Sisyphus: Mostly-Automated Proof Repair for Verified Libraries

Kiran Gopinathan, Mayank Keoliya, Ilya Sergey

National University of Singapore

Pictured: OCaml programmer fixing a broken Coq proof

Let’s write a program!

Let’s write a **verified** program!

Let’s write a **verified** program!

Q: Convert a sequence to an array.

Q: Convert a sequence to an array.

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

OCaml's `'a Seq.t` datatype

```
type 'a t    = unit -> 'a node
and  'a node = Nil
             | Cons of 'a * (unit -> 'a node)
```

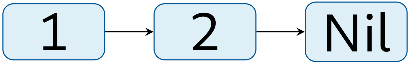
OCaml’s `'a Seq.t` datatype

```
type 'a t    = unit -> 'a node
and  'a node = Nil
             | Cons of 'a * (unit -> 'a node)
```

A *thunked* tail

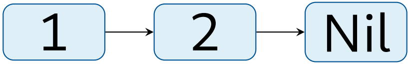
OCaml’s `'a Seq.t` datatype

```
type 'a t    = unit -> 'a node
and  'a node = Nil
             | Cons of 'a * (unit -> 'a node)
```



OCaml’s `'a Seq.t` datatype

```
type 'a t    = unit -> 'a node
and  'a node = Nil
             | Cons of 'a * (unit -> 'a node)
```



```
fun () -> Cons (1, fun () -> Cons (2, fun () -> Nil))
```

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

Q: Convert a sequence to an array.

finite and
effect-free

$$\forall s\ \ell,\ \{s \mapsto \text{Seq}\ \ell\}$$

$$(\text{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \text{Array}\ \ell\}$$

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

“a” **points-to** an array

Q: Convert a sequence to an array.

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

Let’s write
some
code/proofs!

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}$$

$$(\mathsf{to\_array}\ s)$$

$$\exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}\quad (\mathsf{to\_array}\ s)\quad \exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}\quad (\mathsf{to\_array}\ s)\quad \exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}\quad (\mathsf{to\_array}\ s)\quad \exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

$$\forall s\ \ell,\ \{s \mapsto \mathsf{Seq}\ \ell\}\ \ (\mathsf{to\_array}\ s)\ \ \exists a,\ \{a \mapsto \mathsf{Array}\ \ell\}$$

```
let to_array s =                    xcf.
```

$\{s \mapsto \mathsf{Seq}\ \ell\}$

```
let to_array s =                  xcf.
```

$\{s \mapsto \mathsf{Seq}\ \ell\}$

```
let to_array s =
  match s () with
```

xcf.

$\{s \mapsto \mathsf{Seq}\ \ell\}$

```
let to_array s =
  match s () with
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

$$\{s \mapsto \mathsf{Seq}\ [\,]\}$$

```
let to_array s =
  match s () with
  | Nil ->
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

$$\{s \mapsto \mathsf{Seq}\ [\,]\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

$$\{s \mapsto \mathsf{Seq}\ []\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

  – xvalemptyarr.

$$\exists a, \{a \mapsto \texttt{Array}\ []\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

— xvalemptyarr.

$$\{s \mapsto \mathsf{Seq}\,(h\ ::\ tl)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

$\quad$ — xvalemptyarr.

$$\{s \mapsto \mathsf{Seq}\ (h\ ::\ tl)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
```

```
xcf.
xapp; case ℓ as [| h tl]
  — xvalemptyarr.
  —
```

$$\{s \mapsto \mathrm{Seq}\,(h :: tl)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
```

$$\{sz = \mathsf{length}\ \ell;\ s \mapsto \mathsf{Seq}\ (h\ ::\ tl)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

— xvalemptyarr.

—

xapp.

$$\{sz = \mathsf{length}\ \ell;\ s \mapsto \mathsf{Seq}\ (h\ ::\ tl)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
```

```
xcf.
xapp; case ℓ as [| h tl]
  — xvalemptyarr.
  —
   xapp.
```

$$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h\ ::\ tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
```

$$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h :: tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

       a.(i) <- vl

    ) s;
```

```
xcf.
xapp; case ℓ as [| h tl]
  — xvalemptyarr.
  —
   xapp.
   xalloc.
```

$$\{sz = \mathtt{length}\ \ell;\ s \mapsto \mathtt{Seq}\ (h :: tl) * a \mapsto \mathtt{Array}\ (\mathtt{make}\ sz\ h)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

       a.(i) <- vl

    ) s;
```

```
xcf.
xapp; case ℓ as [| h tl]
   - xvalemptyarr.
   -
    xapp.
    xalloc.
    xapp (iteri_spec (λt →
     a ↦ Array (
      t ++ drop (length t)
        (make (length ℓ) h))
    ).
```

$$\{sz = \mathtt{length}\ \ell;\ s \mapsto \mathtt{Seq}\ (h\ ::\ tl) * a \mapsto \mathtt{Array}\ (\mathtt{make}\ sz\ h)\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

        a.(i) <- vl

    ) s;
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec (λt →
    a ↦ Array (
    t ++ drop (length t)
      (make (length ℓ) h))
   ).
```

$$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h :: tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$$

$$t \mathbin{+\!\!+} drop\ (length\ t)\ (make\ (length\ \ell)\ h)$$

