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

François Pottier

CPP 2017
Paris, January 16, 2017
We want verified software...
Therefore, we need VERIFIED LIBRARIES.
The Vocal project is building a verified library of basic data structures and algorithms.

▶ The code is in OCaml.
▶ Verification can be done in higher-order separation logic:
  ▶ Charguéraud’s CFML imports a view of the code into Coq;
  ▶ reasoning is carried out in Coq.

In this talk, I focus on one module: a hash table implementation.
Why verify a hash table implementation?

- a simple and useful data structure

Why talk about it today?

- dynamically allocated; mutable
- equipped with two iteration mechanisms: fold, cascade
The data structure

First-order operations

Iteration via fold

Iteration via cascades

Conclusion
OCaml interface

An excerpt of HashTable.mli.

```ocaml
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
```
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
OCaml interface

An excerpt of HashTable.mli.

```ocaml
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
```

Iteration (producer in control)
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
An excerpt of `HashTable.ml`.

```ocaml
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 following slides... *)
end
```
An excerpt of HashTable.ml.

```ocaml
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 following slides... *)

end
```

A hash table is a record...
An excerpt of HashTable.ml.

```ocaml
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 following slides... *)
end
```

...whose data field is an array of buckets...
OCaml implementation

An excerpt of HashTable.ml.

```
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 following slides... *)
end
```
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,
  h ~> '{
    data := d;
    popu := pop;
    init := init
  }
  d ~> Array data /*
  [ table_inv M init data ] /*
  [ population M = pop ] /*
  [ s = (d, data) ].

Definition h ~> Table M :=
  Hexists s, h ~> TableInState M s.
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
Hexists d pop init data,
h ~> '{
data := d;
popl := pop;
init := init
}
/*
d ~> Array data /*
[/ table_inv M init data ] /*
[/ population M = pop ] /*
[/ s = (d, data) ].

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

A table represents a finite map of keys to (lists of) values.
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
Hexists d pop init data,
h ~> '{
data := d;
popu := pop;
init := init
} */
d ~> Array data /*
[ table_inv M init data ] /*
[ population M = pop ] /*
[ s = (d, data) ].

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

This SL predicate asserts “the table at address h encodes the dictionary M”.
Separation logic invariant (in Coq)

An excerpt of `HashTable_proof.v`.

```
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,
  h ~> '{
    data := d;
    popu := pop;
    init := init
}
  \[ table_inv M init data ] /*
  \[ population M = pop ] /*
  \[ s = (d, data) ].

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

This one names s the current concrete state of the table.
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,
  h ~> '{
    data := d;
    popu := pop;
    init := init
  } \[ table_inv M init data \] \*[ population M = pop \]
  \[ s = (d, data) \].

Definition h ~> Table M :=
  Hexists s, h ~> TableInState M s.
Separation logic invariant (in Coq)

An excerpt of `HashTable_proof.v`.

```
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,  
  h ~> '{
    data := d;
    popu := pop;
    init := init
  } 
  \[ d ~> Array data \]
  \[ table_inv M init data \]
  \[ population M = pop \]
  \[ s = (d, data) \].

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

...whose data field contains a pointer d...
Explicit Type M : key \to list A.

Definition h \sto TableInState M s :=
\[ \text{Hexists } d \text{ pop init data,}
\]
h \sto '{
\text{data := d;}
\text{popu := pop;}
\text{init := init}
}
\* d \sto \text{Array data} \*\n\[ \text{table inv M init data} \]*
\[ \text{population M = pop} \]*
\[ s = (d, data) \].

