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

François Pottier

Vocal meeting
Paris, November 10, 2016

Motivation

Why verify a hash table implementation ?

- a simple and useful data structure
- implements an abstract concept : a dictionary
- dynamically allocated ; mutable
- parametric in an ordered type of keys and a type of values
- equipped with two iteration mechanisms : fold, cascade

A glimpse of the OCaml code

A glimpse of the Coq invariant and specifications

Interface

```
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
```

Cascades

An iterator is an on-demand producer of a sequence of elements.

A cascade, or delayed list, is a particular kind of iterator.

```
type 'a head =
| Nil
| Cons of 'a * (unit -> 'a head)

type 'a cascade =
  unit -> 'a head
```

This type definition does not reveal :

- whether a cascade has mutable internal state, or is pure ;
- whether elements are stored in memory, or computed on demand ;
- whether elements are re-computed when re-demanded, or memoized.

My credo :

> *Cascades are easy to build and use because they are "just like lists".*

Type definitions

A hash table is a record whose `data` field holds a pointer to an array of buckets.

A bucket is an immutable list of key-value pairs.

```
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: see next slides... *)
  ...
end
```

Iteration via fold

The higher-order function `fold` is implemented by two nested loops.

The inner loop is implemented as a tail-recursive function.

```
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
```

Iteration via a cascade

Constructing a cascade is like constructing a list of all key-value pairs.

```
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
```

A glimpse of the OCaml code

A glimpse of the Coq invariant and specifications

Invariant

We define two abstract predicates, `h ~> TableInState M s` and `h ~> Table M`.

This is used later on to demand / guarantee that certain operations are read-only.

```
Implicit Type M : key -> list A.
Implicit Type h : MK.table_ A.
Implicit Type d : loc.
Implicit Type data : list (MK.bucket_ A).

Definition TableInState M s h :=
  Hexists d pop init data,
  h ~> `{                              (* the record *)
    MK.data' := d;
    MK.popu' := pop;
    MK.init' := init
  } \*
  d ~> Array data \*                   (* the array *)
  \[ table_inv M init data ] \*        (* the data invariant *)
  \[ population M = pop ] \*
  \[ s = (d, data) ].                  (* the "concrete state" *)

Definition Table M h :=
  Hexists s, h ~> TableInState M s.
```

Specification of insertion

The effect of `add h k x` is to add the key-value pair `(k, x)` to the dictionary.

```
Theorem add_spec:
  forall M h k x,
  app MK.add [h k x]
    PRE  (h ~> Table M)
    POST (fun _ => Hexists M',
    h ~> Table M' \*
    \[ M' = add M k x ] \*
    \[ lean M -> M k = nil -> lean M' ]).
```

The first-order operations (`remove`, `clear`, ...) have "simple" specifications like this.

Generic specification of iteration order

A generic specification of an iteration mechanism (fold, cascade, . . .)
must be parameterized with a set of possible observations.

The events that can be observed are :

- the production of one element ;
- the production of an end-of-sequence signal.

An observation could be viewed as a series of events
(where an end-of-sequence event, if present, must be the last event).

Alternatively, a set of observations can be directly encoded using two predicates :

```
Variables permitted complete : list A -> Prop.
```

Hash table iteration order

We give concrete definitions of `permitted` and `complete` for our hash table iteration mechanisms.

They are semi-deterministic :

- two key-value pairs for different keys may be observed in any order ;
- two key-value pairs for the same key must be observed most-recent-value-first.

```
Definition permitted kxs :=
  exists M', removal M kxs M'.
Definition complete kxs :=
  removal M kxs empty.
```

Generic specification of iteration via fold

```
Variable  fold : func.
Variables A B C : Type.
Variable  call : func -> A -> B -> ~~B.
Variables permitted complete : list A -> Prop.
Variable  I : list A -> B -> hprop.
Variables S 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 ]).
```

Specification of hash table iteration via fold

The specification of `fold` for hash tables is a special case :

```
Theorem fold_spec_ro:
  forall M s B I,
  Fold MK.fold
    (* Calling convention: *)
    (fun f kx (accu : B) =>
       app f [(fst kx) (snd kx) accu])
    (* Permitted/complete sequences: *)
    (permitted M) (complete M) I
    (* fold requires and preserves this,
       so does not modify the table: *)
    (fun h => h ~> TableInState M s)
    (* f receives this and must preserve
       it, hence can read the table: *)
    (fun h => h ~> TableInState M s).
```

This specification allows read-only access to the table during iteration.

Specification of hash table iteration via fold

If access to the table during iteration is not needed, a simpler spec can be given :

```
Theorem fold_spec:
  forall M B I,
  Fold MK.fold
    (fun f kx (accu : B) =>
       app f [(fst kx) (snd kx) accu])
    (permitted M) (complete M) I
    (* fold requires & preserves this: *)
    (fun h => h ~> Table M)
    (* f cannot access the table: *)
    (fun h => \[]).
```

(This spec does not guarantee that the table is unmodified.)

Generic specification of iteration via a cascade

`c ~> Cascade xs` means that `c` is a valid cascade that will produce a valid continuation of `xs`.

Thus, `xs` represents the elements that have been produced already.

```
Variable  A : Type.
Variable  I : hprop.
Variables permitted complete : list A -> Prop.

Definition Cascade xs c :=
  Hexists S : list A -> func -> hprop,
  S xs c \*
  \[ forall xs c, duplicable (S xs c) ] \*
  \[ forall xs c, S xs c ==> S xs c \* \[ permitted xs ] ] \*
  \[ forall xs c,
     app c [tt]
       INV  (S xs c \* I)
       POST (fun o =>
         match o with
         | Nil      => \[ complete xs ]
         | Cons x c => S (xs & x) c
         end) ].
```

`Cascade` is defined as (an impredicative encoding of) a greatest fixed point.

Specification of hash table iteration via a cascade

The specification of `cascade` uses the same predicates `permitted` and `complete` as the specification of `fold`.

```
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
    ).
```

`cascade` produces a cascade that can be used (only) as long as the hash table remains in the concrete state `s`.

That is, "concurrent modifications" are disallowed.

Statistics

- OCaml : under 150loc
- Coq (abstract dictionaries) : 300loc specs, 300loc proofs
- Coq (concrete hash tables) : 700loc specs, 700loc proofs
- total effort : under 15 man.days