```
let to_array s =
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

       a.(i) <- vl

    ) s;
```

```
xcf.

 —
 xapp.
 xalloc.
 xapp (iteri_spec (λt →
  a ↦ Array (
  t ++ drop (length t)
    (make (length ℓ) h))
 ).
```

$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h :: tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$

$$t ++ drop\ (length\ t)\ (make\ (length\ \ell)\ h)$$

```
let to_array s =
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

        a.(i) <- vl

    ) s;
```

xcf.

—

xapp.

xalloc.

xapp (iteri_spec ($\lambda t \to$

$a \mapsto$ Array (

$t ++ drop\ (length\ t)$

$(make\ (length\ \ell)\ h))$

).

$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h :: tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$

```
let to_array s =
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
      a.(
    ) s;
```

xcf.

—

xapp.

xalloc.

xapp (iteri_spec ($\lambda t \to$

$a \mapsto$ Array (

$t \mathbin{++} drop\ (length\ t)$

$(make\ (length\ \ell)\ h))$

).

$$t \mathbin{++} drop\ (length\ t)\ (make\ (length\ \ell)\ h)$$

$\{sz = \mathtt{length}\ \ell;\ s \mapsto \mathtt{Seq}\ (h :: tl) * a \mapsto \mathtt{Array}\ (\mathtt{make}\ sz\ h)\}$

$$t +\!\!+ drop\ (length\ t)\ (make\ (length\ \ell)\ h)$$

```
let to_array s =
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
      a.(
    ) s;
```

xcf.

—

xapp.

xalloc.

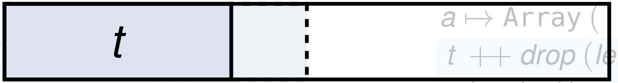
xapp (iteri_spec ($\lambda t \rightarrow$

$a \mapsto$ Array (

$t +\!\!+ drop\ (length\ t)$

$(make\ (length\ \ell)\ h))$

).



$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h\ ::\ tl) * a \mapsto \texttt{Array}\ (\texttt{make}\ sz\ h)\}$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

          a.(i) <- vl

    ) s;
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec (λt →
    a ↦ Array (
    t ++ drop (length t)
      (make (length ℓ) h))
   ).
```

$$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h :: tl) * a \mapsto \texttt{Array}\ \ell\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

       a.(i) <- vl

    ) s;
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec (λt →
    a ↦ Array (
     t ++ drop (length t)
       (make (length ℓ) h))
   ).
```

$$\{sz = \texttt{length}\ \ell;\ s \mapsto \texttt{Seq}\ (h\ ::\ tl) * a \mapsto \texttt{Array}\ \ell\}$$

```
let to_array s =
  match s () with
   | Nil -> [| |]
   | Cons (h, _) ->
     let sz = length s in
     let a = make sz h in
     iteri (fun i vl ->

        a.(i) <- vl

     ) s;
     a
```

```
xcf.
xapp; case ℓ as [| h tl]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec (λt →
    a ↦ Array (
     t ++ drop (length t)
       (make (length ℓ) h))
   ).
```

$$\exists a, \{a \mapsto \texttt{Array}\ \ell\}$$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->

        a.(i) <- vl

    ) s;
    a
```

```
xcf.
xapp; case ℓ as [| h tl]
  — xvalemptyarr.
  —
   xapp.
   xalloc.
   xapp (iteri_spec (λt →
    a ↦ Array (
     t ++ drop (length t)
       (make (length ℓ) h))
   ).
  xval.
```