Definition h \sto Table M :=
\text{Hexists } s, h \sto \text{TableInState M s}.
Separation logic invariant (in Coq)

An excerpt of `HashTable_proof.v`.

```
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,
  h ~> '{
    data := d;
    popu := pop;
    init := init
  } /*
  d ~> Array data /*
  \[ table_inv M init data \] /*
  \[ population M = pop \] /*
  \[ s = (d, data) \].

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

The content of memory is related to M.
Implicit Type M : key -> list A.

Definition h ~> TableInState M s :=
  Hexists d pop init data,
  h ~> '{
    data := d;
    popu := pop;
    init := init
  } */
  d ~> Array data \*
  /
  [ table_inv M init data ] \*
  /
  [ population M = pop ] \*
  /
  [ s = (d, data) ].

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

The address and content of the array are exposed under the name s.

We use \( s \) to demand / guarantee that certain operations are \textit{read-only}. 
Implicit Type M : key -> list A.

Definition h -> TableInState M s :=
  Hexists d pop init data,
  h -> `{ d := d;
    popu := pop;
    init := init
  } */
  d ~> Array data /*
  \[ table_inv M init data \] /*
  \[ population M = pop \] /*
  \[ s = (d, data) \].

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

We hide s when we do not care about it.

We use s to demand / guarantee that certain operations are read-only.
The data structure

First-order operations

Iteration via fold

Iteration via cascades

Conclusion
Specifying a first-order operation : insertion

The effect of \( \text{add } h \ k \ x \) is to add the key-value pair \((k, x)\) to the dictionary.

This is stated as a Hoare triple:

\[
\text{Theorem add_spec:}
\]
\[
\text{forall } M \ h \ k \ x, \\
\text{app } MK.\text{add } [h \ k \ x] \\
\text{PRE } (h \rightarrow \text{Table } M) \\
\text{POST } (\text{fun } _ \Rightarrow \text{Hexists } M', \\
\h \rightarrow \text{Table } M' \ast \\
[ M' = \text{add } M \ k \ x ] \ast \\
[ \text{lean } M \rightarrow M' k = \text{nil } \rightarrow \text{lean } M']) \).
\]
The effect of \( \text{add } h \ k \ x \) is to add the key-value pair \((k, x)\) to the dictionary.

This is stated as a Hoare triple:

\[
\text{Theorem add spec: } \\
\forall M \ h \ k \ x, \\
\text{app MK.add } [h \ k \ x] \\
\text{PRE (h ~> Table M) } \\
\text{POST (fun } _ => \text{ Hexists M’,} \\
\text{h ~> Table M’ \*} \\
\text{\[ M’ = add M k x \] \*} \\
\text{\[ lean M ~> M k = nil ~> lean M’ \]).}
\]
Specifying a first-order operation: insertion

The effect of \( \text{add } h \ k \ x \) is to add the key-value pair \((k, x)\) to the dictionary.

This is stated as a Hoare triple:

\[
\text{Theorem add\_spec:}
\forall M \ h \ k \ x,
\text{app } MK.\text{add}[h \ k \ x]
\text{ PRE (h \Rightarrow Table } M)\\
\text{ POST (fun } _ \Rightarrow \text{ Hexists } M',
\ h \Rightarrow \text{ Table } M' \\
\ \boxed{\left[ M' = \text{ add } M \ k \ x \right]} \ \boxed{\left[ \text{ lean } M \Rightarrow M' \mid \text{ nil} \Rightarrow \text{ lean } M' \right]}).
\]

...requires a valid table...
The effect of \texttt{add }\, h \, k \, x \texttt{is to add the key-value pair } (k, x) \texttt{to the dictionary.}

This is stated as a Hoare triple:

\begin{verbatim}
Theorem add_spec:
  forall M h k x, 
  app MK.add [h k x]
  PRE (h \leadsto Table M)
  POST (fun _ => Hexists M',
        h \leadsto Table M' \*
        \[ M' = add M k x \] \*
        \[ lean M \rightarrow M k = nil \rightarrow lean M' \]).
\end{verbatim}
The effect of \texttt{add \ h \ k \ x} is to add the key-value pair \((k, \ x)\) to the dictionary. This is stated as a Hoare triple:

\[
\text{Theorem add\_spec:} \\
\text{forall } M \ h \ k \ x, \\
\text{app } \text{MK}.\text{add } [h \ k \ x] \\
\text{PRE } (h \sim> \text{Table } M) \\
\text{POST } (\text{fun } _ \Rightarrow \text{Hexists } M', \\
\ h \sim> \text{Table } M' \times \\
\ [ \ M' = \text{add } M \ k \ x ] \times \\
\ [ \ \text{lean } M \rightarrow M \ k = \text{nil} \rightarrow \text{lean } M']).
\]
The data structure

First-order operations

Iteration via fold

Iteration via cascades

Conclusion
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

Writing a specification for a fold raises some questions:

▶ in what order does the consumer receive the key-value pairs?
▶ is the consumer allowed to access the table for reading? for writing?
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

A loop over the data array...
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

...a loop over a linked list...

Writing a specification for a fold raises some questions:  
▶ in what order does the consumer receive the key-value pairs?  
▶ is the consumer allowed to access the table for reading? for writing?
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

...a call to the consumer.

Writing a specification for a fold raises some questions:
▶ in what order does the consumer receive the key-value pairs?
▶ is the consumer allowed to access the table for reading? for writing?
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

Writing a specification for a fold raises some questions:

- **in what order** does the consumer receive the key-value pairs?
- is the consumer allowed to **access** the table for reading? for writing?
Really a matter of specifying which orders the consumer may observe.

The events that can be observed by a consumer are:

- the production of one element;
- the end of the sequence (this event occurs at most once, and occurs last).

An observation can be defined as a sequence of events.

A set of observations can be described by two predicates (Filliâtre and Pereira):

