Verifying a hash table and its iterators in higher-order separation logic

François Pottier
Inria Paris *
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, translated to Coq by the CFML tool, and verified using higher-order separation logic, embedded in Coq.

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

* This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

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 [3], 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:

• The data structure is abstract: the client must be aware that it represents a dictionary, but must not 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 [20, 18, 19] 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.

• 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).

• The data structure permits iteration via two distinct mechanisms, namely “folds” (which are 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. The specifications of folds and cascades are necessarily somewhat complex: both involve first-class functions, read-only access to a mutable data structure, nondeterminism (the existence of several permitted sequences of elements), and unboundedness (the existence of infinite permitted 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 (non-blank, non-comment) lines of OCaml code. It offers the same op-
eralizations as the functor 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 [22].

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” [9] is a producer of a sequence of elements with 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 [16, 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.

We distinguish two kinds of iterators, which, although closely related, lead in practice to very different coding styles. An “implicit iterator” is inherently modifiable. When queried, it implicitly consumes the element that it returns: a subsequent query to this iterator produces the following element of the sequence. An “explicit iterator” does not do so: a subsequent query produces the same element again (if such a query is permitted1). In addition to an element, an explicit iterator returns another explicit iterator, which gives access to the remainder of the sequence.

Several programming languages promote implicit iterators as part of their standard libraries. In Java, for instance, the Iterator interface requires iterators to offer a method next, which consumes and returns the next element, if there is one. It also requires a method hasNext, which tells whether one more element is available.

Whereas implicit iterators must have mutable state, explicit iterators can often be implemented without side effects. In our experience, this makes them easier to build and to reason about. For this reason, we favor explicit iterators.

Furthermore, we drop the requirement that iterators must offer a hasNext method. With implicit iterators, hasNext can be quite useful, as it allows determining whether an element is available without consuming that element. With (multiple-use) explicit iterators, one can simply use next for this purpose.

Thus, an iterator offers just one method, namely next. In other words, an iterator is a function. It takes no argument (or, in OCaml, an argument of type unit) and returns either nothing or a pair of an element and another iterator. We refer to this particular kind of iterators as “cascades”. Their definition appears in Figure 1. The auxiliary type 'a head represents a response, that is, either nothing (Nil) or a pair of an element and a cascade (Cons). A cascade is a function which, when queried, returns a response.

If one replaced unit -> 'a head with just 'a head on line 3 of Figure 1, then 'a head would be isomorphic to '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 [8] 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 [4] in advocating explicit iterators. Although he briefly considers adopting delayed lists as the universal type of sequences [4, §4], motivated by efficiency considerations, he settles for distinct, ad hoc abstract types of tree iterators [4, §2], hash table iterators, and so on.

---

1 A second query could be forbidden. Indeed, an explicit iterator could come with the proviso that it can be used at most once. The reason could be that this iterator has mutable internal state or affects the outside world. With this in mind, explicit iterators are no less general than implicit iterators.
module type HashedType = sig
  type t
  val equal : t -> t -> bool
  val hash : t -> int
end

module Make (K : HashedType) : sig
  type key = K.t
  type '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

Jane Street’s library Core.Sequence follows the same approach, but, thanks to an existential quantification over the type of the iterator’s internal state, is able to define a single abstract type of all sequences. This type is interconvertible with 'a cascade. We use existential quantification in the same way, at the specification level, to give a nonrecursive definition of the abstract predicate Cascade (§4.6).

3. Hash tables

The signature and implementation of our HashTable module are shown in Figures 2 and 3. They are modeled after the Hashtbl module found in OCaml’s standard library, with a few minor differences.

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, Make, which takes a module K of signature HashedType as an argument (Figure 2). The type of keys, key, is defined as a synonym for K.t. There are no requirements on the type of values. For this reason, the type of hash tables, 'a t, is parameterized with 'a, the type of values.

module Make (K : HashedType) = struct
  (* Type definitions. *)
  type key = K.t
  type 'a t = ('a , 'b , 'c) 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 3. The implementation file HashTable.ml

The type of hash tables, 'a t, is defined internally as a record of three fields (Figure 3), namely: a data array, data; an integer population count, popu; and an integer initial capacity, 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 that the entry that was most recently added appears earliest in the list and is the one returned by find.

