A Modular Way to Reason About Iteration

Jean-Christophe Filliâtre Mário Pereira

LRI, Univ. Paris-Sud, CNRS, Inria Saclay

INRIA Paris - Séminaire Gallium
Mars 7, 2016

iteration: hello old friend!

many different forms of iteration

- cursors
- higher-order functions (*fold*, *iter*)
- streams
- etc.

our goal: specify and verify the enumeration of a "collection" of elements.

in this work : cursors and higher-order iterators

a specification that takes into account

- either finite or infinite iterations
- either deterministic or non-deterministic iterations
- traversal of a possibly mutable collection
- not necessarily the traversal of a data structure

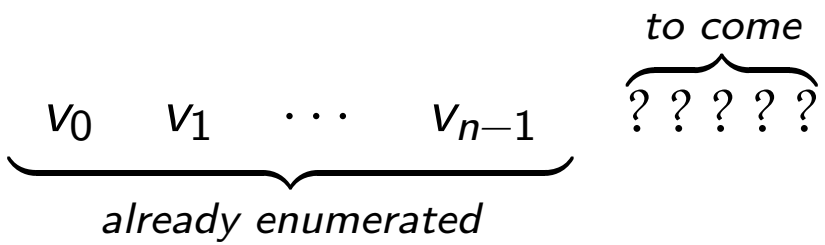
verify programs

- implementing an iteration (producers)
- using an iteration (consumers)

abstraction: consumer verified without any knowledge about the implementation of the producer

iteration formalized with two predicates

- *completed* indicates whether we are done with the enumeration
- *enumerated* characterizes the elements already produced so far



traversal of an array

$$\begin{aligned} \mathit{enumerated}\ v\ a &\stackrel{\text{def}}{=} \forall\, i.\, 0 \leq i < |v| \implies v[i] = a[i] \\ \mathit{completed}\ v\ a &\stackrel{\text{def}}{=} |v| = |a| \end{aligned}$$

non-deterministic traversal of a set

$$\begin{aligned} \mathit{enumerated}\ v\ s &\stackrel{\text{def}}{=} \mathit{distinct}\ v \wedge \forall\, x.\ x \in v \Longrightarrow x \in s \\ \mathit{completed}\ v\ s &\stackrel{\text{def}}{=} |v| = \mathit{card}(s) \end{aligned}$$

deterministic traversal

$$\begin{aligned} \mathit{enumerated}\ v\ s &\stackrel{\text{def}}{=} \mathit{prefix}\ v\ (\mathit{to\_seq}\ s) \\ \mathit{completed}\ v\ s &\stackrel{\text{def}}{=} |v| = \mathit{card}(s) \end{aligned}$$

example: iterated function

$$\begin{aligned} \mathit{enumerated}\ v\ (x_0, f) &\stackrel{\text{def}}{=} \forall\ i.\ 0 \leq i < |v| \implies v[i] = f^i(x_0) \\ \mathit{completed}\ v\ (x_0, f) &\stackrel{\text{def}}{=} \mathit{false} \end{aligned}$$

$$x_0,\ f(x_0),\ f(f(x_0)),\ f(f(f(x_0))),\ \ldots$$

elements of v are characters

character `EOF` marks the end of the channel

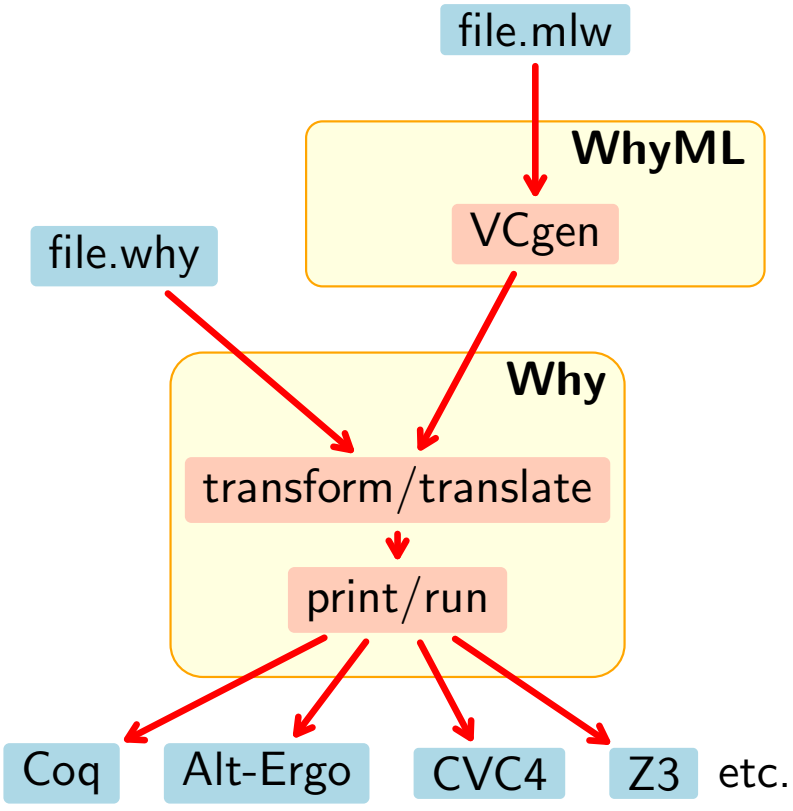
$$\begin{aligned} \mathit{enumerated}\ v\ c &\stackrel{\text{def}}{=} \ldots \wedge \forall i.\ 0 \leq i < |v| - 1 \implies v[i] \neq \texttt{EOF} \\ \mathit{completed}\ v\ c &\stackrel{\text{def}}{=} |v| > 0 \wedge v[|v| - 1] = \texttt{EOF} \end{aligned}$$