$\exists a, \{a \mapsto \mathtt{Array}\ \ell\}$

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h i
    iteri (fun i vl ->

       a.(i) <- vl

    ) s;
    a
```

xcf.

xapp; case ℓ as $[|\ h\ tl]$

– xvalemptyarr.

–

xapp.

xalloc.

xapp (iteri_spec $(\lambda t \to$

$a \mapsto$ Array (

$t \mathbin{++} drop\ (length\ t)$

$(make\ (length\ \ell)\ h))$

).

xval.

Qed.

Let’s write a **verified** program!

Let’s write a **verified** program!

Conclusion:

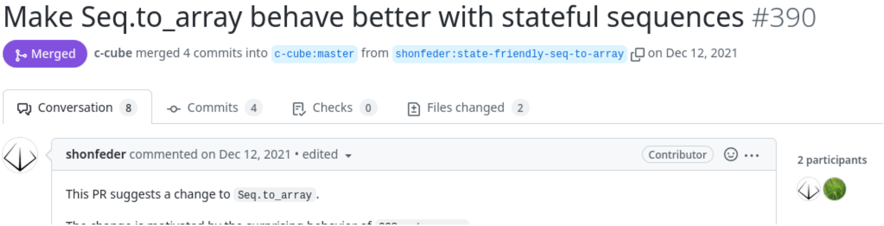
Writing verified code is hard

Writing verified code is hard

Writing verified code is hard…

A problem arises..

A problem arises..



Make Seq.to_array behave better with stateful sequences #390

c-cube merged 4 commits into `c-cube:master` from `shonfeder:state-friendly-seq-to-array` on Dec 12, 2021

shonfeder commented on Dec 12, 2021 • edited

This PR suggests a change to `Seq.to_array`.

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Completely different implementation...

Old

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
    iteri (fun i vl ->
       a.(i) <- vl
    ) s;
    a
```

New

```
let to_array s =
  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = len - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

Completely different implementation…

…proof must be redone.

Writing verified code is hard…

Maintaining

~~Writing~~ verified code is hard…

Maintaining ~~Writing~~ verified code is hard…

…can we make it easier?

Contributions

1. **Sisyphus**: tool to *repair* proofs over changes.

2. **PDT**: technique to *efficiently* test invariants.

3. Evaluation on 10 real OCaml programs and their changes

New Program:

```
let to_array s =

  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = sz - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

New Program:

```
let to_array s =

  let sz, ls = fold_left

  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = sz - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

How to generate a proof script?

New Program:

```
let to_array s =

  let sz, ls = fold_left
```

How to generate a proof script?

```
  match ls with
  | [] -> [| |]
  | init :: rest ->
```

Observation: proofs are syntax-directed

```
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

New Program:

```
let to_array s =

  let sz, ls = fold_left

    (0, []) t in
  match ls with
  | [] -> [| |]
  | init :: rest ->

    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

How to generate a proof script?

Observation: proofs are syntax-directed

```
let to_array s =

  let sz, ls = fold_left
    (fun (i, acc) x ->
      (i+1, x::acc))
    (0, []) l in
  match ls with
  | [] -> [| |]
  | init :: rest ->
    let a = make sz init in
    let idx = sz - 2 in
    List.fold_left
     (fun i vl ->
       a.(i)<-vl; i-1)
     idx rest;
    a
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

xcf.

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

xcf.

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

xcf.

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).

    xvals.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).

    xvals.
```

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).

    xvals.
```

How to fill in the holes?

```
let to_array s =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) s in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz -
       List.fold_left
         (fun i x ->
           a.(i)<-x; i-1)
         idx rest in
       a
```

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
                                rest
    (fun acc ls =>
     (??))).

    xvals.
```

Key challenges for proof repair

① *Generating* candidate invariants

② *Testing* generated invariants

Key challenges for proof repair

1. *Generating* candidate invariants

2. *Testing* generated invariants

Generating candidate invariants

How to fill in holes?

Generating candidate invariants

How to fill in holes?

Use the old program and proofs...

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

Programs are different…

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

Programs are different…

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
      a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

…but have similarities.

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil        -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil           -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil            -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
  match l () with
  | Nil        -> [| |]
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    iteri
      (fun i x -> a.(i) <- x)
      l;
    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
          (i + 1, x :: acc))
      (0, []) l in
  match ls with
  | [] -> [| |]
  | init::rest ->
    let a = make sz init in
    let idx = sz - 2 in
    let _ =
      List.fold_left
        (fun i x -> a.(i) <- x; i - 1)
        idx rest in
    a
```

Generating candidate invariants

```
let to_array l =
  match l () with
  | Nil          -> [| |]
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    iteri
      (fun i x -> a.(i) <- x)
      l;
    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
    | [] -> [| |]
    | init::rest ->
      let a = make sz init in
      let idx = sz - 2 in
      let _ =
        List.fold_left
          (fun i x -> a.(i) <- x; i - 1)
          idx rest in
      a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil         -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
        a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
        a
```

Generating candidate invariants

```
let to_array l =
   match l () with
   | Nil        -> [| |]
   | Cons (x, _) ->
       let len = length' l in
       let a = make len x in
       iteri
         (fun i x -> a.(i) <- x)
         l;
       a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
       a
```

