Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic
François Pottier

To cite this version:
François Pottier. Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic. Certified Programs and Proofs, Jan 2017, Paris, France. hal-01417102

HAL Id: hal-01417102
https://inria.hal.science/hal-01417102
Submitted on 15 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic

François Pottier
Inria Paris, France *
Francois.Pottier@inria.fr

Abstract
We describe the specification and proof of an (imperative, sequential) hash table implementation. The usual dictionary operations (insertion, lookup, and so on) are supported, as well as iteration via folds and iterators. The code is written in OCaml and verified using higher-order separation logic, embedded in Coq, via the CFML tool and library. This case study is part of a larger project that aims to build a verified OCaml library of basic data structures.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords Verification, abstraction, iteration, modularity

1. Introduction
Since the days of Floyd and Hoare [11, 13], tremendous progress has been made in the area of program verification. We have witnessed advances in program logics, verification condition generators, SMT solvers, and interactive proof assistants. A number of landmark examples of verified software have appeared, including compilers [24, 34], static analyzers [16], model checkers [7], operating system kernels [17], cryptographic protocol implementations [1], and so on.

These impressive achievements offer a glimpse of a bright future where all software can in principle be verified. Yet, at present, verifying a nontrivial application or library still requires many man-years of effort. Therefore, we believe that it is time to begin building libraries of verified general-purpose utility components, with the double aim of speeding up the development of verified applications and offering dependable components to authors of unverified software.

The work presented in this paper is part of the Vocal project, which aims at developing such a verified library of general-purpose data structures and algorithms. Vocal uses OCaml as its implementation language, because it is safe, concise, modular, and efficient.

We describe one case study, namely the specification and proof of a mutable hash table implementation. This task is carried out using the CFML tool and library [5], which offer a higher-order total-correctness separation logic, embedded in Coq, for a subset of OCaml. Admittedly, expressing the specification of a sequential data structure, and proving that its implementation meets this specification, are not viewed today as challenging tasks. Nevertheless, we believe that this case study is worth documenting, as it involves a number of nontrivial aspects, namely abstraction, parameterization, and (perhaps most importantly) a generic treatment of iteration.

Abstraction The data structure is abstract: the client is aware that it represents a dictionary, but cannot know how it is laid out in the heap. Also, the data structure is mutable: one must reason about ownership of heap fragments. We use an abstract separation logic predicate [29, 27, 28] to simultaneously delimit a uniquely-owned heap fragment, impose an invariant upon its content, and relate its content with the abstraction that it is intended to represent.

Parameterization The data structure is parametric in the type of keys, which must be equipped with equality and hash functions, and in the type of values, which is unconstrained. This is expressed in OCaml and in Coq via a functor (for keys) and via polymorphism (for values).

Iteration Iteration is enabled by two distinct mechanisms, namely "folds" (higher-order functions) and "cascades" (which can be described both as delayed lists and as a form of iterators). Although folds are the predominant means of iteration in the OCaml world, we promote cascades, as they are more versatile than folds and easy to implement. Both are independent of hash tables: their types and specifications should be fixed, once and for all, outside of the hash table module. For a number of reasons, these specifications are necessarily somewhat complex:

* This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.
They involve first-class functions: the consumer, in the case of folds; the producer, in the case of cascades.

They involve read access to a mutable data structure, which creates a well-known “concurrent modification” problem [32]. In order to preserve the producer’s integrity, they must forbid mutation while iteration is in progress.\footnote{One could also allow mutation, as long as it preserves the producer’s invariant. We leave that to future work.} This requires declaring existing cascades invalid when mutation occurs (§4.3, §4.8).

They must support nondeterminism, that is, allow the order in which elements are produced to remain partly or entirely unspecified.

The specification of cascades must also support producing infinite sequences of elements.

To tame this complexity, we propose generic, reusable specifications of folds and cascades, and show that they are easily instantiated for hash tables.

Our hash table implementation is under 150 (nonblank, noncomment) lines of OCaml code. It offers the same set of operations as \texttt{Hashtbl\_Make} in OCaml’s standard library. Our Coq code includes roughly 300 lines of statements and 300 lines of proofs at the abstract level of dictionaries and roughly 700 lines of statements and 700 lines of proofs at the concrete level of hash tables. Our code, specifications, and proofs are available online \cite{code}.

In the following, we first present and justify our definition of cascades, which is independent of hash tables (§2). We then present the general structure as well as some excerpts of our OCaml code for hash tables (§3). Then, we present our Coq proof (§4), putting emphasis on its general structure, on the hash table invariant, and on the specifications of a few key operations.

2. Folds, Iterators, and Cascades

A “fold” \cite{fold} is a producer of a sequence of elements which has the ability of submitting elements to the consumer. It expects to receive (as an argument) a function that represents the processing by the consumer of one element, and invokes this function whenever it wishes to produce an element.

An iterator \cite{iterator, Chapter 6}, on the other hand, is an on-demand producer of a sequence of elements. That is, an iterator is an object that can be queried by a consumer, when desired, for an element of the sequence. In response to such a query, an iterator returns an element if one is available (that is, if the sequence is nonempty) and nothing otherwise (that is, if the sequence is empty).

Because they leave control to the consumer, iterators are more versatile than folds. They allow abandoning an iteration before it is finished. (For this reason, they can produce conceptually infinite sequences, whereas folds cannot.) They allow simultaneously iterating over several sequences. In fact, an iterator can be easily wrapped as a fold, whereas the converse adaptation requires control operators. This remark may lead one to believe that iterators must be more difficult to implement than folds. This need not be the case: as shown in the present paper (§3), producing a cascade of elements is just as easy as producing a list of elements.