example: symbol generator

to generate, at each step, a fresh symbol

$$\begin{aligned} \mathit{enumerated}\ v\ () &\stackrel{\text{def}}{=} \mathit{distinct}\ v \\ \mathit{completed}\ v\ () &\stackrel{\text{def}}{=} \mathit{false} \end{aligned}$$

experimental validation with Why3

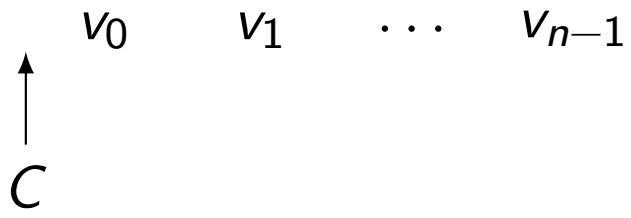


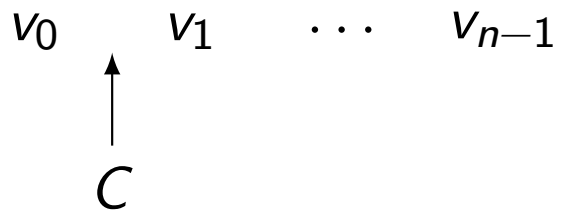
http://why3.lri.fr/

cursors in a picture

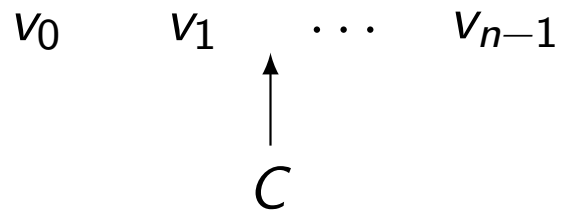
$$v_0 \qquad v_1 \qquad \cdots \qquad v_{n-1}$$

cursors in a picture



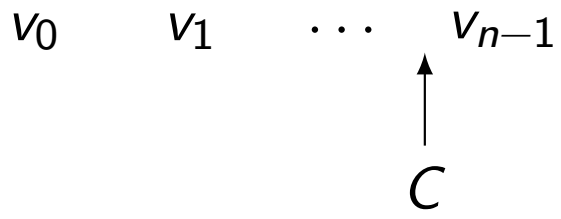


$$next\ C \quad = \quad v_0$$



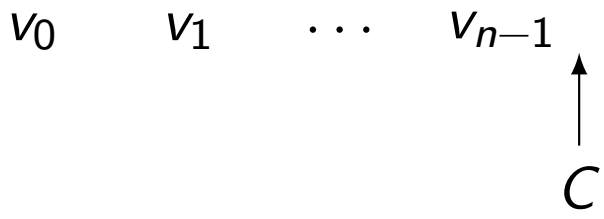
$$next\ C = v_0$$
$$next\ C = v_1$$

cursors in a picture



$$
\begin{aligned}
next\ C &= v_0 \\
next\ C &= v_1 \\
&\cdots \\
next\ C &= v_{n-2}
\end{aligned}
$$

cursors in a picture



$$
\begin{aligned}
\mathit{next}\ C &= v_0 \\
\mathit{next}\ C &= v_1 \\
&\cdots \\
\mathit{next}\ C &= v_{n-2} \\
\mathit{next}\ C &= v_{n-1}
\end{aligned}
$$

```
c ← create_cursor (...)
while has_next(c) do
  x ← next(c)
  ...
```

cursor specification

```
type elt
type collection

type cursor model {
   mutable visited   : seq elt;
           collection: collection; }
```

cursor specification

```
type elt
type collection

type cursor model {
   mutable visited   : seq elt;
           collection: collection; }

predicate enumerated cursor
predicate completed  cursor
```

cursor specification

```
type elt
type collection

type cursor model {
   mutable visited   : seq elt;
           collection: collection; }

predicate enumerated cursor
predicate completed  cursor

val has_next (c: cursor) : bool
  requires { enumerated c }
  ensures  { result ↔ not (completed c) }
```

