2) ->
       (* there exists a list which is a model  for it2 *)
       (ex l2: list(Elt), model(it2, c, l2));
```

The properties in Code 11 explicit the informal specifications given in section 3 of the `model` statement which is thus a key component of our implementation. The last statement expresses that when stepping an iterator we still have some model.

We now can specify iterators that visit only once an element of a set using inheritance as in Code 12.

Code 12

```
species UniqueIterator (Elt is Setoid,
                        L is Utils(Elt),
                        Col is CollectionAsSet(Elt, L)) =
inherit Iterator(Elt, L, Col) ;

property model_step_unique :
all it it2: Self, all e: Elt, all l l2: list(Elt), all c : Col,
  model (it, c, l) -> has_next(it) -> step_it (it) = (e, it2)  ->
    model(it2, c, l2) -> not(L!mem(l2, e));
end
```

The Java informal specifications introduce the notion of iterators from which the last iterated element can be removed from the collection it belongs to. This is achieved using the optional remove functionality, on the contrary we rely on FoCaLiZe inheritance to specify the remove functionality. In Code 13 we define a species of removable iterators that inherits from basic iterators and adds new specifications.

Code 13

```
species RemovableIterator(Elt is Setoid,
                          L is Utils(Elt),
                          Col is Collection(Elt, L)) =
  inherit Iterator(Elt, L, Col) ;

  signature remove: Self -> Self ;
  property remove_spec :
    all it it2: Self, all e: Elt, all l l2: list(Elt), all c : Col,
      model (it, c, l) -> has_next(it) ->
        step_it (it) = (e, it2)  -> model(it2, c, l2) ->
```

```
          model(remove(it2), Col!remove(e, c), l2);

  signature get_collection: Self -> Col ;
end
```

The statement `remove_spec` explains the behavior of the `remove` function, we also provide a `get_collection` operation to retrieve the new collection that results from the application of `remove` on an iterator.

5.2 Implementations

In this subsection, we describe our implementation of iterators which use sequences as representation and also a generic implementation of removable iterators.

First, in the species `SequenceIterator` (see Code 14), we represent an iterator by a sequence containing the elements left to be treated by the iterator. We use operations from `Sequence` (`head`, `tail` as in Section 4) to traverse the values.

Code 14

```
species SequenceIterator (Elt is Setoid, L is Utils(Elt),
                          LCol is Sequence(Elt, L)) =
inherit Iterator(Elt, L, LCol);

representation = LCol ;

let tolist (l : Self) = LCol!tolist (l);

logical let model (it: Self, c, l) =
  (all x : Elt, L!mem (l, x)
   <->
   L!mem (tolist(it), x));

let start (c : LCol) : Self = c;

let has_next (it : Self) = not(LCol!is_empty (it));

let step_it (it) =
  if has_next (it) then (LCol!head (it), LCol!tail (it))
  else focalize_error ("no more elements") ;

let measure_it (c) = LCol!size (c);
```

We have defined the `model` statement which expresses that the list of elements to visit should be the list view of the iterator. Here this view is the list view of the underlying sequence as defined by the `tolist` function.

The overall `SequenceIterator` species is 150 lines of FoCaLiZe code where proofs mostly involve unfolding definitions. For iterators that visit an element only once we implemented the `UniqueSeqIterator` species by simple inheritance and we proved the `model_step_unique` property in 40 lines of FoCaLiZe.

We use a species `RemovableGenericIterator` to implement the remove feature. It takes an iterator as argument as in Code 15. To implement removable iterators we encapsulate a general iterator with the necessary information to keep track of the last element returned and of the collection of remaining values after deletion. In Code 15 we use the `PFailed` value to reflect that no element can be removed from the collection. As in Java we enforce that only the last visited element can be removed.

Code 15

```
type partial('a) =
  | PFailed
  | PUnfailed('a);;

species RemovableGenericIterator(Elt is Setoid, L is Utils(Elt),
                                 Col is Collection(Elt, L),
                                 It is Iterator(Elt, L, Col)) =

inherit RemovableIterator(Elt, L, Col);

(* basic iterator, last element returned, current collection *)
representation = It * (partial(Elt) * Col) ;

logical let model(it, c, l) =
  It!model(fst(it), c, l);

let get_collection(it: Self) = snd (snd (it) );

let remove(it: Self): Self =
  let i = fst(it) and re = snd(it) in
  let e = fst(re) and col = snd(re) in
  match e with
    | PFailed -> it
    | PUnfailed(x) -> (i, (PFailed , Col!remove(x, col)));

let start(c) = (It!start(c), (PFailed, c));