For maximum interoperability, one should fix, once and for all, the types of folds and iterators. These types serve as universal interfaces, which all producers implement, and all consumers rely upon. For folds, this is easy enough. (See, for instance, the type of \texttt{fold} in Figure 3.) For iterators, however, the question admits several reasonable, yet rather different, answers. In particular, when queried, what should an iterator return? Should it return just an element, or a pair of an element and another iterator? An iterator of the first type, or “implicit” iterator, must have mutable internal state. The element that it returns is implicitly consumed: a subsequent query to this iterator produces the following element of the sequence. An iterator of the second type, or “explicit” iterator, does not necessarily have mutable internal state. In addition to an element, it returns another explicit iterator, which gives access to the remainder of the sequence. A subsequent query to the original iterator, if permitted, produces the same element again.\footnote{If an explicit iterator has neither mutable internal state nor any effect on the outside world, then it is “persistent”: it can be queried several times, producing the same result every time. Otherwise, it might be “ephemeral”, which means that it may be queried at most once. For greatest generality, we allow both possibilities. It is easy to wrap an implicit iterator as an ephemeral explicit iterator, and an explicit iterator as an implicit one. Therefore, there is no loss of generality in adopting “explicit iterators”, as opposed to “implicit iterators”, as our universal type of iterators. There is in fact a net gain, since an explicit iterator can be persistent, whereas an implicit iterator cannot.}

These two types of iterators, although closely related, lead in practice to very different coding styles. Whereas implicit iterators must have mutable state, explicit iterators can often be implemented without side effects, which makes them easier to build and to reason about. For this reason, we favor explicit iterators. This is in contrast with Java, whose \texttt{Iterator} interface describes implicit iterators: the method \texttt{next} consumes and returns the next element, if there is one.

Java iterators must also offer a method \texttt{hasNext}, which tells whether one more element is available. We drop this requirement. With implicit iterators, \texttt{hasNext} can be quite useful, as it allows testing whether an element is available without consuming it. With (persistent) explicit iterators, one can simply use \texttt{next} for this purpose.
An iterator that offers just one method (namely, \texttt{next}) is a function. It takes an argument of type \texttt{unit} and returns either nothing or a pair of an element and another iterator. We refer to this particular type of iterators as “cascades”. Their definition appears in Figure 1. The auxiliary type \(\texttt{a head}\) represents a response, that is, either nothing (\texttt{Nil}) or a pair of an element and a cascade (\texttt{Cons}). A cascade is a function which, when queried, returns a response.

If one replaced \texttt{unit -> \texttt{a head}} on line 3 of Figure 1, then \(\texttt{a head}\) would be isomorphic to \(\texttt{a list}\). A cascade is a “delayed list”, that is, a list that does not necessarily exist in memory but is produced, element by element, on demand. A cascade is usually built or used in the same manner one would build or use a list, except “delays” and “forces” are thrown in where necessary.

In summary, we propose delayed lists, under the name of “cascades”, as a universal type of sequences. To a reader with a functional programming background, this should come as no surprise: almost thirty years ago, the designers of Haskell [14] put forward lazy lists, sometimes also known as “streams”, as a universal type of sequences. Cascades and streams are equally expressive: one can easily convert one to the other. We favor cascades over streams because we think that memoization should not be the default behavior: instead, it should be explicitly requested when desired.

We follow Filliâtre [8] in advocating explicit iterators. Although he briefly considers adopting delayed lists as the universal type of sequences [8, §4], motivated by efficiency considerations, he settles for distinct, ad hoc abstract types of tree iterators [8, §2], hash table iterators, and so on.

Jane Street’s library \texttt{Core.Sequence} also defines distinct types of iterators for distinct data structures. Yet, thanks to an existential quantification, it is able to give a single abstract type to all of them. This type is interconvertible with \(\texttt{a cascade}\). We use existential quantification in the same way, at the specification level, to give a nonrecursive definition of the abstract predicate \texttt{Cascade} (§4.7).

3. Hash Tables

The signature and implementation of our \texttt{HashTable} module are shown in Figures 2 and 3. They are modeled after the \texttt{Hashtbl} module found in OCaml’s standard library, with a few minor differences.\footnote{At version 4.03, OCaml’s \texttt{Hashtbl} module was modified to use mutable lists. This exploits a feature of OCaml 4.03 which CFML does not yet support, namely “mutable inline records”. We stick with immutable lists.}

A hash table represents a dictionary, that is, a mapping of keys to values. The type of keys must be equipped with an equality test and a hash function. For this reason, everything is wrapped in a functor, \texttt{Make}, which takes a module \(K\) of signature \texttt{HashedType} as an argument (Figure 2). The type of keys, \texttt{key}, is a synonym for \(K.t\). There are no requirements on the type of values. For this reason, the type of hash tables, \(\texttt{a t}\), is parameterized with \(\texttt{a}\), the type of values.

\begin{verbatim}
module type HashedType = sig
  type t
  val equal: t -> t -> bool
  val hash: t -> int
end

module Make (K : HashedType) : sig
  type 'a t = 'a t
  (* Creation. *)
  val create: int -> 'a t
  val copy: 'a t -> 'a t
  (* Insertion and removal. *)
  val add: 'a t -> key -> 'a -> unit
  val remove: 'a t -> key -> unit
  (* Lookup. *)
  val find: 'a t -> key -> 'a option
  val population: 'a t -> int
  (* Iteration. *)
  val fold: (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  val cascade: 'a t -> (key * 'a) cascade
  (* ... more operations, not shown. *)
end

Figure 2. The interface file HashTable.mli

module Make (K : HashedType) = struct
  (* Type definitions. *)
  type key = K.t
  type 'a bucket =
    Void
  | More of key * 'a * 'a bucket
  type 'a table = {
    mutable data: 'a bucket array;
    mutable popu: int;
    init: int;
  }
  type 'a t = 'a table
  (* Operations. *)
  (* add: see Figure 4. *)
  (* fold: see Figure 5. *)
  (* cascade: see Figure 6. *)
  (* other operations: not shown. *)
end

Figure 3. The implementation file HashTable.ml
\end{verbatim}

The type of hash tables, \(\texttt{a t}\), is defined internally as a record of three fields (Figure 3), namely: a data array, \texttt{data}; an integer population count, \texttt{popu}; and an integer initial capacity, \texttt{init}. Each entry in the data array holds an immutable list of key-value pairs, or “bucket”.