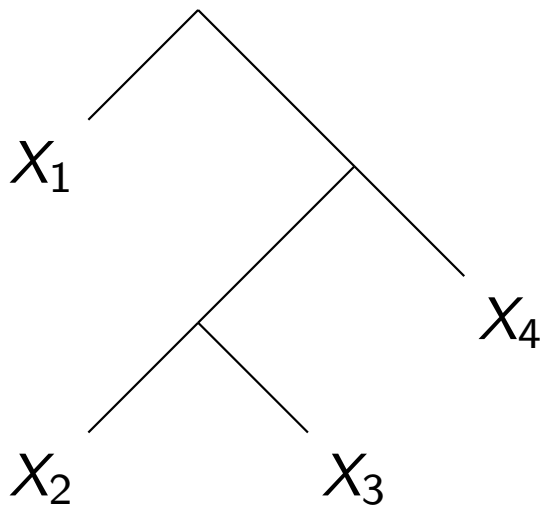
```
type elt
type collection

type cursor model {
   mutable visited   : seq elt;
           collection: collection; }

predicate enumerated cursor
predicate completed  cursor

val has_next (c: cursor) : bool
  requires { enumerated c }
  ensures  { result ↔ not (completed c) }

val next (c: cursor) : elt
  requires { enumerated c }
  requires { not (completed c) }
  writes   { c }
  ensures  { enumerated c }
  ensures  { c.visited = snoc (old c.visited) result }
```

verification example

1. specify the in-order traversal of a binary tree
2. realize it using a cursor
3. application to the same fringe problem

1. specify the in-order traversal

```
type tree = Empty | Node tree elt tree

function elements (t : tree) : seq elt = match t with
  | Empty       → empty
  | Node l x r → elements l ++ cons x (elements r)
  end
```

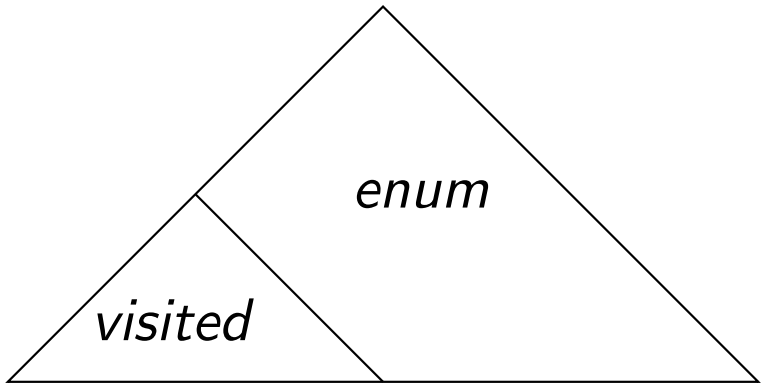


1. specify the in-order traversal

```
type tree = Empty | Node tree elt tree

function elements (t : tree) : seq elt = match t with
  | Empty      → empty
  | Node l x r → elements l ++ cons x (elements r)
  end


predicate enumerated (c: cursor) =
  prefix c.visited (elements c.collection)

predicate completed (c: cursor) =
  length c.visited = length (elements c.collection)
```

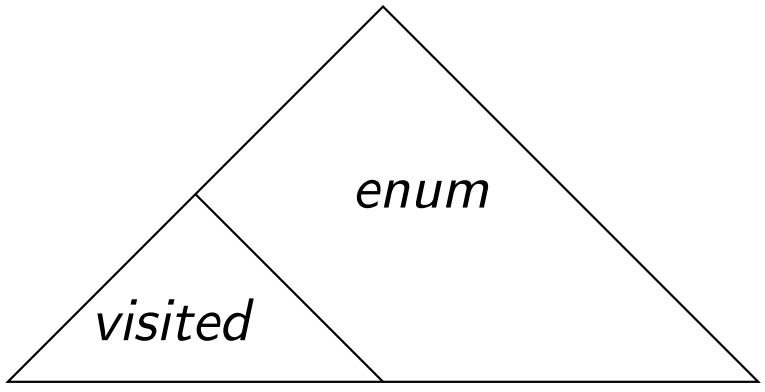
2. realize the in-order traversal with a cursor

```
type enum = Done | Next elt tree enum

type cursor = {
  ghost mutable   visited: seq elt;
        mutable      enum: enum;
                collection: tree; }
```