Generating candidate invariants

Q: How to discover these similarities automatically?

```
let to_array l =
  match l () with
  | Nil         -> [| |]
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    iteri
      (fun i x -> a.(i) <- x)
      l;
    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
  match ls with
    | [] -> [| |]
    | init::rest ->
      let a = make sz init in
      let idx = sz - 2 in
      let _ =
        List.fold_left
          (fun i x -> a.(i) <- x; i - 1)
          idx rest in
      a
```

Generating candidate invariants

Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

```
let to_array l =
  match l () with
  | Nil -> [| |]
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    iteri
      (fun i x -> a.(i) <- x)
      l;
    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
      (i + 1, x :: acc))
      (0, []) l in
  match ls with
    | [] -> [| |]
    | init::rest ->
      let a = make sz init in
      let idx = sz - 2 in
      let _ =
        List.fold_left
          (fun i x -> a.(i) <- x; i - 1)
          idx rest in
      a
```

Generating candidate invariants

Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

Identify “similar” program points through traces.

```
let to_array l =

  match l () with

  | Cons (x, _) ->

    let len = length' l in

    iteri

      (fun i x -> a.(i) <- x)

      l;

    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
      (0, []) l in
    match ls with
     | [] -> [| |]
     | init::rest ->
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x; i - 1)
           idx rest in
       a
```

Generating candidate invariants

Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

Identify “similar” program points through traces.

Use invariants from old proof to synthesise invariants for new one.

```
let to_array l =
  match l () with
  | Nil -> [| |]
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    iteri
      (fun i x -> a.(i) <- x)
      l;
    a
```

```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
        (i + 1, x :: acc))
      (0, []) l in
  match ls with
  | [] -> [| |]
  | init::rest ->
    let a = make sz init in
    let idx = sz - 2 in
    let _ =
      List.fold_left
        (fun i x -> a.(i) <- x; i - 1)
        idx rest in
    a
```

Key challenges for proof repair

① *Generating* candidate invariants

② *Testing* generated invariants

Key challenges for proof repair

① *Generating* candidate invariants

② *Testing* generated invariants

Testing Candidate Invariants

Testing Candidate Invariants

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).

    xvals.
```

Testing Candidate Invariants

```
xcf.

xapp (...).

case l as [| init rest].
  - xmatch 0. xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
    (fun acc ls =>
     (??))).