```
Variables permitted complete : list A -> Prop.
```
This is a higher-order specification: an implication between Hoare triples.

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 \*
    \[ complete xs \]).
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 \* \[ complete xs \]).
This is a higher-order specification: an implication between Hoare triples.

Definition Fold := \( \forall f \, c \),

\[
\text{forall x xs accu,}
\text{permitted (xs & x) -> call f x accu}
\]

\[\text{PRE ( S' c \* I xs accu)}\]

\[\text{POST (fun accu => S' c \* I (xs & x) accu)}\]

\[\rightarrow\]

\[\text{forall accu,}
\text{app fold [f c accu]}
\]

\[\text{PRE (S c \* I nil accu)}\]

\[\text{POST (fun accu => Hexists xs,}
\text{S c \* I xs accu \*}
\text{[ complete xs ])}\].
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 \* \[ complete xs \]).
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 \*
    \[ complete xs \]).

The spec is parameterized over a loop invariant I.
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 \*
    \[ complete xs \]).
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

Definition Fold :=\(\forall f \ c,\)\{
\(\forall x \ \text{xs} \ \text{accu},\)
\(\text{permited (xs & x)} \rightarrow\)
\(\text{call } f \ x \ \text{accu}\)
\(\text{PRE } (S' c \ \ast \ I \ \text{xs} \ \text{accu})\)
\(\text{POST } (\text{fun accu } \Rightarrow S' c \ \ast \ I (\text{xs} \ \& \ x) \ \text{accu})\)
\} \rightarrow
\(\forall \text{accu},\)
\(\text{app } \text{fold} [f \ c \ \text{accu}]\)
\(\text{PRE } (S c \ \ast \ I \ \text{nil} \ \text{accu})\)
\(\text{POST } (\text{fun accu } \Rightarrow \text{Hexists xs,}\)
\(S c \ \ast \ I \ \text{xs} \ \text{accu} \ \ast\)
\([\text{complete xs}])\).
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 \*
    \[ complete xs \]).

The spec is parameterized over SL predicates S and S'.
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 ]).

The producer requires S access to the collection.
Specifying fold – in general

This is a higher-order specification: an implication between Hoare triples.

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 
        \[ complete xs \]).
Specifying an iteration order – for hash tables

For hash tables, we give concrete definitions of permitted and complete:

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

where removal M kxs M’ means that from M one may remove the key-value-pair sequence kxs to obtain M’.

This specification is 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.
The specification of \texttt{fold} for hash tables is an instance of the general spec:

\begin{verbatim}
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/completed sequences: *)
  (permitted M) (complete M) I
  (* fold requires & preserves this: *)
  (fun h => h ~> TableInState M s)
  (* f receives and must preserve this: *)
  (fun h => h ~> TableInState M s).
\end{verbatim}
The specification of fold for hash tables is an instance of the general spec:

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/completes sequences: *)
    (permitted M) (complete M) I
    (* fold requires & preserves this: *)
    (fun h => h ~> TableInState M s)
    (* f receives and must preserve this: *)
    (fun h => h ~> TableInState M s).

The predicates permitted and complete for hash tables.

This spec allows read-only access to the table during iteration, and guarantees that iteration itself is a read-only operation.
The specification of `fold` for hash tables is an instance of the general spec:

```
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/completer sequences: *) 
  (permitted M) (complete M) I 
  (* fold requires & preserves this: *) 
  (fun h => h => TableInState M s) 
  (* f receives and must preserve this: *) 
  (fun h => h => TableInState M s).
```

`fold` guarantees that the table is not modified.

This spec allows read-only access to the table during iteration, and guarantees that iteration itself is a read-only operation.
Specifying fold – for hash tables

The specification of \( \text{fold} \) for hash tables is an instance of the general spec:

\[
\begin{align*}
\text{Theorem } \text{fold\_spec\_ro} : \\
\forall M \ s \ B \ I, \\
\text{Fold } MK.\text{fold} \\
\quad (* \text{Calling convention: } *) \\
\quad (\text{fun } f \ kx \ (\text{accu} : B) => \\
\quad \quad \text{app } f \ [(\text{fst } kx) \ (\text{snd } kx) \ \text{accu}]) \\
\quad (* \text{Permitted/complete sequences: } *) \\
\quad (\text{permitted } M) \ (\text{complete } M) \ I \\
\quad (* \text{fold requires & preserves this: } *) \\
\quad (\text{fun } h \Rightarrow h \leadsto \text{TableInState } M \ s) \\
\quad (* f \text{ receives and must preserve this: } *) \\
\quad (\text{fun } h \Rightarrow h \leadsto \text{TableInState } M \ s).
\end{align*}
\]

This spec allows read-only access to the table during iteration, and guarantees that iteration itself is a read-only operation.
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 => []).
```
Specifying fold – for hash tables

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

Theorem fold_spec:
for all 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 => []).
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 => []).
The data structure

First-order operations

Iteration via fold

Iteration via cascades

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

What should be the type of an iterator?
What should be the type of an iterator?

```java
public interface Iterator<E> {
    E next () throws NoSuchElementException;
    boolean hasNext();
}
```
What should be the type of an iterator?

```java
public interface Iterator<E> {
    E next () throws NoSuchElementException;
    boolean hasNext();
}
```

This interface:
- requires the iterator to be mutable;
- is more complex than strictly necessary.
A **cascade**, or *delayed list*, is perhaps the simplest possible form of iterator.

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

and 'a head =  
    | Nil  
    | Cons of 'a * 'a cascade
```
A *cascade*, or *delayed list*, is perhaps the simplest possible form of iterator.

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

and 'a head =
  | Nil
  | Cons of 'a * 'a cascade
```

Computation occurs on demand...

This definition offers an abstract, consumer-oriented view. It 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.
A cascade, or delayed list, is perhaps the simplest possible form of iterator.

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

and 'a head =
  | Nil
  | Cons of 'a * 'a cascade
```

...yielding either end-of-sequence...
A *cascade*, or *delayed list*, is perhaps the simplest possible form of iterator.

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

and 'a head =  
  | Nil  
  | Cons of 'a * 'a cascade
```

...or an element and a tail.
A cascade, or delayed list, is perhaps the simplest possible form of iterator.

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

and 'a head =  
  | Nil  
  | Cons of 'a * 'a cascade
```

This definition offers an abstract, consumer-oriented view. It 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.
Cascades

A cascade, or delayed list, is perhaps the simplest possible form of iterator.

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

and 'a head =
| Nil
| Cons of 'a * 'a cascade
```

This definition offers an abstract, consumer-oriented view. It 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 recomputed when re-demanded, or memoized.