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 1 b, where data is the data array, i is the index of the most recently fetched

2 At version 4.03, OCaml’s 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.

3 Upon reflection, this feature seems of dubious interest. If heavily used, it could lead to a degradation in 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).
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

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

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 4. Implementation of insertion

Figure 5. Implementation of iteration via fold

bucket, and b is the suffix of that bucket that remains to be traversed. They are immutable, therefore persistent. Let us 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.
Require Import HashTable_ml.

Module Type HashedTypeSpec.
  Include HashedType_ml.
  Notation key := t_.
  Parameter E : key -> key -> Prop.
  Parameter Eequiv : equiv E.
  Parameter H : key -> int.
  Parameter hash_spec: computes hash H.
End HashedTypeSpec.

Module MakeSpec (K : HashedTypeSpec).
  Import K.
  Module MK := Make_ml(K).
  Variable A : Type.
  (* Invariant: see Figure 8. *)
  (* Specifications and proofs. *)
  (* add: see Figure 9. *)
  (* fold: see Figure II. *)
  (* cascade: see Figure 13. *)
End S.
End MakeSpec.

Figure 7. The spec and proof file HashTable_proof.v

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 \([t]_\).

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 now clarify this aspect. 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 \(a\) is viewed in Coq as an (ideal) integer value, \(a\) is reflected as \(\text{int}\) of type \(\text{int}\), is viewed in Coq as an ideal integer value.

For instance, the type \(\text{int}\) is compatible with \(\text{int}\) and \(\text{int}\) hash \(\text{int}\) computes hash \(\text{int}\). 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 handwritten file 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, \text{Make}, whose parameter \(K\) has signature HashedType, this file defines a functor, \text{MakeSpec}, whose parameter \(K\) has signature HashedTypeSpec. This signature, which we define, extends HashedType_ml and expresses our requirements about the OCaml functions \(\text{equal}\) and 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 equal hashes); (2) \(\text{equal}\) decides key equivalence; and (3) \(\text{hash}\) computes a key’s hash.

Inside the body of the functor \text{MakeSpec}, we apply the functor \text{Make_ml} (the auto-generated Coq counterpart of the OCaml functor \text{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 \(\text{’a}\). The specifications and proofs in this section become polymorphic in \(A\) when the section ends.

We can now define the separation logic predicates \(\text{Table}\) and \(\text{TableInState}\), 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), and cascade (§4.6), 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

5 The Coq signature HashedType_ml is the auto-generated Coq counterpart of the OCaml signature HashedType. It requests the existence of a type \(t_\) and of two functions \(\text{equal}\) and \(\text{hash}\) of type function.

6 The predicate \text{Proper} is part of Coq’s standard library. The predicates \text{computes} and \text{decides}, defined by CFML, are abbreviations for Hoare triples.

An OCaml module named \(M\) is transformed to a Coq module named \(M_ml\). Module types are similarly renamed. Thus, the OCaml module type HashedType is translated to a Coq module type named HashedType_ml.

For more details about CFML, the reader is referred to Charguéraud’s paper [2].

4.2 Setup

The OCaml code in the file HashTable.ml is transformed by the CFML generator into characteristic formulae, stored in the Coq file 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 handwritten file 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 function, \text{Make}, whose parameter \(K\) has signature HashedType, this file defines a function, \text{MakeSpec}, whose parameter \(K\) has signature HashedTypeSpec. This signature, which we define, extends HashedType_ml and expresses our requirements about the OCaml functions \(\text{equal}\) and 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 equal hashes); (2) \(\text{equal}\) decides key equivalence; and (3) \(\text{hash}\) computes a key’s hash.

Inside the body of the functor \text{MakeSpec}, we apply the functor \text{Make_ml} (the auto-generated Coq counterpart of the OCaml functor \text{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 \(\text{’a}\). The specifications and proofs in this section become polymorphic in \(A\) when the section ends.

We can now define the separation logic predicates \(\text{Table}\) and \(\text{TableInState}\), 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), and cascade (§4.6), 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

5 The Coq signature HashedType_ml is the auto-generated Coq counterpart of the OCaml signature HashedType. It requests the existence of a type \(t_\) and of two functions \(\text{equal}\) and \(\text{hash}\) of type function.