    xvals.
```

Testing Candidate Invariants

`fold_left_spec`

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\ \{I\ res\ (t\ +\!+\ [v])\}\end{array}\right) \to$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathsf{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\ \{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

Loop Invariant

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\ \{I\ res\ (t\ +\!+\ [v])\}\end{array}\right) \rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathrm{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\ \{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c} \{I\ acc'\ t\} \\ (f\ acc'\ v) \\ \exists res, \{I\ res\ (t\ \mathbin{++}\ [v])\} \end{array}\right) \rightarrow$$

$$\begin{array}{c} \{I\ acc\ []\} \\ (\mathsf{List.fold\_left}\ f\ acc\ \ell) \\ \exists res, \{I\ res\ \ell\} \end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c} \{ I\ acc'\ t \} \\ (f\ acc'\ v) \\ \exists res,\ \{ I\ res\ (t\ +\!\!+\ [v]) \} \end{array} \right) \rightarrow$$

$$\begin{array}{c} \{ I\ acc\ [\,] \} \\ (\,\mathsf{List}\,.\,\mathsf{fold\_left}\ f\ acc\ \ell\,) \\ \exists res,\ \{ I\ res\ \ell \} \end{array}$$

Preservation of invariant

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res, \{I\ res\ (t\ {+}{+}\ [v])\}\end{array}\right) \to$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathsf{List.fold\_left}\ f\ acc\ \ell)\\ \exists res, \{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\ \{I\ res\ (t\ +\!\!+\ [v])\}\end{array}\right) \rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\text{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\ \{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

If I holds on t

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\{I\ res\ (t\ \mathbin{++}\ [v])\}\end{array}\right)\rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathrm{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{matrix} \{I\ acc'\ t\} \\ (f\ acc'\ v) \\ \exists res, \{I\ res\ (t\ ++\ [v])\} \end{matrix}\right) \rightarrow$$

$$\begin{matrix} \{I\ acc\ []\} \\ (\mathrm{List.fold\_left}\ f\ acc\ \ell) \\ \exists res, \{I\ res\ \ell\} \end{matrix}$$

If I holds on t

after calling f...

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{r} \{I\ acc'\ t\} \\ (f\ acc'\ v) \\ \exists res, \{I\ res\ (t \mathbin{++} [v])\} \end{array}\right) \rightarrow$$

$$\begin{array}{c} \{I\ acc\ []\} \\ (\mathrm{List.fold\_left}\ f\ acc\ \ell) \\ \exists res, \{I\ res\ \ell\} \end{array}$$

If I holds on t

after calling f...

I holds on $t \mathbin{++} [v]$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\(f\ acc'\ v)\\\exists res,\{I\ res\ (t\ +\!\!+\ [v])\}\end{array}\right)\rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\(\mathsf{List.fold\_left}\ f\ acc\ \ell)\\\exists res,\{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\{I\ res\ (t\ \mathbin{++}\ [v])\}\end{array}\right) \rightarrow$$

Initial condition

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathrm{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\{I\ res\ \ell\}\end{array}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res, \{I\ res\ (t \mathbin{++} [v])\}\end{array}\right) \rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathrm{List\,.\,fold\_left}\ f\ acc\ \ell)\\ \exists res, \{I\ res\ \ell\}\end{array}$$

I holds on entire ℓ

Testing Candidate Invariants

`fold_left_spec:`

How to *test* a candidate for I?

$$\{I\ acc\ []\}$$
$$(\text{List.fold\_left}\ f\ acc\ \ell)$$
$$\exists res, \{I\ res\ \ell\}$$

Testing Candidate Invariants

`fold_left_spec:`

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\ \begin{array}{c}\{I\ acc'\ t\}\\ (f\ acc'\ v)\\ \exists res,\ \{I\ res\ (t\ +\!\!+\ [v])\}\end{array}\right) \rightarrow$$

$$\begin{array}{c}\{I\ acc\ []\}\\ (\mathrm{List.fold\_left}\ f\ acc\ \ell)\\ \exists res,\ \{I\ res\ \ell\}\end{array}$$

Depends on
logical variables

Testing Candidate Invariants

`fold_left_spec:`

$\Big(\qquad \{ I\ acc'\ t \} \qquad \Big)$

How to *test* a candidate for *I* ?

$$\{ I\ acc\ [\,] \}$$
$$(\,\mathsf{List}\,.\,\mathsf{fold\_left}\ f\ acc\ \ell)$$
$$\exists res, \{ I\ res\ \ell \}$$

Testing Candidate Invariants

`fold_left_spec:`

How to *test* a candidate for I?

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\quad (f\ acc'\ v)\quad \exists res, \{I\ res\ (t\ +\!\!+\ [v])\}\right) \to$$

$$\{I\ acc\ []\}$$
$$(\mathsf{List.fold\_left}\ f\ acc\ \ell)$$
$$\exists res, \{I\ res\ \ell\}$$

Testing Candidate Invariants

`fold_left_spec:`

How to *test* a candidate for I?

$$\forall I\ f\ acc\ \ell,\ \left(\forall acc'\ v\ t,\qquad (f\ acc'\ v)\qquad\right) \rightarrow$$

Idea: Use the proof term of `fold_left_spec`

$$\{I\ acc\ []\}$$

$$(\mathrm{List.fold\_left}\ f\ acc\ \ell)$$

$$\exists res, \{I\ res\ \ell\}$$

Testing Candidate Invariants

```
let fold_left f acc ls =

  match ls with
  | [] ->

    acc
  | h :: t ->

    let acc' = f acc h in

    fold_left f acc' t
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  {I  acc  ℓ'}
  match ls with
  | [] ->

    acc
  | h :: t ->

    let acc' = f acc h in

    fold_left f acc' t
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  {I  acc  ℓ'}
  match ls with
  | [] ->
    {I  acc  ℓ'}
    acc
  | h :: t ->

    let acc' = f acc h in

    fold_left f acc' t
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  {I acc ℓ'}
  match ls with
  | [] ->
    {I acc ℓ'}
    acc
  | h :: t ->
    {I acc ℓ'}
    let acc' = f acc h in

    fold_left f acc' t
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  {I acc ℓ′}
  match ls with
  | [] ->
    {I acc ℓ′}
    acc
  | h :: t ->
    {I acc ℓ′}
    let acc' = f acc h in
    {I acc′ (ℓ′ ++ [h])}
    fold_left f acc' t
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  $\{I\ \ acc\ \ \ell'\}$
  match ls with
  | [] ->
    $\{I\ \ acc\ \ \ell'\}$
    acc
  | h :: t ->
    $\{I\ \ acc\ \ \ell'\}$
    let acc' = f acc h in
    $\{I\ \ acc'\ \ (\ell' +\!\!+ [h])\}$
    fold_left f acc' t
    $\{I\ \ acc\ \ (\ell' +\!\!+ \ell)\}$
```

Testing Candidate Invariants