We follow OCaml’s standard library and allow a bucket to contain several entries for the same key, with the convention
let index h k = (K.hash k) land (Array.length h.data - 1)

let rec resize_aux h = function
  | Void -> ()
  | More (k, x, b) ->
    resize_aux h b;
    let i = index h k in
    h.data.(i) <- More (k, x, h.data.(i))

let resize h = let old = h.data in
  let nsize = Array.length old * 2 in
  if nsize < Sys.max_array_length
  then begin
    h.data <- Array.make nsize Void;
    for i = 0 to Array.length old - 1 do
      resize_aux h old.(i)
    done
  end

let add h k x = let i = index h k in
  h.data.(i) <- More (k, x, h.data.(i));
h.popu <- h.popu + 1;
if h.popu > 2 * Array.length h.data
  then resize h

Figure 4. Implementation of insertion

let rec fold_aux f b accu = match b with
  | Void -> accu
  | More (k, x, b) ->
    let accu = f k x accu in
    fold_aux f b accu

let fold f h accu = let data = h.data in
  let state = ref accu in
  for i = 0 to Array.length data - 1 do
    state := fold_aux f data.(i) !state
  done;
  !state

Figure 5. Implementation of iteration via fold

that the entry that was most recently added appears earliest in the list and is the one returned by find.4

Figure 4 shows the code for insertion. The auxiliary function index computes the index in the data array where the key k should be stored. It assumes that the length of the array is a power of two and uses the “logical and” operator as an efficient way of computing a remainder. The auxiliary functions resize_aux and resize are in charge of resizing the hash table by transferring the data to a new array whose size is twice that of the previous array. resize_aux is written in such a way as to preserve the ordering of entries in the case where there are multiple entries for a single key.

Figure 5 shows the code for iteration where the producer has control, that is, fold. The code involves two nested loops: an outer loop over the data array and an inner loop (implemented as a tail-recursive function, fold_aux) over each bucket. Each key-value pair (k, x) is presented to the client via a call to the user-supplied function f.

Figure 6 shows the code for iteration where the consumer has control, that is, cascade. All of the cascades constructed here (that is, the main cascade and its suffixes) take the form fun () -> cascade_aux data i b, where data is the data array, i is the index of the most recently fetched bucket, and b is the suffix of that bucket that remains to be traversed. They are immutable, therefore persistent. We again emphasize the similarity between cascades and lists: if one removed the delays “fun () ->” and replaced the cascade constructors Nil and Cons with the list constructors [] and ::, then this code would produce a list of all key-value pairs in the table.

4. Specification

4.1 CFML in a Nutshell

The CFML package [5] consists of three main components, namely: a library of Coq definitions and lemmas; a characteristic formula generator; and a suite of Coq tactics.

The Coq library introduces the concepts of separation logic. A (heterogeneous) heap is a finite map of memory

let rec cascade_aux data i b = match b with
  | More (k, x, b) ->
    Cons (k, x),
    fun () -> cascade_aux data i b
  | Void ->
    let i = i + 1 in
    if i < Array.length data then
      cascade_aux data i data.(i)
    else
      Nil

let cascade h =
  let data = h.data in
  let b = data.(0) in
  fun () ->
    cascade_aux data 0 b

Figure 6. Implementation of iteration via cascade

4 Upon reflection, this feature seems of dubious interest. If heavily used, it could degrade the performance of resize and find. We retain it, but encourage the use of “lean” tables, which have at most one entry per key (§4.4). Our specifications make it easy for the user to prove that her tables remain lean.
We write $x \sim > P$ for the empty heap assertion, $\emptyset F$ (where $F$ is a proposition) for a pure assertion, $P \& Q$ for the separating conjunction of the assertions $P$ and $Q$, and $\exists x, P$ for an existentially quantified assertion. If $P$ is a separation logic predicate, we write $x \rightarrow P$ (“$x$ points to $P$”) for the assertion $P$.

Assuming that a big-step operational semantics of the programming language of interest (in our case, a subset of OCaml) is given, the library proceeds to define a Hoare logic. By definition, the Hoare triple $\{P\} f x \{Q\}$, which we write $\text{app} \ f \ [x] \ \text{PRE} \ P \ \text{POST} \ Q$, means that, if run in a heap that satisfies the assertion $P \& F$, the application of the function $f$ to the value $x$ is safe and terminates, producing a value $y$ and a heap that satisfies the assertion $Q \ y \ & true \ & F$.

We write $\text{app} \ f \ [x] \ \text{INV} \ P \ \text{POST} \ Q$ for the Hoare triple $\{P\} f x \{P \& Q\}$, where the precondition $P$ is preserved.

The generator (which is implemented in OCaml, re-using part of the OCaml front-end) transforms a well-typed OCaml term $t$ into a Coq term $\tau t$, known as the “characteristic formula” for $t$. By construction, this term represents the set of all pairs $(P, Q)$ such that the Hoare triple $\{P\} t \{Q\}$ is valid. Thus, assuming that the generator is correct, in order to establish that $\{P\} t \{Q\}$ is a valid specification for $t$, it suffices to prove in Coq that the pair $(P, Q)$ is a member of the set $\tau t$.