Cascades are easy to build and use because they are "just like lists".
A cascade – for hash tables

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

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

```ocaml
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
```
...with a delay.
A cascade – for hash tables

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  
    else Nil

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

The cascade must not be used after the table is modified!
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

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

Definition c ~> Cascade xs :=
  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) ].
```
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

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

Definition $c \leadsto \text{Cascade} \; \mathit{xs} :=$

\[
\begin{align*}
\text{Hexists } S : \text{list } A \to \text{func } \to hprop, \\
S \; \mathit{xs} \; c \; \ast \\
\left[ \forall \mathit{xs} \; c, \text{duplicable } (S \; \mathit{xs} \; c) \right] \ast \\
\left[ \forall \mathit{xs} \; c, S \; \mathit{xs} \; c \implies S \; \mathit{xs} \; c \ast \left[ \text{permitted } \mathit{xs} \right] \right] \ast \\
\left[ \forall \mathit{xs} \; c, \right. \\
\left. \text{app } c \; [tt] \\
\text{INV } (S \; \mathit{xs} \; c \ast I) \\
\text{POST } (\text{fun } o \implies \\
\begin{cases} \\
\text{Nil} & \implies \left[ \text{complete } \mathit{xs} \right] \\
\text{Cons } x \; c & \implies S \; (\mathit{xs} \; \& \; x) \; c \\
\end{cases}
\end{align*}
\]

The cascade has internal invariant $S...$
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

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

Definition c ~> Cascade xs :=
  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) \].
```
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

Variable $I : hprop$.
Variables permitted complete : list A.

Definition $c \mapsto \text{Cascade } xs :=$
\[
\text{Hexists } S : \text{list } A \rightarrow \text{func } \rightarrow hprop, \\
S \times c \times \\
[ \forall \times c, \text{duplicable } (S \times c) ] \times \\
[ \forall \times c, S \times c \Rightarrow S \times c \times [ \text{permitted } \times ] ] \times \\
[ \forall \times c, \\
\text{app } c \text{[tt]} \\
\text{INV } (S \times c \times I) \\
\text{POST } (\text{fun } o \Rightarrow \\
\text{match } o \text{ with} \\
| \text{Nil} \Rightarrow \text{complete } \times \text{] } \\
| \text{Cons } x c \Rightarrow S (\times s \& x) c \\
\text{end} ]].
\]
Specifying a cascade – in general

A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this **co-inductive** specification.

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

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

The consumer may assume that the partial sequence produced so far is permitted.
Specifying a cascade – in general

A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

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

Definition c ~> Cascade xs :=
  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).

Upon termination, the consumer may deduce that the sequence is complete.
```
A cascade is a function that returns an element and a cascade.

We use an impredicative encoding of this co-inductive specification.

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

Definition c ~> Cascade xs :=
  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) \].
```

The cascade has access to an underlying data structure.
Specifying a cascade – for hash tables

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
  ).

"Concurrent modifications" are disallowed.
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
           ).

Same predicates permitted
and complete as in fold.
Theorem cascade_spec:
  forall h M s,
  app MK.cascade [h]
  INV (h \rightarrow TableInState M s)
  POST (fun c =>
    c \rightarrow Cascade
    (h \rightarrow TableInState M s)
    (permitted M) (complete M)
    nil
  ).

“Concurrent modifications” are disallowed.
The data structure

First-order operations

Iteration via fold

Iteration via cascades

Conclusion
Conclusion

I have shown arguably nice specifications expressed in vanilla Separation Logic.

- No magic wands, fractional permissions, or other black wizardry.

A few statistics:

- Under 150 loc of OCaml code.
- Dictionaries about 600 loc of Coq specs and proofs.
- Hash tables about 1500 loc of Coq specs and proofs.

Total effort about 15 man.days, but a lot of expertise still required.

Future work:

- verifying more data structures;
- making the system more accessible.