```
let fold_left f acc ls =
  $\{I \ acc \ \ell'\}$
  match ls with
  | [] ->
    $\{I \ acc \ \ell'\}$
    acc
  | h :: t ->
    $\{I \ acc \ \ell'\}$
    let acc' = f acc h in
    $\{I \ acc' \ (\ell' +\!\!+ [h])\}$
    fold_left f acc' t
    $\{I \ acc \ (\ell' +\!\!+ \ell)\}$
```

Describes **exactly** how *I* is maintained

Testing Candidate Invariants

```
let fold_left f acc ls =
  $\{I\ acc\ \ell'\}$
  match ls with
  | [] ->
    $\{I\ acc\ \ell'\}$
    acc
  | h :: t ->
    $\{I\ acc\ \ell'\}$
    let acc' = f acc h in
    $\{I\ acc'\ (\ell' +\!\!+ [h])\}$
    fold_left f acc' t
    $\{I\ acc\ (\ell' +\!\!+ \ell)\}$
```

Describes **exactly** how *I* is maintained

Proof-driven Testing

Testing Candidate Invariants

`fold_left_spec`

Testing Candidate Invariants

$$\texttt{fold\_left\_spec}\ ?I\ \ f\ \ 2\ \ [2;1]\ \ ?HI$$

Testing Candidate Invariants

Instantiate with concrete arguments..

$\texttt{fold\_left\_spec}\ ?I\ \ f\ \ 2\ \ [2;1]\ \ ?HI$

Testing Candidate Invariants

Instantiate with concrete arguments..

`fold_left_spec` $?I \; f \; 2 \; [2;1] \; ?HI$

Testing Candidate Invariants

Instantiate with concrete arguments..

`fold_left_spec` $?I\ \ f\ 2\ [2;1]\ ?HI$

...with existentials for proof arguments

Testing Candidate Invariants

$\texttt{fold\_left\_spec}\ ?I\ \ f\ \ 2\ \ [2; 1]\ \ ?HI$

Testing Candidate Invariants

$\texttt{fold\_left\_spec}\ ?I\ \ f\ \ 2\ \ [2;1]\ \ ?HI$

Testing Candidate Invariants

`fold_left_spec` $?I \; f \; 2 \; [2; 1] \; ?HI$

reduce proof term

↓

Testing Candidate Invariants

`fold_left_spec` $?I\ f\ 2\ [2;1]\ ?HI$

reduce proof term

↓

```
(* ...reduced proof term... *)
```

Testing Candidate Invariants

`fold_left_spec` $?I\ f\ 2\ [2; 1]\ ?HI$

reduce proof term

`(* ...reduced proof term... *)`

custom proof extraction

Testing Candidate Invariants

```
assert (I len []);
let acc = f len 2 in
assert (I acc [2]);
let acc = f acc 1 in
assert (I acc [2; 1]);
()
```

Testing Candidate Invariants

```
assert (I len []);
let acc = f len 2 in
assert (I acc [2]);
let acc = f acc 1 in
assert (I acc [2; 1]);
()
```

Simulates an
execution of
`List.fold_left`

Testing Candidate Invariants

Instantiate I with embedding of candidate invariant…

```
assert (I len []);
let acc = f len 2 in
assert (I acc [2]);
let acc = f acc 1 in
assert (I acc [2; 1]);
()
```

Testing Candidate Invariants

Instantiate I with embedding of candidate invariant…