The Coq tactics provided by CFML help the user carry out such a proof. They are intended to give the user the illusion that she is applying the reasoning rules of separation logic directly to the OCaml code.

So far, we have mentioned “functions” and “values”, but have said nothing about their types. Let us clarify. CFML does not use a universal Coq type of OCaml values. Instead, the construction of characteristic formulae is type-directed. An OCaml value of type $\tau$ is reflected as a Coq value whose type $\tau$, regardless of its argument and result types. An OCaml reference (or mutable record) is viewed as a value of abstract type $\text{func}$, regardless of the type of its content.

Thus, the OCaml module type $\text{HashedType}$ is translated to a Coq module type named $\text{HashedType}\_ml$.

For more details about CFML, the reader is referred to Charguéraud’s paper [4].

### 4.2 Setup

The OCaml code in the file $\text{HashTable}\_ml$ is transformed by the CFML generator into characteristic formulae, stored in the Coq file $\text{HashTable}\_ml.v$. As a user of CFML, we need not inspect the content of this file; we just load it.

We place our specifications and proofs in the hand-written file $\text{HashTable}\_proof.v$, whose architecture is shown in Figure 7. This file is modeled after the OCaml source file: whereas the OCaml code is wrapped in a functor, Make, whose parameter $K$ has signature $\text{HashedType}$, this file defines a functor, $\text{MakeSpec}$, whose parameter $K$ has signature $\text{HashedType}\_spec$. This signature, which we define, extends $\text{HashedType}\_ml$ and expresses our requirements about the OCaml functions $\text{equal}$ and $\text{hash}$: there must exist an equivalence relation $E$ on keys and a hash function $H$ on keys such that (1) $H$ is compatible with $E$ (that is, equivalent keys have the same hash), and of two functions $\text{equal}$ and $\text{hash}$ of type $\text{func}$.

This is a standard notion of total correctness. The fact that an arbitrary frame $F$ is preserved means that the code cannot part of the heap for which it has no access rights. The use of the conjunct $true$ means that the postcondition $Q y$ describes not necessarily the entire final heap, but possibly only a fragment of it.

It may seem strange that type information is lost. In fact, assertions such as “this function maps integers to integers” or “this reference contains an integer value” can be expressed and proved using the program logic.
equal hashes); (2) equal decides key equivalence; and (3) hash computes a key’s hash.\(^3\)

Inside the body of the functor MakeSpec, we apply the functor Make_ml (the auto-generated Coq counterpart of the OCaml functor Make) to K, and refer to the result as MK, so, for instance, MK.add refers to the OCaml hash table insertion function.

Finally, we open a Coq section and introduce a type variable A, which we use as the Coq counterpart of the OCaml type variable ‘a. The specifications and proofs in this section become polymorphic in A when the section ends.

We can now define the separation logic predicates Table and TableInState, which describe how a well-formed hash table is laid out in memory and what information it represents. This is done in the next section (§4.3). Then, we describe the specifications of the OCaml functions add (§4.4), fold (§4.5, §4.6), and cascade (§4.7, §4.8), which rely on these predicates. In the interest of space, we omit the specifications of all other functions, and omit all proofs.

4.3 Model and Invariant

What abstraction does a hash table represent? An obvious answer is: a dictionary, that is, roughly speaking, a mapping of keys to values. More specifically, because we allow a bucket to contain several entries for a single key (§3), a “dictionary” for our purposes can be defined as a function of keys to lists of values, that is, a Coq object of type key -> list A. We let M (for “model”, or “map”) range over such dictionaries (Figure 8, line 1).

For a function M to be representable by a hash table, M must satisfy certain properties. First, M must not distinguish two equivalent keys; that is, Proper (E ==> eq) M must hold. Second, M must have finite domain. That is, there must exist a set of keys D such that (1) D is finite; (2) D is irredundant, that is, two equivalent keys in D are equal; and (3) M maps to nil every key that is not equivalent to some key in D. We write is_domain D M for the conjunction of these conditions (Figure 8).

What is a hash table, and how is it laid out in memory? In OCaml, a value h of type 'a table is the address of a mutable record. In Coq, it is reflected as a value of type MK.table_ A\(^9\) (Figure 8, line 2). The content of such a record is described by a points-to assertion, an example of which appears in Figure 8, lines 30–34. Such an assertion claims the unique ownership of the record at address h and at the same time states that its three fields contain the values d, pop, and init, respectively.

In OCaml, the field access expression h.data, which has type 'a bucket array, evaluates to the address of a mutable array of buckets. In Coq, we usually write d\(^8\) for the address of this array: d has type loc. We usually write data for the content of this array: data has type list (MK.bucket_ A) (Figure 8, lines 3–4).

We now define several predicates which describe how a hash table is laid out in memory and how this concrete representation is related with the abstract model of the hash table, namely, a dictionary M.

The proposition content M data (lines 6–8) indicates that the array data contains all of the key-value pairs required by M, stored at appropriate offsets.\(^10\) This implies

\(^3\)The predicate Proper is part of Coq’s standard library. The predicates computes and decides, defined by CFML, are abbreviations for Hoare triples.

\(^4\)This is the auto-generated counterpart of the OCaml type 'a table. Because this is a mutable record type, MK.table_ A is defined by CFML as a synonym for loc.

\(^5\)We write k // data for (Z.land (H k) (length data - 1)), that is, the remainder of the hash of the key k by the length of the array data. The function bfilter k, whose definition is omitted, filters a bucket, producing a list of the key-value pairs whose key is equivalent to k.

