

# 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.

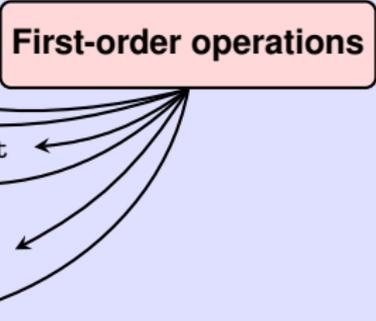
```
module Make (K : HashedType) : sig
  type key = K.t
  type 'a t
  (* Creation. *)
  val create:      int -> 'a t
  val copy:       'a t -> 'a t
  (* Insertion and removal. *)
  val add:        'a t -> key -> 'a -> unit
  val remove:     'a t -> key -> unit
  (* Lookup. *)
  val find:       'a t -> 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.

```
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
```

First-order operations



# OCaml interface

An excerpt of HashTable.mli.

```
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 opt
  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)**

# OCaml interface

An excerpt of HashTable.mli.

```
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 opt
  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  
(consumer in control)**

# OCaml implementation

An excerpt of HashTable.ml.

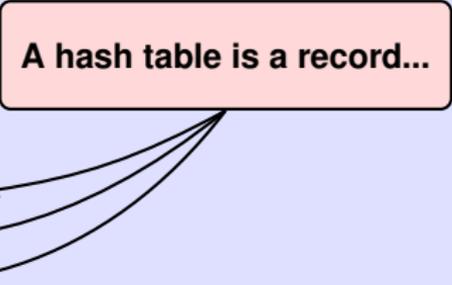
```
module Make (K : HashedType) = struct
  (* Type definitions. *)
  type key = K.t
  type 'a bucket =
    Void
  | More of key * 'a * 'a bucket
  type 'a table = {
    mutable data: 'a bucket array;
    mutable popu: int;
    init: int;
  }
  type 'a t = 'a table
  (* Operations: see following slides... *)
end
```

# 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
```

**A hash table is a record...**



# 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
```

**...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
```

**...where a bucket is a list  
of key-value pairs.**

## 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.
```

## 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.
```

**A table represents  
a finite map  
of keys to (lists of) values.**

## Separation logic invariant (in Coq)

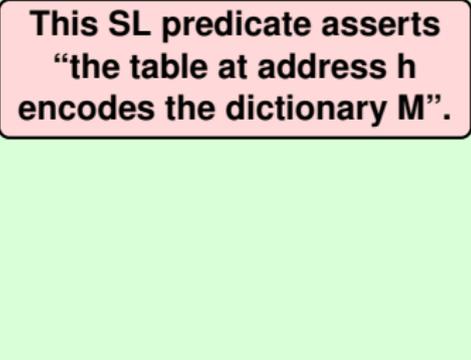
An excerpt of `HashTable_proof.v`.

```
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.
```

**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
```

```
  } \*
```

```
  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 one names *s*  
the current concrete state  
of the table.**

## 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.
```

**There must be a record at  
address h...**

## 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...**

## 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.
```

**...to an array.**

## Separation logic invariant (in Coq)

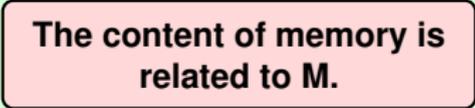
An excerpt of HashTable\_proof.v.

```
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.**



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

We use `s` to demand / guarantee that certain operations are **read-only**.

## 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.
```

**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 `add h k x` is to add the key-value pair  $(k, x)$  to the dictionary.

This is stated as a Hoare triple :

```
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' ]).
```

## Specifying a first-order operation : insertion

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

This is stated as a Hoare triple :

```
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 function call `add h k x...`**

## Specifying a first-order operation : insertion

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

This is stated as a Hoare triple :

```
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' ]).
```

**...requires a valid table...**

## Specifying a first-order operation : insertion

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

This is stated as a Hoare triple :

```
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' ]).
```

**...and produces a valid table...**

## Specifying a first-order operation : insertion

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

This is stated as a Hoare triple :