```
assert (I len []);
let acc = f len 2 in
assert (I acc [2]);
let acc = f acc 1 in
assert (I acc [2; 1]);
()
```

…prune candidate if assertion raised.

Key challenges for proof repair

① *Generating* candidate invariants

② *Testing* generated invariants

Key challenges for proof repair

① *Generating* candidate invariants

…using *relevant* expressions from old proof

② *Testing* generated invariants

Key challenges for proof repair

① *Generating* candidate invariants

...using *relevant* expressions from old proof

② *Testing* generated invariants

...using *proof-driven testing*

Research Questions

① Is Sisyphus effective at repairing proofs?

② Does Sisyphus repair proofs in reasonable time?

③ What changes does Sisyphus handle poorly?

Benchmark Programs

- 14 OCaml programs and their changes
- 10 from real-world OCaml codebases
- ...such as `containers` or Jane Street’s `core`

Benchmark Programs

Example	Data Structure	Refactoring
seq_to_array	Array, Seq	IterOrd, DataStr
make_rev_list†	Ref	Mutable/Pure
tree_to_array†	Array, Tree	IterOrd, DataStr
array_exists	Array	Mutable/Pure
array_find_mapi	Array, Ref	Pure/Mutable
array_is_sorted	Array	Pure/Mutable
array_findi	Array	Pure/Mutable
array_of_rev_list	Array	DataStr
array_foldi	Array	Pure/Mutable
array_partition	Array	DataStr
stack_filter‡	Stack	DataStr
stack_reverse‡	Stack	DataStr
sll_partition‡	SLL	Mutable/Pure, IterOrd
sll_of_array‡	Array, SLL	IterOrd

RQ1: Effectiveness of proof repair

Name	Time (old)			# Admits / # Obligations	Time (new)
	Spec	Proof	Total		
seq_to_array	1hrs	1hr	2hrs	3 / 5	17m
make_rev_list	5m	5m	10m	0 / 2	-
tree_to_array	4hrs	1hr	5hrs	2 / 4	18m
array_exists	10m	20m	30m	2 / 4	12m
array_find_mapi	30m	1hr	1.5hrs	2 / 5	12m
array_is_sorted	1hr	3hrs	4hrs	2 / 5	2m
array_findi	30m	1hr	1.5hrs	3 / 7	9m
array_of_rev_list	5m	1hr	1hr	2 / 3	3m
array_foldi	10m	5m	15m	0 / 1	-
array_partition	30m	2hrs	2.5hrs	3 / 3	5m
stack_filter	1hr	30m	1.5hrs	3 / 3	11m
stack_reverse	1.5hrs	30m	2hrs	1 / 1	30s
sll_partition	1hr	1hr	2hrs	0 / 2	-
sll_of_array	1.5hrs	30m	2hrs	0 / 1	-

RQ1: Effectiveness of proof repair

Name	Time (old)			# Admits / # Obligations	Time (new)
	Spec	Proof	Total		
seq_to_array	1hrs	1hr	2hrs	3 / 5	17m
make_rev_list	5m	5m	10m	0 / 2	-
tree_to_array	4hrs	1hr	5hrs	2 / 4	18m
array_exists	10m	20m	30m	2 / 4	12m
array_find_mapi	30m	1hr	1.5hrs	2 / 5	12m
array_is_sorted	1hr	3hrs	4hrs	2 / 5	2m
array_findi	30m	1hr	1.5hrs	3 / 7	9m
array_of_rev_list	5m	1hr	1hr	2 / 3	3m
array_foldi	10m	5m	15m	0 / 1	-
array_partition	30m	2hrs	2.5hrs	3 / 3	5m
stack_filter	1hr	30m	1.5hrs	3 / 3	11m
stack_reverse	1.5hrs	30m	2hrs	1 / 1	30s
sll_partition	1hr	1hr	2hrs	0 / 2	-
sll_of_array	1.5hrs	30m	2hrs	0 / 1	-

RQ1: Effectiveness of proof repair

Name	Time (old)			# Admits / # Obligations	Time (new)
	Spec	Proof	Total		
seq_to_array	1hrs	1hr	2hrs	3 / 5	17m
make_rev_list	5m	5m	10m	0 / 2	-
tree_to_array	4hrs	1hr	5hrs	2 / 4	18m
array_exists	10m	20m	30m	2 / 4	12m
array_find_mapi	30m	1hr	1.5hrs	2 / 5	12m
array_is_sorted	1hr	3hrs	4hrs	2 / 5	2m
array_findi	30m	1hr	1.5hrs	3 / 7	9m
array_of_rev_list	5m	1hr	1hr	2 / 3	3m
array_foldi	10m	5m	15m	0 / 1	-
array_partition	30m	2hrs	2.5hrs	3 / 3	5m
stack_filter	1hr	30m	1.5hrs	3 / 3	11m
stack_reverse	1.5hrs	30m	2hrs	1 / 1	30s
sll_partition	1hr	1hr	2hrs	0 / 2	-
sll_of_array	1.5hrs	30m	2hrs	0 / 1	-

RQ2: Efficiency of proof repair

Example	Time (s)				Total (s)
	Generation	Extraction	Testing	Remaining	
seq_to_array	28.57	1.95	20.36	5.28	58
make_rev_list	$\leq 10ms$	3.36	$\leq 10ms$	11.95	15
tree_to_array	6.75	1.95	2.98	13.32	25