---

**Figure 8.** The hash table invariant
The specification of insertion (whose code was shown in Figure 4) appears in Figure 9. It takes the form of a theorem, add_spec, whose statement is a Hoare triple about the OCaml function add, which in Coq is known as MK.add.

We expect this triple to express the informal idea that “if h is a hash table, then the function call add h k x affects this table in such a way that the key-value pair \((k, x)\) is added to the dictionary that this table represents”.

Formally, the precondition \(h \rightarrow Table M\) expresses an assumption that the table initially represents a dictionary \(M\). The conjunction \(h \rightarrow Table M'\) in the postcondition, where \(M'\) is existentially quantified, means that, after the call, the table represents a dictionary \(M'\). These dictionaries are related by the equation \(M' = add M k x\). This equation refers to an add operation that we define in Coq at the level of dictionaries. Its two-line definition (not shown; see HashTable_model.v in the online archive [31]) says that \(M' k'\) is \(x :: M k\) if the keys \(k\) and \(k'\) are equivalent, and is \(M k\) otherwise.

The last conjunct in the postcondition is intended to facilitate the use of “lean” hash tables, which have at most one entry per key. By definition, the proposition lean \(M\) means \(forall k, length (M k) <= 1\). The implication lean \(M -> M k = nil -> lean M'\) states that if the table is initially lean and if there is no entry in it for the key \(k\), then, after insertion, the table remains lean. This is a lemma about the dictionary-level function add. Building it into the postcondition of the OCaml function add is redundant, but saves the user the trouble of manually applying this lemma.

4.5 Iteration via Fold, in General

The function fold (Figure 5), which allows iterating over all key-value pairs in a hash table, is one specific instance of the general concept of a “fold”. It is worth defining this concept, once and for all, so as to avoid repeating this slightly verbose and complicated definition every time we come across an instance of it.

We adopt the convention that a fold is a function of three arguments \(f, c,\) and \(accu\), where:

- \(c\) is a “collection” of some sort, out of which a sequence of elements can be drawn or computed;
- \(accu\) is the initial value of the “accumulator”, a state which the consumer is allowed to explicitly maintain throughout the iteration;

The specification of insertion (whose code was shown in Figure 4) appears in Figure 9. It takes the form of a theorem,
Variable fold : func.

Variables A B C : Type.

Variable call : func -> A -> B -> --B.

Variable permitted : list A -> Prop.

Variable complete : list A -> Prop.

Variables I : list A -> B -> hprop.

Variables S : C -> hprop.

Variable S' : C -> hprop.

Definition Fold :=
  forall f c,
      (forall x xs accu,
       permitted (xs & x) ->
       call f x accu
       PRE (S' c \* I xs accu)
       POST (fun accu =>
         S c \* I (xs & x) accu)
       ) ->
       forall accu,
       app fold [f c accu]
       PRE (S c \* I nil accu)
       POST (fun accu => Hexists xs,
         S c \* I xs accu \*
         \[ complete xs ]).

Figure 10. A generic specification of fold functions

- f is a function, which represents the consumer; when applied to an element and to an accumulator, it must return an updated accumulator.

This informal description is translated into a formal specification, and made more precise, in Figure 10. There, Fold is defined as an abbreviation for the specification of a “fold” function, fold. It states that, provided the user-supplied function f behaves in a certain manner (that is, satisfies a certain Hoare triple), fold itself behaves as desired (that is, satisfies another Hoare triple).

Since a call to fold encapsulates an iteration, it should be no surprise that the specification is parameterized with a loop invariant I. This invariant is itself parameterized over the sequence xs of elements that have been seen so far and over the current accumulator accu. The precondition of fold contains I nil accu, which means that the user must establish the invariant (of the empty list, and of the initial accumulator). Its postcondition contains I xs accu, which means that, at the end, the invariant still holds (of the list of elements that have been enumerated, and of the final accumulator). Naturally, this requires that f preserve the invariant. Our assumption about f states that, if (before a call to f) the invariant holds (of the elements xs seen so far and of the accumulator accu that is passed to f), then after this call the invariant should still hold (of the updated accumulator accu that is returned by f).

The producer may need some sort of permission to access the collection c: this is represented by the assertion S c in the pre- and postcondition of fold. The consumer may or may not be given a permission to access the collection: this is represented by the assertion S’ c in the pre- and postcondition of f.14

The parameter call encodes the calling convention of the function f. The notation --B in the type of call is short for hprop -> (B -> hprop) -> Prop: this means that call f x accu should be applied to a precondition and postcondition. In the simplest scenario, call is instantiated in such a way that call f x accu expands to app f [x accu]: this indicates that f is applied to an element and an accumulator. However, there exist other calling conventions: for instance, when we iterate over a hash table, an “element” is in fact a key-value pair, and we follow the convention that f is applied to three arguments: key, value, and accumulator. This is expressed by instantiating call so that call f (k, x) accu is app f [k x accu] (Figure 11, lines 10–11).

Finally, the parameters permitted and complete tell which sequences of elements the producer is allowed to emit (or, dually, which sequences of elements the consumer may observe). In short,

- permitted xs means that the “incomplete” sequence xs can be observed by the consumer. That is, this sequence of elements, possibly followed by more elements, can be observed.
- complete xs means that the “complete” sequence xs can be observed by the consumer. That is, this sequence of elements, followed by the termination of fold, can be observed.

In the simplest scenario, where the producer is finite and deterministic, the sequence ys that will be enumerated is known ahead of time. In that case, permitted xs should be prefix xs ys and complete xs should be xs = ys. The specification also allows for scenarios where the sequence of elements is infinite and/or not known ahead of time. When iterating over a set s, for instance, the order in which the elements of s are presented to the consumer is usually unspecified [33, 22]. As observed by Filliâtre and Pereira [9], this is described by defining permitted xs to mean “the elements of xs are pairwise distinct and form a subset of s” and complete xs to mean “the elements of xs are pairwise distinct and form the set s”.