6 The predicate \text{Proper} is part of Coq’s standard library. The predicates \text{computes} and \text{decides}, defined by CFML, are abbreviations for Hoare triples.
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_ A7 (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 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. This implies Proper (E ==> eq) M, a property of M that was pointed out earlier. The proposition no_garbage data (lines 10–14) states that the array data contains no other key-value pairs. The proposition table_inv M init data (lines 16–21) combines the properties discussed up to this point, and records the fact that the length of the array data is a power of two.

The above propositions are pure: they have type Prop. We now wish to define an assertion, of type hprop, which asserts that a well-formed hash table exists in the heap. More specifically, we would like the assertion h -> Table M to hold if the heap contains at address h a hash table that represents the dictionary M (and does not contain anything else). Furthermore, we would like to define a more informative assertion h -> TableInState M s, meaning that the hash table h represents the dictionary M and is in the concrete state s. Our purpose is to be able to express the policy that “updating the hash table invalidates all existing iterators”, or in other words, that “concurrent modifications” are forbidden. An update could affect either the data field of the record h or the array h.data. Thus, we let a state s be a pair (d, data), and define the type state accordingly (Figure 8, lines 23–24). We define the assertion h -> TableInState M s as a separating conjunction of the (uniquely-owned) record at address h (lines 30–34), the (uniquely-owned) array at address d (line 35), the pure invariant that was previously dis-

7This 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.

8We 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.
Theorem add_spec:
  forall M h k x,
  app MK.add [h k x]
  \<\!
  \begin{array}{c}
  \text{PRE (h -> Table M)}
  \\
  \text{POST (fun _ => Hexists M', h -> Table M')}
  \\
  \end{array}
  \!\\! \!
  \begin{array}{c}
  \langle M' = \text{add } M \ k \ x \rangle
  \\
  \langle \text{lean } M -> M \ k = \text{nil} -> \text{lean } M' \rangle.
  \end{array}
\]

\textbf{Figure 9.} The specification of add

cussed (line 36), a constraint on the popu field\(^9\) (line 37), and the equation \(s = (d, \text{data})\) (line 38). Finally, the assertion \(h \rightarrow \text{Table } M\) is defined simply by abstracting away the concrete state, that is, by quantifying existentially over \(s\) in \(h \rightarrow \text{TableInState } M\ s\).

A user of the HaskellTable module must be aware of the meaning of the separation logic assertions \(h \rightarrow \text{Table } M\) and \(h \rightarrow \text{TableInState } M\ s\), as these assertions appear in the specifications of the hash table operations (§4.4–§4.6). She must understand that a table \(h\) represents a dictionary \(M\) of type \(\text{key} \rightarrow \text{list } A\). She must understand that a hash table is, at every moment, in a certain concrete state \(s\), and that some operations (such as fold and cascade) do not affect its concrete state. That is all a user needs to know. She should view Table and TableInState as abstract predicates, and view state as an abstract type. The concrete definitions of these abstract entities are of course used in our proof, but are not part of the specification of the HaskellTable module.

\subsection*{4.4 Insertion}

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 \(\text{MK.add}\).

We expect this triple to express the informal idea that "if \(h\) is a hash table, then the function call \(\text{add } h \ k \ x\) 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 \(h \rightarrow \text{Table } M\) expresses an assumption that the table initially represents a dictionary \(M\). The conjunct \(h \rightarrow \text{Table } M'\) 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 \(M' = \text{add } M \ k \ x\). This equation refers to an add operation that we define in Coq at the level of dictionaries. Its two-line definition (not shown) says that \(M' \ k'\) is \(x :: M \ k'\) if the keys \(k\) and \(k'\) are equivalent, and is \(M \ k'\) otherwise.

\(^9\)population \(M\) is defined as the sum of the lengths of the lists \(M \ k\), where \(k\) ranges over some domain \(D\) of \(M\). This sum does not depend on the choice of \(D\), that is, on the choice of a representative element in each equivalence class of keys.

\begin{verbatim}
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]  
  \<\!
  \begin{array}{c}
  \text{PRE (S c \* I xs accu)}
  \\
  \text{POST (fun accu => Hexists xs,}
  \text{S c \* I xs accu \*}
  \\
  \text{\langle \text{complete } xs \rangle).}
  \end{array}
\]

\textbf{Figure 10.} A generic specification of fold functions

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 \(\text{lean } M\) means \(\forall k, \text{length } (M \ k) \leq 1\). The implication \(M' \rightarrow M' = \text{nil} \rightarrow \text{lean } M'\) 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 simple lemma about the dictionary-level function add. Building it into the postcondition of the OCaml function add is not necessary, but saves the user the trouble of manually applying this lemma.