2. realize the in-order traversal with a cursor

```
type enum = Done | Next elt tree enum

type cursor = {
  ghost mutable   visited: seq elt;
        mutable      enum: enum;
                collection: tree; }
```

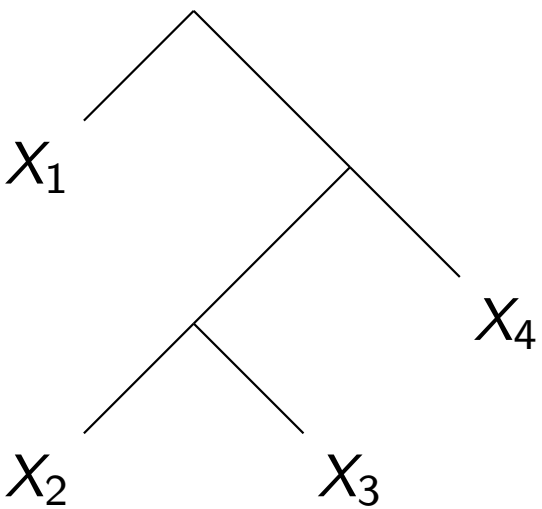


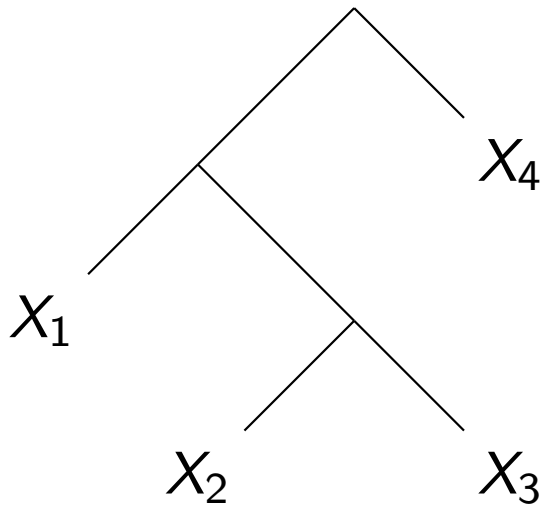

```
  function enum_elements (e: enum) : seq elt =
    match e with
    | Done         → empty
    | Next x r e → cons x (elements r ++ enum_elements e)
    end

  predicate enumerated (c: cursor) =
    elements c.collection = c.visited ++ enum_elements c.enum
```

3. application to the *same fringe* problem

decide whether two trees have the same in-order elements

```
let same_fringe (t1 t2: tree) : bool
  ensures { result ↔ elements t1 = elements t2 }
```

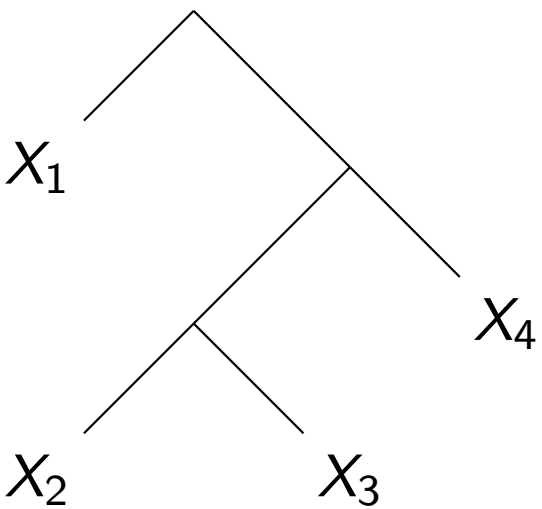


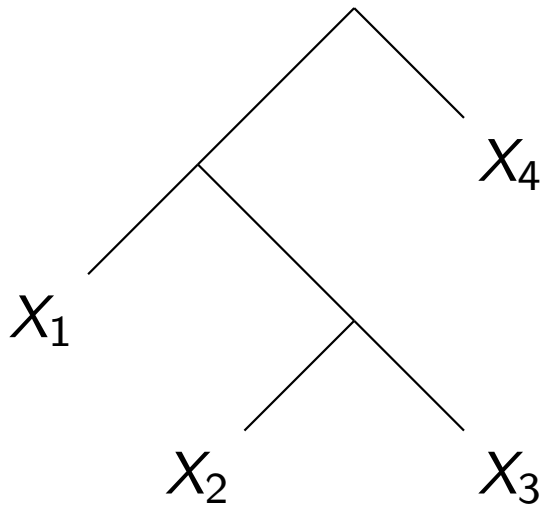


3. application to the *same fringe* problem

decide whether two trees have the same in-order elements

```
let same_fringe (t1 t2: tree) : bool
  ensures { result ↔ elements t1 = elements t2 }
= eq_cursors (create_cursor t1) (create_cursor t2)
```