array_exists	$\leq 10ms$	3.30	$\leq 10ms$	13.23	17
array_find_mapi	$\leq 10ms$	2.13	$\leq 10ms$	13.95	17
array_is_sorted	$\leq 10ms$	2.04	$\leq 10ms$	15.38	18
array_findi	$\leq 10ms$	2.13	$\leq 10ms$	19.07	22
array_of_rev_list	1.72	2.82	0.96	15.62	21
array_foldi	$\leq 10ms$	488.89	$\leq 10ms$	15.00	504
array_partition	3.51	69.73	2.62	17.53	95
stack_filter	$\leq 10ms$	81.88	$\leq 10ms$	21.53	104
stack_reverse	$\leq 10ms$	88.42	$\leq 10ms$	16.94	105
sll_partition	$\leq 10ms$	426.62	$\leq 10ms$	16.43	443
sll_of_array	0.02	55.98	0.01	13.33	69

RQ2: Efficiency of proof repair

Example	Time (s)				Total (s)
	Generation	Extraction	Testing	Remaining	
seq_to_array	28.57	1.95	20.36	5.28	58
make_rev_list	$\leq$ 10*ms*	3.36	$\leq$ 10*ms*	11.95	15
tree_to_array	6.75	1.95	2.98	13.32	25
array_exists	$\leq$ 10*ms*	3.30	$\leq$ 10*ms*	13.23	17
array_find_mapi	$\leq$ 10*ms*	2.13	$\leq$ 10*ms*	13.95	17
array_is_sorted	$\leq$ 10*ms*	2.04	$\leq$ 10*ms*	15.38	18
array_findi	$\leq$ 10*ms*	2.13	$\leq$ 10*ms*	19.07	22
array_of_rev_list	1.72	2.82	0.96	15.62	21
array_foldi	$\leq$ 10*ms*	488.89	$\leq$ 10*ms*	15.00	504
array_partition	3.51	69.73	2.62	17.53	95
stack_filter	$\leq$ 10*ms*	81.88	$\leq$ 10*ms*	21.53	104
stack_reverse	$\leq$ 10*ms*	88.42	$\leq$ 10*ms*	16.94	105
sll_partition	$\leq$ 10*ms*	426.62	$\leq$ 10*ms*	16.43	443
sll_of_array	0.02	55.98	0.01	13.33	69

RQ2: Efficiency of proof repair

Example	Time (s)				Total (s)
	Generation	Extraction	Testing	Remaining	
seq_to_array	28.57	1.95	20.36	5.28	58
make_rev_list	$\leq 10ms$	3.36	$\leq 10ms$	11.95	15
tree_to_array	6.75	1.95	2.98	13.32	25
array_exists	$\leq 10ms$	3.30	$\leq 10ms$	13.23	17
array_find_mapi	$\leq 10ms$	2.13	$\leq 10ms$	13.95	17
array_is_sorted	$\leq 10ms$	2.04	$\leq 10ms$	15.38	18
array_findi	$\leq 10ms$	2.13	$\leq 10ms$	19.07	22
array_of_rev_list	1.72	2.82	0.96	15.62	21
array_foldi	$\leq 10ms$	488.89	$\leq 10ms$	15.00	504
array_partition	3.51	69.73	2.62	17.53	95
stack_filter	$\leq 10ms$	81.88	$\leq 10ms$	21.53	104
stack_reverse	$\leq 10ms$	88.42	$\leq 10ms$	16.94	105
sll_partition	$\leq 10ms$	426.62	$\leq 10ms$	16.43	443
sll_of_array	0.02	55.98	0.01	13.33	69

RQ3: Failure Modes

- Repair assumes components from old proof are sufficient for new one.

- Quality of repair degrades when this fails to hold.

e.g. `array_partition`’s pure obligations required fact

$$\texttt{filter}\, p\, (\texttt{filter}\, p\, \ell) = \texttt{filter}\, p\, \ell$$

not present in original proof.

RQ3: Failure Modes

- Repair assumes components from old proof are sufficient for new one.
- Quality of repair degrades when this fails to hold.
- In some cases, repair may even fail.

RQ3: Failure Modes

```
let to_array s =
  let batches = (* .. *) in
  let res =
      Array.make (* .. *) in
  List.iter (fun batch ->
      let dst = (* .. *) in
      Array.copy batch res dst)
    batches;
  res
```

RQ3: Failure Modes

Invariant requires *flattening* operation...

```
let to_array s =
  let batches = (* .. *) in
  let res =
      Array.make (* .. *) in
  List.iter (fun batch ->
      let dst = (* .. *) in
      Array.copy batch res dst)
    batches;
  res
```

RQ3: Failure Modes

Invariant requires *flattening* operation...

```
let to_array s =
  let batches = (* .. *) in
  let res =
      Array.make (* .. *) in
  List.iter (fun batch ->
      let dst = (* .. *) in
      Array.copy batch res dst)
    batches;
  res
```

...not present in old proof.

Summary

1. **Sisyphus**: tool to *repair* proofs over changes.

2. **PDT**: technique to *efficiently* test invariants.

3. Evaluation on 10 real OCaml programs and their changes