\subsection*{4.5 Iteration via fold}

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 \(f, c, \text{and } \text{accu}\), where:

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

\[
\begin{array}{c}
\end{array}
\]
• $f$ 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 $I\text{ nil accu}$, which means that the user must establish the invariant (of the empty list, and of the initial accumulator). Its postcondition contains $I\text{ xs accu}$, which means that, at the end, the invariant still holds (of the list $xs$ 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 list of elements $xs \cdot x$ and 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 $S \cdot c$ 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 $S' \cdot c$ in the pre- and postcondition of $f$.

The parameter call encodes the calling convention of the function $f$. In the simplest scenario, it is instantiated in such a way that call $f \cdot x \cdot accu$ expands to $app \cdot f \cdot [x \cdot accu]$: 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 $f \cdot (k, x) \cdot accu$ is $app \cdot f \cdot [k \cdot x \cdot accu]$ (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,

\begin{itemize}
  \item 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.
  \item 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.
\end{itemize}

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 $xs \cdot ys$ 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 [23, 14]. As observed by Filliâtre and Pereira [5], this is described by defining permitted $xs$ to mean “the elements of $xs$ are infinite 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 $(xs \cdot k \cdot x)$ 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 seen so far, including $x$, is permitted. Dually, every time it wishes to produce some element $x$, the producer must prove that extending the sequence of elements seen so far with $x$ is permitted.

The proposition complete $xs$ in the postcondition of fold (Figure 10, line 25) means that, once the consumer observes that fold has terminated, it may assume that the sequence of elements seen so far is complete.

We have defined the specification of a “fold”, once and for all. Let us now see how this specification is instantiated in the case of hash tables. This is done in Figure 11.

The first thing is to declare which sequences of key-value pairs may be observed by the user. This is done by choosing appropriate instantiations of permitted and complete. It is clear that the specification must be nondeterministic: as we do not control the hash function, we cannot know ahead of time in which order the key-value pairs will be discovered as the data array is scanned. We could adopt a fully nondeterministic specification, where any permutation of the multiset of key-value pairs in the dictionary $h$ is permitted. Yet, one thing we can guarantee is that, if there are several key-value pairs for a single key $k$ (that is, if the list $h \cdot k$ contains more than one element), then these pairs are presented to the consumer in a most-recent-first fashion [12]. So, we choose to specify that “the order in which the key-value pairs are produced corresponds to a possible sequence of removals”.

\begin{itemize}
  \item A typical invariant might be: “$accu$ is the sum of the elements $xs$ that have been processed so far”.
  \item If $S' \cdot c$ is $S \cdot c$, then the consumer has full access to the collection. If $S' \cdot c$ is the empty heap assertion $[]$, then the consumer has no access to it.
\end{itemize}
We define the predicate \( \text{removal} \ M \ \text{kxs} \ M' \) to mean that, starting from the dictionary \( M \), it is possible to remove the key-value pairs in the sequence \( \text{kxs} \), one after the other, and that this process yields the dictionary \( M' \). This definition is based on a function \( \text{remove} \ M \ k \), which is defined at the level of dictionaries, just like the dictionary-level add \((\S 4.4)\), 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 \(\text{Figure 11, lines } 1-4\).

We are now ready to state the specification of the \( \text{fold} \) function on hash tables, as an instance of \( \text{Fold} \), which was defined in \(\text{Figure 10} \). In fact, we make two such statements, both of which are instances of \( \text{Fold} \). The first statement, \( \text{fold.spec.ro} \), gives the consumer read-only access to the hash table, and guarantees that \( \text{fold} \) itself does not modify the table. The second statement, \( \text{fold.spec} \), 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 \( \text{fold.spec.ro} \), the parameters \( S \) and \( S' \) of \(\text{Figure 10} \) are instantiated with \( \text{fun } h \Rightarrow h \Rightarrow \text{TableInState} \ M \ s \) 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 \( \text{fold.spec} \), this time, the parameter \( S \) of \(\text{Figure 10} \) is instantiated with \( \text{fun } h \Rightarrow h \Rightarrow \text{Table} \ M \), whereas \( S' \) is instantiated with \( \text{fun } h \Rightarrow \emptyset \). 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 \( \text{fold.spec} \) follows from \( \text{fold.spec.ro} \). Starting from the assertion \( h \Rightarrow \text{Table} \ M \), we expand the definition of \( \text{Table} \) \(\text{Figure 8, line } 40 \) and obtain \( h \Rightarrow \text{TableInState} \ M \ s \), for a fresh \( s \), which names the current concrete state of the hash table. We then apply \( \text{fold.spec.ro} \) to justify the call to \( \text{fold} \). (The consumer gets access to \( h \Rightarrow \text{TableInState} \ M \ s \), but, using the frame rule, we hide this assertion from him.) Finally, by re-introducing an existential quantifier, we move from \( h \Rightarrow \text{TableInState} \ M \ s \) back to \( h \Rightarrow \text{Table} \ M \).