The assumption permitted (xs & x) in the spec of f (Figure 10, line 14) means that, every time an element x is produced, the consumer may assume that the sequence of elements xs & x13 and of the updated accumulator accu is the sum of the elements xs & x13 and of the updated accumulator accu that is returned by f).

12 A typical invariant might be: “accu is the sum of the elements xs that have been processed so far”.

13 xs & x is sugar for xs ++ x :: nil.

14 If S’ c is S c, then the consumer has full access to the collection. If S’ c is the empty heap assertion [], then the consumer has no access to it.
1  Definition permitted kxs :=
2     exists M', removal M kxs M'.
3  Definition complete kxs :=
4     removal M kxs empty.
5
6  Theorem fold_spec_ro:
7     forall M s B I,
8     Fold MK.fold
9       (* Calling convention: *)
10      (fun f kx (accu : B) =>
11       -- app f [(fst kx) (snd kx) accu])
12      (* Permitted/complete sequences: *)
13      (permitted M) (complete M) I
14      (* fold requires and preserves this,
15       so does not modify the table: *)
16      (fun h => h ˜> TableInState M s)
17      (* f receives this and must preserve
18       it, hence can read the table: *)
19      (fun h => h ˜> TableInState M s).
20      (fun h => \[
21  Theorem fold_spec:
22     forall M s B I,
23     Fold MK.fold
24       (fun f kx (accu : B) =>
25        -- app f [(fst kx) (snd kx) accu])
26       (permitted M) (complete M) I
27       (* fold requires & preserves this: *)
28       (fun h => h ˜> Table M)
29       (* f cannot access the table: *)
30       (fun h => \[]).

Figure 11. Two specifications of fold

in a most-recent-first fashion.\(^{15}\) So, we choose to specify that
"the order in which the key-value pairs are produced corre-
sponds to a possible sequence of removals". We define
the predicate removal M kxs M' to mean that, starting from
the dictionary M, it is possible to remove the key-value pairs
in the sequence kxs, one after the other, and that this process
yields the dictionary M'. This definition is based on a function
remove M k, which is defined at the level of dictionaries (see
HashTable_model.v in the online archive [31]) and whose
effect is to remove the front element of the list M k. Based on
removal, the definitions of permitted and complete are
straightforward (Figure 11, lines 1–4).

We are now ready to state the specification of the fold
function on hash tables, as an instance of Fold, which was
declared in Figure 10. In fact, we make two such statements,
both of which are instances of Fold. The first statement,
fold_spec_ro, gives the consumer read-only access to the
hash table, and guarantees that fold itself does not modify
the table. The second statement, fold_spec, is slightly
simpler and easier to use, but gives the consumer no access
to the table. It is a corollary of the previous statement.

In fold_spec_ro, the parameters S and S' of Figure 10
are instantiated with fun h => h -> TableInState M s
where s is fixed throughout. Thus, producer and consumer
both have access to the table, but cannot alter its concrete
representation: the table must remain in state s. In other
words, they both have read-only access to the table.

In fold_spec, this time, the parameter S of Figure 10
is instantiated with fun h => h -> Table M, whereas S'
is instantiated with fun h => \[]. That is, the producer
needs full access to the table, while the consumer gets
no access. This specification is strictly weaker than the
previous one, but in practice is often good enough. It only
takes a few lines of reasoning to prove that fold_spec
follows from fold_spec_ro. Starting from the assertion
h -> Table M, we expand the definition of Table (Figure 8,
line 40) and obtain h -> TableInState M s, for a fresh s,
which names the current concrete state of the hash table.
We then apply fold_spec_ro to justify the call to fold.
(The consumer gets access to h -> TableInState M s,
but, using the frame rule, we hide this assertion from him.)
Finally, by re-introducing an existential quantifier, we move
from h -> TableInState M s back to h -> Table M.

Since fold is implemented using two nested loops, its
proof requires exhibiting two loop invariants. Fortunately,
both can be obtained as specializations of the invariant
that we need in the proof of cascade. Thus, we are able to avoid
most of the duplication of effort between fold and cascade.
A more elegant way of avoiding this duplication would be to
define fold in terms of cascade, using a generic combinator
that converts a finite cascade into a fold. Our cascade library

\(^{15}\) The documentation of OCaml’s standard library module Hahtbl makes
this guarantee. This illustrates a situation where the production order is
partly, but not fully, determined.
Theorem cascade_spec:
forall h M s, app MK.cascade [h]
INV (h -> TableInState M s)
POST (fun c =>
  c -> Cascade
  (h -> TableInState M s)
  (permitted M) (complete M)
  nil)
).

---

Figure 13. The specification of cascade

A cascade. Furthermore, we wish to allow a cascade to produce an infinite sequence of elements. Therefore, the formal definition of cascades should be co-inductive. We reflect this in Coq via an impredicative encoding, that is, via an existential quantification (Figure 12, line 7). In effect, Cascade is defined as the greatest separation logic predicate S that satisfies the following three conditions:

- The assertion S xs c is duplicable (that is, it entails S xs c \* S xs c). This means that cascades must have no uniquely-owned internal state.\(^\text{16}\)
- The assertion S xs c allows deducing permitted xs. Thus, at every time, the consumer may assume that the sequence of elements produced so far is permitted.
- Conjoined with the invariant I, the assertion S xs c allows invoking the function c. This call must preserve S xs c \* I\(^\text{17}\) and must return either Nil, in which case the consumer may assume that the sequence xs of the elements produced so far is complete, or Cons x c, where c is a valid cascade which is expected to produce a continuation of the sequence xs & x.