```
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' ]).
```

**...representing a dictionary  
with one more key-value pair.**

The data structure

First-order operations

**Iteration via fold**

Iteration via cascades

Conclusion

## Fold – for hash tables

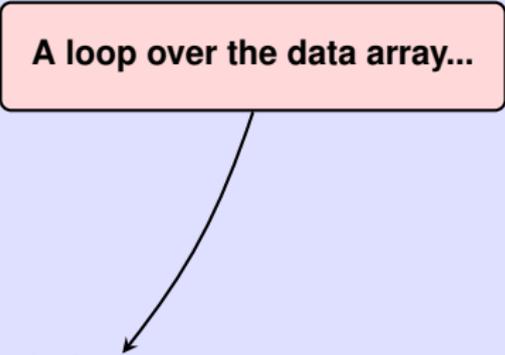
```
let rec fold_aux f b accu =
  match b with
  | Void ->
    accu
  | More(k, x, b) ->
    let accu = f k x accu in
    fold_aux f b accu

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
```

## Fold – for hash tables

```
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...**



## Fold – for hash tables

```
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...**

## Fold – for hash tables

```
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
```

**...a call to the consumer.**

```
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
```

## Fold – for hash tables

```
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 ?

## Specifying an iteration order – in general

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.
```

## 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 permitted and complete.**

## 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 ]).
```

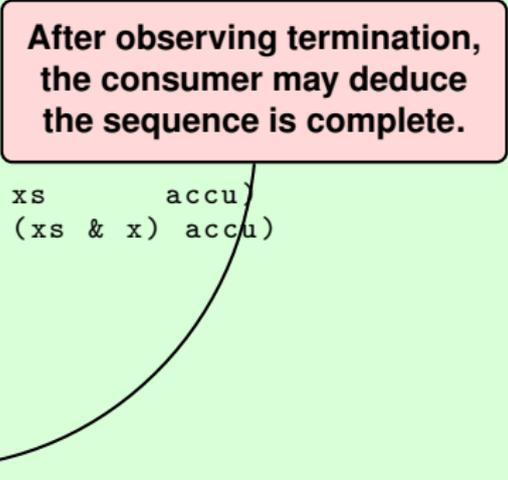
**The consumer may assume every partial sequence she observes is permitted.**

## 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 ]).
```

**After observing termination,  
the consumer may deduce  
the sequence is complete.**

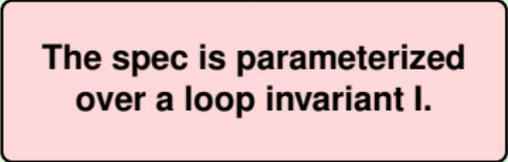


## Specifying fold – in general

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 ]).
```

**The consumer  
must preserve 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 ]).
```

**The whole iteration is then guaranteed to preserve 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 ]).
```

**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 ]).
```

**The producer gets S' access,  
which may be weaker.**

## 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.

## Specifying fold – for hash tables

The specification of `fold` for hash tables is an instance of the general spec :

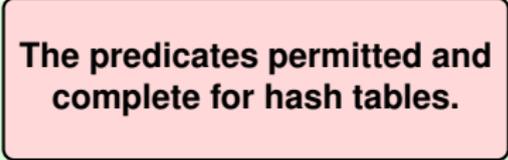
```
Theorem fold_spec_ro:
  forall M s B I,
  Fold MK.fold
    (* Calling convention: *)
    (fun f kx (accu : B) =>
      app f [(fst kx) (snd kx) accu])
    (* Permitted/complete sequences: *)
    (permitted M) (complete M) I
    (* fold requires & preserves this: *)
    (fun h => h ~> TableInState M s)
    (* f receives and must preserve this: *)
    (fun h => h ~> TableInState M s).
```

## Specifying fold – for hash tables

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/complete 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.**



## Specifying fold – for hash tables

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/complete 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 `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/complete 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 consumer cannot modify  
the table.**

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

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 kx 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:
  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 => \[]).
```

**fold does not guarantee that the table is unchanged.**

## Specifying fold – for hash tables

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 consumer gets no  
access to the table.**

The data structure

First-order operations

Iteration via fold

**Iteration via cascades**

Conclusion

# Iterators

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

# Iterators

What should be the type of an iterator ?

# Iterators

What should be the type of an iterator ?

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

## Iterators

What should be the type of an iterator ?

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



**NOT  
GREAT**

This interface :

- ▶ **requires** the iterator to be mutable ;
- ▶ is more **complex** than strictly necessary.