3. application to the *same fringe* problem

decide whether two trees have the same in-order elements

```
let rec eq_cursors (c1 c2: cursor) : bool
  ...
= match has_next c1, has_next c2 with
  | False, False → True
  | True, True   → next c1 = next c2 && eq_cursors c1 c2
  | _            → False
  end
```

demo same fringe

higher-order iterators

```
iter: (elt → unit) → collection → unit
```

traditional approach: higher-order (program) logic

alternative: enumerated / completed predicates

how: translate higher-order iterators into first-order Why3 code

verifying HO iterator - specification

```
type list = Nil | Cons elt list

let rec iter (f: elt → unit) (l: list)


= match l with
  | Nil       → ()
  | Cons x l → f x; iter f l
  end
```

verifying HO iterator - specification

```
type list = Nil | Cons elt list

let rec iter (f: elt → unit) (l: list)
  with { enumerated (visited, t) = ...
         completed  (visited, t) = ... }
= match l with
  | Nil       → ()
  | Cons x l → f x; iter f l
  end
```

verifying HO iterator - translation

```
val visited: ref (seq elt)

let iter_correct (l0: list)
  requires { !visited == empty }
  requires { enumerated (empty, l0) }
  ensures  { enumerated (!visited, l0) }
  ensures  { completed (!visited, l0) }
= let f x = visited := snoc !visited x in
  let rec iter0 (l: list) : unit =
    match l with
    | Nil       → ()
    | Cons x l → f x; iter0 l
    end in
  iter0 l0
```

demo *iter*

using HO iterator - specification

```
let uniq (l: list)
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  iter (fun x →

          if not (H.mem x h) then
            begin H.add x h; r := Cons x !r end)
        l;
  !r
```

using HO iterator - specification

```
let uniq (l: list)
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  iter (fun x →
        (* user
            loop
             invariant *)
         if not (H.mem x h) then
           begin H.add x h; r := Cons x !r end)
       l;
  !r
```

using HO iterator - specification

```
let uniq (l: list)
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  iter (fun (ghost v) x →
        (* user
            loop
             invariant *)
         if not (H.mem x h) then
           begin H.add x h; r := Cons x !r end)
       l;
  !r
```

using HO iterator - specification

```
let uniq (l: list)
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  iter (fun (ghost v) x →
         invariant { distinct !r }
         invariant { forall x. mem x v ↔ mem x !r }
         invariant { forall x. H.contains h x ↔ mem x !r }
         if not (H.mem x h) then
           begin H.add x h; r := Cons x !r end)
       l;
  !r
```

using HO iterator - translation

replace *iter* with a cursor:

```
let uniq_correct (l: list elt) : list elt
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  let _c = create_cursor l in

  !r
```

using HO iterator - translation

replace *iter* with a cursor:

```
let uniq_correct (l: list elt) : list elt
  ensures { distinct result }
  ensures { forall x. mem x result ↔ mem x l }
= let h = H.create () in
  let r = ref Nil in
  let _c = create_cursor l in
  while has_next _c do
    invariant { enumerated _c }
    invariant { let v = _c.visited in
                    ...user loop invariant... }
    let x = next _c in
     if not (H.mem x h) then
       begin H.add x h; r := Cons x !r end
  done;
  !r
```

demo *uniq*

a modular way to specify iteration

verification of programs realizing / using iterators with Why3

Why3 source code available at

`http://www.lri.fr/~mpereira/iteration/`

soundness proof of higher-order iterators proof method

application to other higher-order iterators

```
fold: (elt → 'a → 'a) → collection → 'a → 'a
```

verify realistic code making heavy use of higher-order iterators (e.g. OCamlGraph's code)

Itérer avec confiance
J.-C. Filliâtre and M. Pereira
JFLA'16
`https://hal.inria.fr/hal-01240891`

A Modular Way to Reason About Iteration
J.-C. Filliâtre and M. Pereira
submitted to NFM'16
`https://hal.inria.fr/hal-01281759`

questions?