4.8 Iteration via Cascade, for Hash Tables

The specification of the cascade function for hash tables appears in Figure 13. Like fold_spec_ro, this function requires the table to be in a specific concrete state, named s; it requires (and preserves) h -> TableInState M s (line 4). It returns a function c, which is a valid cascade.

This cascade has invariant h -> TableInState M s (line 7), which means that it remains valid (and usable) only as long as the table remains in state s. In the contrapositive, this means that any update of the hash table implicitly invalidates all existing cascades. On the other hand, a call to an operation whose specification explicitly guarantees that the table is

\(^\text{16}\)This restriction simplifies reasoning about cascades, as it means that every cascade must be persistent (therefore, can be aliased without danger). Cascades whose implementation involves memoization (thunks) can be made to fall within the scope of this restriction. However, this rules out ephemeral cascades. We leave it to future work to remove this restriction.

\(^\text{17}\)This means, roughly speaking, that the call must have no side effect. In particular, the cascade that has just been queried is still valid, and can be queried again, if desired.

---

\begin{verbatim}
1 Variable A: Type.
2 Variable I: hprop.
3 Variable permitted: list A -> Prop.
4 Variable complete: list A -> Prop.
5
6 Definition Cascade xs c :=
7  Exists S : list A -> func -> hprop,
8  S xs c \* ...
9  \[
10    forall xs c, duplicable (S xs c) \* ...
11    \[
12      for all xs c, ...
13      app c [tt]
14      INV (S xs c \* I)
15      POST (fun c =>
16        match o with
17          | Nil =>
18            \[
19              complete xs
20            \]
21          | Cons x c =>
22            S (xs & x) c
23        end) \].

Figure 12. A generic specification of cascades
\end{verbatim}
not altered, such as population, find, fold, etc. does not invalidate the cascades in existence.

The sequences of elements that this cascade can produce are described by permitted $M$ and complete $M$, whose definitions were given earlier (Figure 11, lines 1–4).

The final nil (Figure 13, line 9) means that no elements have been produced yet.

Although there is not enough space to describe the proof of cascade, let us say that, in order to prove that cascade produces a valid cascade, we must provide a witness for the existential quantifier $\exists c$ (Figure 12, line 7). We remark that every (sub-)cascade that we construct is of the form $\text{fun } () \to \text{cascade_aux data i b.}$ So, we define $S \equiv c$ to mean that $c$ is a closure of this form, for certain values of $data, i$ and $b$, and we add a constraint relating $xs$ (the elements produced already) with $data, i$ and $b$ (which together form a “pointer” into the data structure).

5. Related Work

Proofs of pure programs Several proof assistants, including Coq and Isabelle/HOL, are also purely functional programming languages, where one can implement algorithms and prove them correct. These algorithms can be either executed within the proof assistant or translated to another programming language, such as OCaml, SML, or Haskell. In the Coq world, examples of purely functional, verified data structures include sets and maps, implemented as binary search trees [10]. In the Isabelle world, the Archive of Formal Proofs contains many examples.

The Isabelle Collections Framework [22, 26] identifies several abstract concepts, such as sequences, sets and maps, of which it offers efficient pure implementations, based on binary search trees, hash tables, tries, etc. A programmer who wishes to use the framework expresses her intent at the level of mathematical sets and maps and relies on a refinement machinery [19] to pick suitable implementations.

Régis-Gianas and Pottier [33] describe a Hoare logic which cannot reason about side effects, but tolerates them, including mutable state, nondeterminism, and divergence. They verify an implementation of sets as binary search trees, including a fold function and persistent iterators, with nondeterministic specifications: the order in which elements are enumerated is unspecified. They do not have a universal type of iterators or generic specifications of folds and iterators.

Proofs of imperative programs Nanevski et al. [28] describe Ynot, a higher-order separation logic, embedded in Coq via a monad. They prove the correctness of two imperative implementations of maps, based respectively on hash tables and on splay trees. Their code is polymorphic in the types of keys and values, and uses type abstraction to protect its implementation details. They have a fold operation, whose specification is unfortunately arguably rather complicated and does not allow read access to the data structure while iteration is in progress. They do not have an iterator. The Imperative/HOL framework [2] equips Isabelle/HOL with the ability to produce imperative code. It offers a monad within which one can use references, arrays, and exceptions. It is however restricted to references of “first-order type”, which means that computations cannot be stored in the heap.

Lammich [20, 21], based on earlier work with Meis [23], develops a separation logic on top of Imperative/HOL. Using this logic, he extends the Isabelle Collections Framework with verified imperative data structures, such as a heap-based implementation of priority maps. This approach to verifying imperative data structures and algorithms is comparable with ours insofar as they are both based on separation logic. They differ in that we write executable code first (dividing it into several modules, if necessary, to achieve separation of concerns and impose abstraction barriers) and afterwards prove it correct with respect to a high-level specification, whereas Lammich first writes high-level code which he proves correct, and later (via one or more explicit or automated refinement steps) transforms this code into an executable form.

Proofs of folds and iterators Specifications for folds can be found, for instance, in Régis-Gianas and Pottier’s work [33] as well as the Isabelle Collections Framework [22]. There, the invariant $I$ is parameterized over the set of remaining elements, whereas, here (Figure 10), it is parameterized over the sequence of past elements. When iterating over a data structure, these approaches are equally expressive. Our specification style may be slightly more general in that it should also be able to describe nondeterministic producers whose set of elements is not determined in advance.