# Cascades

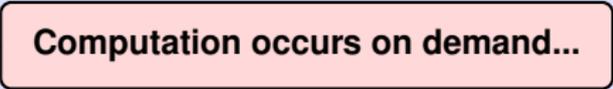
A [cascade](#), or [delayed list](#), is perhaps the simplest possible form of iterator.

```
type 'a cascade =  
  unit -> 'a head  
  
and 'a head =  
| Nil  
| Cons of 'a * 'a cascade
```

# Cascades

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...**

# Cascades

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
```

**...yielding either end-of-sequence...**

# Cascades

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
```



**...or an element and a tail.**

# Cascades

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
```

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](#).

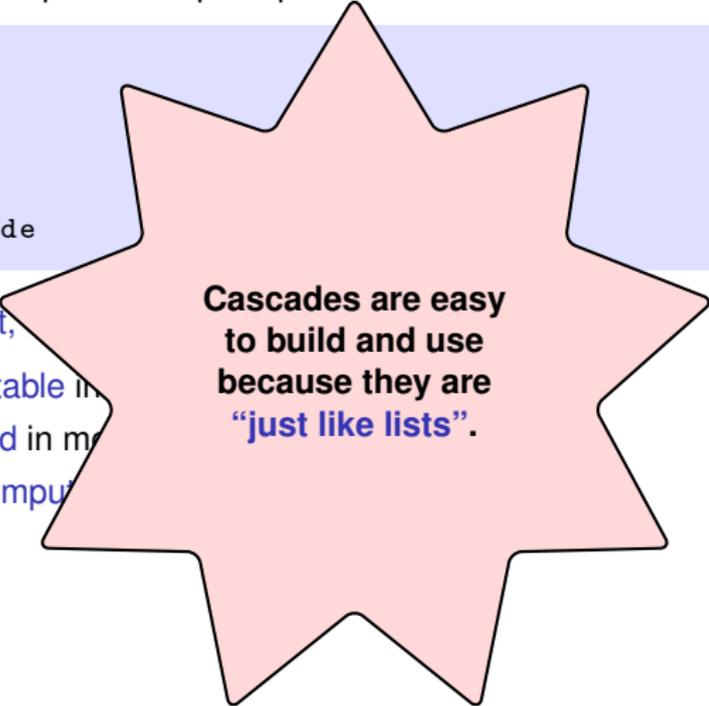
# Cascades

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

```
type 'a cascade =  
  unit -> 'a head  
  
and 'a head =  
| Nil  
| Cons of 'a * 'a cascade
```

This definition offers an **abstract**,

- ▶ whether a cascade has **mutable** in
- ▶ whether elements are **stored** in m
- ▶ whether elements are **re-compu**



**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...

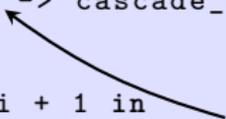
```
let rec cascade_aux data i b =
  match b with
  | More (k, x, b) ->
      Cons (
        (k, x),
        fun () -> cascade_aux data i b
      )
  | Void ->
      let i = i + 1 in
      if i < Array.length data then
        cascade_aux data i data.(i)
      else
        Nil

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

## A 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  
      cascade_aux data i  
    else  
      Nil
```



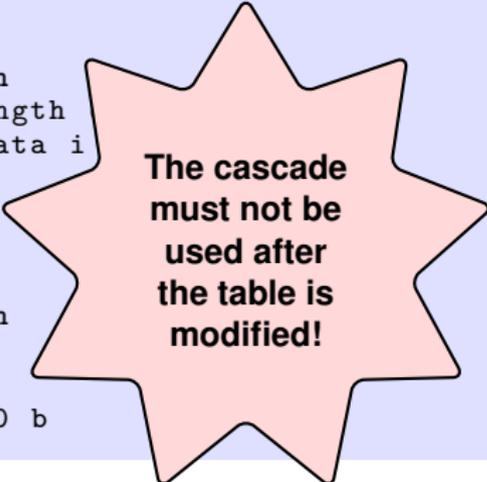
**...with a delay.**

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

## 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  
      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!**

## 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.

```
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) ].
```

## 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.

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

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 internal invariant S...**

## 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.

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

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) ].
```

**...which must be duplicable.  
A cascade is persistent.**

## 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.

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

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) ].
```

**Calling *c* requires validity and produces another valid cascade.**

## 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.

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

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 spec is parameterized over permitted and complete.**

## 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.

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

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 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.

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

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.**

## 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.

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

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.**

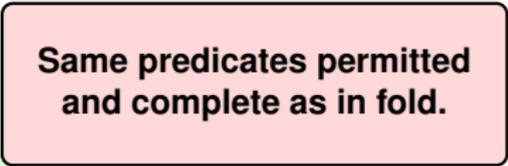
## Specifying a cascade – for hash tables

```
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
  ).
```

## 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
  ).
```

**Same predicates permitted  
and complete as in fold.**



## 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  
  ).
```

**The cascade can be used  
only as long as the table  
remains in state s.**

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