let step_it(it) =
  if has_next(it)
  then
    let c = It!step_it(fst(it)) in
      (fst(c), (snd(c), (PUnfailed(fst(c)), snd(snd(it)))))
  else focalize_error("no more elements");
```

This species implements all of the basic iterators specifications (for instance `has_next` or `model_start` of `Iterator`) together with features of `remove`. Basic iterator methods are implemented by de-structuring the iterator's representation and using the corresponding methods of the embedded basic iterator.

We also provide a `RemovableUniqueGenericIterator` species which inherits removable iterators and proves that an element is only visited once. The overall code for the two species implementing remove facilities is 300 lines of FoCaLiZe code.

6 Related Work

Collections and iterators have been studied from a verification point of view by different researchers, mainly in the context of Java or C#. Existing approaches differ a lot and some of them are mentioned below.

Besides static verification methods, run-time verification has been used, it allows for example the verification of safe enumeration, by monitoring and predictive analysis such as in [16].

Formal specification and deductive methods and tools have tackle the problem of safe iterators or safe use of iterators. A very early specification of Alphard like iterators using traces has been given by Lamb in [12]. Iterators have also been studied in the context of the refinement calculus [10]. In this work, iterators are translated into catamorphisms.

Some approaches (see e.g. [11] and [15]) and use higher order separation logic to verify some properties on iterators. It is strongly linked to an imperative implementations with shared mutable heap structures and thus not in the scope of our approach.

Model oriented specification to describe how iterators behave is a largely adopted approach (see e.g. [18], [8], [4], [9], [14]). Contracts are associated to collections and iterators in the form of pre and post-conditions and invariants. In Java and JML context, this kind of specification may use model fields [3] such as in [8] where the abstract model of a collection is a bag. From specification and code, verification conditions are generated and then proved, automatically or not, by a theorem prover. Our approach is close to this model based style, however we don't fix a choice for the model. We only specified it as a logical statement and its definition is left to the implementor of the iterators. This avoids explicit bag abstraction and we believe, allows more flexibility to describe iterators.

Such uses of specification contracts, model fields in particular, usually allow modular verification. Our approach facilitates also this modular verification since we are able to verify functions dealing with iterators according to their interface and thus using only the specifications of iterators (`model_start`, `model_next` etc.).

The standard Coq library provides a modular specification for finite maps and sets very similar to those found in the Ocaml library implemented using lists and efficient trees. Iterators are not featured, however a `fold` function is proposed. Also Filliâtre (see [6]) generalizes a `fold` function into efficient persistent iterators for Ocaml.

The Isabelle Collections Framework (ICF) [13] provides a unified framework for using verified collection data structures in Isabelle/HOL formalizations. They

come with iterators which are implemented as generalized fold combinators. The ICF supports maps, sets and sequences together with generic algorithms using the Isabelle abstraction facilities. Iterators are created using a continuation function, a state transformer function and an initial state. Support for reasoning is achieved using an invariant and specifications are provided for maps, sets and sequences. Our implementation only provides some support for sequences but we have designed a general framework in which maps and sets can be implemented. The ICF heavily relies on higher order functions whereas FoCaLiZe emphasizes on first order statements which are often easier to understand by programmers. We thus remain in the same spirit than the Java Collection Framework but we use persistent data and allow certification.

7 Conclusion

In this paper we have presented a formal specification of collections and iterators together with a verified implementation of iterators for sequential lists, using the FoCaLiZe environment. The overall FoCaLiZe formal development with specifications, code and proofs, contains around 1600 lines[4]. We have used as much as possible FoCaLiZe inheritance and parameterization in order to get a flexible, adaptable and reusable formal development.

As perspectives we plan to specify and implement iterators for other kind of collections such as trees or maps. Then, in our development it is possible to state that when there is no more element to visit, every element in the collection has been visited. It would also be interesting to exploit the fact that a sequence is an ordered aggregate of elements and thus specify and define iterators that enumerate the elements of such a collection respecting their order.

Furthermore since we have defined iterators as a species they are normal FoCaLiZe values and we are able to manipulate them as in Code 15. This should allow the combination of different iterators and thus provide generic species implementing combinators of iterators.

References

1. Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems. ACM, New York (2006)
2. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
3. Breunesse, C.-B., Poll, E.: Verifying jml specifications with model fields. In: Formal Techniques for Java-like Programs. Proceedings of the ECOOP 2003 Workshop (2003)
4. Cok, D.R.: Specifying java iterators with jml and esc/java2. In: Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems, SAVCBS 2006, pp. 71–74. ACM (2006)

[4] Available at the unlisted url `http://www.ensiie.fr/~rioboo/iterators.fcl`

5. Dubois, C., Hardin, T., Vigui Donzeau-Gouge, V.: Building certified components within focal. In: Loidl, H.-W. (ed.) Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004, München, Germany. Trends in Functional Programming, vol. 5, pp. 33–48. Intellect (2006)
6. Filliâtre, J.-C.: Backtracking iterators. In: Proceedings of the ACM Workshop on ML 2006, Portland, Oregon, USA, pp. 55–62. ACM (2006)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading (1994)
8. Huisman, M.: Verification of java's abstractcollection class: A case study. In: Boiten, E.A., Möller, B. (eds.) MPC 2002. LNCS, vol. 2386, pp. 175–194. Springer, Heidelberg (2002)
9. Jacobs, B., Meijer, E., Piessens, F., Schulte, W.: Iterators revisited: Proof rules and implementation. In: Workshop on Formal Techniques For Java-like Programs, FTFJP (2005)
10. King, S., Morgan, C.: An iterator construct for the refinement calculus. In: Fourth Irish Workshop on Formal Methods (2000)
11. Krishnaswami, N.R.: Reasoning about iterators with separation logic. In: Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems, SAVCBS 2006, pp. 83–86. ACM (2006)
12. Lamb, D.A.: Specification of iterators. IEEE Trans. Software Eng. 16(12), 1352–1360 (1990)
13. Lammich, P., Lochbihler, A.: The isabelle collections framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
14. Leino, K.R.M., Monahan, R.: Dafny meets the verification benchmarks challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010)
15. Malecha, G., Morrisett, G.: Mechanized verification with sharing. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 245–259. Springer, Heidelberg (2010)
16. Meredith, P., Roşu, G.: Runtime verification with the RV system. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 136–152. Springer, Heidelberg (2010)
17. Rioboo, R.: Invariants for the FoCaL language. Annals of Mathematics and Artificial Intelligence 56(3-4), 273–296 (2009)
18. Weide, B.W.: Savcbs 2006 challenge: Specification of iterators. In: Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems, SAVCBS 2006, pp. 75–77. ACM (2006)