Charguéraud [3, Section 4.4] proposes a specification for a fold function, named $\text{iter}$, on (mutable) lists. It is a low-level specification, in that the user is not required to provide a loop invariant; instead, the specification states that the effect of $\text{iter } f \ xs$ is the sequential composition of the effects of the calls $f \ x$, where $x$ ranges over the elements of the list $xs$. Besides, Charguéraud considers a “deep” list (that is, a list that owns its elements) and allows the function $f$ to mutate the elements. In contrast, our specification of fold does not mention the ownership of the elements; it is up to the user to reason about it.

Although the Isabelle Collections Framework encourages the use of folds, nothing in it seems to prevent the use of iterators. In fact, Lammich and Meis’ work [21] includes iterators on mutable lists and on hash tables. Their iterators are restricted, though, in that the iterator owns the underlying data structure, which implies that at most one iterator at a time can exist and that the data structure cannot be accessed while an iterator is active. Also, Lammich and Meis do not propose a universal type or specification of iterators.

Krishnaswami et al. [18] propose a specification and proof, in higher-order separation logic, for mutable lists equipped with implicit iterators. Multiple iterators can exist at once, and are invalidated if the underlying collection is modified. Two functions which create iterators out of iterators, namely
filter and map2, are supported. Krishnaswami et al. do not propose a universal type or universal specification of iterators. Furthermore, because they parameterize the abstract predicate for iterators with the list of elements that the iterator will produce, their iterators are deterministic and finite.

Haack and Hurlin [12] present several generic specifications, in separation logic with fractional permissions, for Java iterators. They consider both read-only and read-write iterators (which have a remove method), allow multiple read-only iterators to co-exist, and allow a lone read-only iterator to become read-write. They consider both “shallow” and “deep” collections, whereas, by saying nothing about the ownership of the elements, we have considered only the former situation. Haack and Hurlin’s specifications focus on ownership transfer and ignore functional correctness: they do not specify which sequences of elements an iterator must (or may) produce.

Filliâtre and Pereira [9] propose a generic specification of implicit iterators (under the name of “cursors”) and verify several iterator implementations and clients using Why3. The style in which we specify the set of possible behaviors of a producer, which supports nondeterminism as well as infinite behaviors, is inspired by their work: indeed, our predicates permitted and complete correspond roughly to enumerated and completed there. We show that this style can be used to specify not only iterators, but also folds, and, more generally, any kind of (possibly nondeterministic, possibly infinite) producer.

Polikarpova et al. [30] prove the functional correctness of the general-purpose data structure library EiffelBase2. The library includes a hierarchy of classes for various kinds of iterators. The base class V_INPUT_STREAM has three deferred methods, namely off (are we at the end?), item (what is the current item?), and forth (move forward). At this level, there is no specification of the sequences of elements that the iterator is allowed to produce. One level down in the hierarchy, V_ITERATOR describes a bidirectional iterator, backed by a data structure whose model is a finite sequence. Such an iterator is therefore deterministic. This class offers many methods, with specifications. The library includes an implementation of hash tables, including iterators. The class V_HASH_TABLE_ITERATOR inherits from V_ITERATOR, which seems disputable, since a hash table in principle represents a dictionary, not a sequence. As far as we can tell, the specification of the hash table iterator is not abstract: it reveals that the sequence of keys produced by the iterator is the concatenation of the keys found in all buckets. Ideally, the specification shown to the user should not even mention “buckets”, which are an implementation detail.

6. Conclusion
We have described the specification and proof, using CFML and Coq, of a hash table implementation. Iteration via folds and via cascades (a form of iterators) is supported. Multiple cascades can exist simultaneously and are valid as long as the table is not modified. We have given generic specifications of folds and cascades, which we have instantiated for hash tables. We have shown that, whichever iteration mechanism is chosen, the space of legal sequences can be specified via permitted and complete predicates.

Some strengths of CFML are that it allows writing OCaml code exactly in the desired form, does not require littering the code with annotations, and, if desired, allows establishing multiple specifications for a single function. Its main weakness is that interactive proof still requires considerable expertise and effort.

Much work remains to be done, on hash tables and on the Vocal project.

We should verify one or more program components that use hash tables, so as to experimentally confirm that our proposed specification of hash tables is indeed as strong as we believe it is.

Our specification of hash tables effectively requires keys to be immutable.18 It is agnostic as to whether values are immutable or mutable: it is up to the user to reason about the ownership of values, if necessary. We might wish to study “deep” or “nested” tables [12, 26], that is, to allow or facilitate scenarios where keys and/or values are owned by the hash table. This requires transfers of ownership between the table and its user, whose description seems challenging.

We should verify many more examples of producers and consumers, so as to confirm that our proposed specifications of folds and cascades are general enough, and are not so strong that they cannot be implemented, or so weak that they cannot be used. In particular, we would like to develop and verify a full-fledged cascade library, along the lines of Haskell’s list library or Jane Street’s Core.Sequence. Such a library would contain many cascade combinators which act as producers, consumers, or both at the same time.

Our generic specification of cascades covers only persistent cascades. We would like to relax it to cover ephemeral cascades2 as well. We would also like to investigate whether we could tolerate updating a mutable data structure while iteration is in progress, provided the updates preserve the producer’s invariant.

We currently use OCaml’s “safe” array access operations, which perform a runtime array bounds check. We would like to remove these checks when they are provably redundant. This is not as simple as it may sound, as we would like to guarantee memory safety even in the presence of unverified client code, which may violate our preconditions.

We currently pretend that OCaml integers are unbounded, which is not true: they are 31- or 63-bit integers in two’s-complement representation. We would like to plug this hole without creating undue clutter in our proofs, perhaps by exploiting Clochard et al.’s ideas [6].

18 Indeed, according to Figure 7, the user-supplied functions equneq and hash must be able to exploit a key without receiving any access permission for it. This rules out mutable keys, which in separation logic are governed by a unique access permission.
References