Since \( \text{fold} \) 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 \( \text{cascade} \). Thus, we are able to avoid most of the duplication of effort between \( \text{fold} \) and \( \text{cascade} \). A more elegant way of avoiding this duplication would be to define \( \text{fold} \) in terms of \( \text{cascade} \), using a generic combinator that converts a finite cascade into a fold. Our cascade library offers such a combinator. However, as of now, the OCaml compiler is not able to optimize this indirect definition of \( \text{fold} \) as much as we would like, so we stick with a direct definition.

### 4.6 Iteration via \text{cascade}

As in the previous section, it is worth defining the concept of a “cascade”, once and for all, in a general setting. The function \( \text{cascade} \) \(\text{Figure 6} \), which constructs a cascade of all key-value pairs in a hash table, is just an instance of this general concept.

So, what is a “cascade”? As explained earlier \((\S 2)\), it is an on-demand producer of a sequence of elements. More precisely, it is a function, which, when invoked (with an argument \(c\)) returns either \(\text{Nil}\) or \(\text{Cons}\) accompanied with an element and another cascade. This description is translated into a formal definition, and made more precise, in \(\text{Figure 12} \). There, we define the assertion \( \text{Cascade} \ \text{xs} \ c \), which can also be written \( c \rightarrow \text{Cascade} \ \text{xs} \). This assertion means that \( c \) is a valid cascade and that the sequence of elements \( \text{xs} \) has already been produced, so that \( c \) is now expected to produce a legal continuation of \( \text{xs} \).

Like \( \text{Fold} \) \((\S 4.5)\), this definition is parameterized over the predicates \( \text{permitted} \) and \( \text{complete} \), which, together, specify which finite or infinite sequences can be observed. It is also parameterized over an invariant \( I \), which typically describes a data structure that the cascade needs to access,
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

where c is a valid cascade which is expected to produce a continuation of the sequence xs \& x.

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 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 Hexists S (Figure 12, line 7). We remark that every (sub-)cascade that we construct is of the form fun () -> cascade_aux data i b. So, we define S xs c to mean that c is indeed a closure of this form, for certain values of data, i and b, and we add a constraint that relates xs (the elements produced already) with data, i and b (which together form a “pointer” into the concrete data structure).

5. Related work

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 [6]. In the Is-
abelle world, the Archive of Formal Proofs contains many examples.

Régis-Gianas and Pottier [23] 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 immutable 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.

Nanevski et al. [19] 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 Isabelle Collections Framework [14, 17] identifies several abstract concepts, such as sequences, sets and maps, of which it offers efficient 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 [11] to pick suitable implementations. The code thus obtained is pure.

The Imperative/HOL framework [1] 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 [12, 13], based on earlier work with Meis [15], integrates these lines of work. On top of Imperative/HOL, he develops a full-fledged separation logic. 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 into executable code.

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 [15] 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. [10] 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 [7] 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 [5] 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. [21] 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 (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.

Much work remains to be done, on hash tables and on the Vocal project. We should verify clients of the hash table module, so as to confirm that our specifications are usable. We would like to remove array bounds checks when they are provably redundant, and to safely do so even in the presence of unverified client code. We would like to account for the gap between machine integers and ideal integers, which we have ignored so far. We would like to study “deep” or “nested” tables [7, 17], that is, to allow reasoning about a situation where keys and/or values are mutable and uniquely owned. We would like to develop and verify a full-fledged cascade library, which would be analogous to Haskell’s list

References


