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.
OCaml’s 'a Seq.t datatype

```ocaml
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 \texttt{Seq.t} datatype

type 'a t = unit -> 'a node 
and 'a node = Nil 
\quad | \text{Cons of 'a } \times (\text{unit } \rightarrow \ 'a \text{ 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))
Let’s write a verified program!

Q: Convert a sequence to an array.
Let’s write a verified program!

Q: Convert a sequence to an array.
let to_array s =
let to_array s =
  match s () with


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

```
let to_array s =
   match s () with
   | Nil  -> [ | | ]
let to_array s =
  match s () with
  | Nil -> []
  | Cons (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;
    a
let to_array s =
  match s () with
  | Nil  -> []
  | Cons (h, _) ->
    let sz = length s in
    let a = make sz h in
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;
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
Let’s write a verified program!

Q: Convert a sequence to an array.
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.

\[ \forall s \ell, \{ s \mapsto \text{Seq } \ell \} \]

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

\[ \exists a, \{ a \mapsto \text{Array } \ell \} \]
Q: Convert a sequence to an array.

∀s ℓ, {s ↦ Seq ℓ}

(to_array s)

∃a, {a ↦ Array ℓ}
Q: Convert a sequence to an array.

\[ \forall s \ell, \{ s \mapsto \text{Seq } \ell \} \]

(to_array s)

\[ \exists a, \{ a \mapsto \text{Array } \ell \} \]
Q: Convert a sequence to an array.

\[ \forall s \in \mathbb{L}, \{ s \mapsto \text{Seq} \in \mathbb{L} \} \]

\[ \text{(to_array } s) \]

\[ \exists a, \{ a \mapsto \text{Array} \in \mathbb{L} \} \]
Q: Convert a sequence to an array.

\[ \forall s \; \ell, \{ s \mapsto \text{Seq } \ell \} \]

(to_array \; s)

\[ \exists a, \{ a \mapsto \text{Array } \ell \} \]

“a” points-to an array
Q: Convert a sequence to an array.

\[ \forall s \ell, \{ s \mapsto \text{Seq } \ell \} \]

(to_array s)

\[ \exists a, \{ a \mapsto \text{Array } \ell \} \]

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

Using CFML2
verification library

Charguéraud (2023)
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.

xcf.xapp; case ℓ as [ | h tl] |
  - xvalemptryarr.
  -
    xapp.
    xalloc.
    xapp (iteri_spec (λt →
      a ↦ Array (t ++ drop (length t)
      (make (length ℓ) h)))
    ).
    xval.
\begin{verbatim}
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
\end{verbatim}
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. xafapp; case ℓ as [ | h t |]
    — xvalemptyarr.
    —
        xapp.
        xalloc.
        xapp (iteri SPEC (λ t ->
            a的带领下 Array (t ++ drop (length t)
            (make (length ℓ) h)))
        ).
        xval.
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.

taxapp; case ℓ as [[ h tl]
  — xvalempyarr.
  —
    taxapp.
    txalloc.
    txapp (iteri_spec (λ t →
      a → Array (  
        t ⊢ drop (length t) 
        (make (length ℓ) h))
    )).
    txval.

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]
  — xvalempyarr.
  —
    xapp.
    xalloc.
    xapp (iterispec (λ t →
      a → Array (t ++ drop (length t)
        (make (length ℓ) h)))
    ).
    xval.
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]
  — xvalemptryarr.
  —
  xapp.
xalloc.
xapp (iteri_spec (λt →
    a ↦ Array (t ++ drop (length t)
      (make (length ℓ) h))
    )).
  xval.
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 t l |
  — xvalemptyarr.
  —
  xapp.
  xalloc.

xapp (iteri_spec (λt →
  a → Array(
    t ++ drop (length t)
    (make (length ℓ) h))
  ).
  xval.
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 (λ t →
      a → Array (t ++ drop (length t)
        (make (length ℓ) h))
    )).
xval.
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.
```ocaml
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
```

```ocaml
xcf.
xapp; case ℓ as [ | h tl] − xvalemptryarr.
−
xapp.
xalloc.
xapp (iteri_spec (λt →
    a → Array (t ++ drop (length t)
      (make (length ℓ) h)))
 ).
xval.
```
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|]
  - xvalempyarr.
    -
      xapp.
      xalloc.
      xapp(iteri_spec(λ t →
        a ↦ Array(
          t ++ drop(length t)
          (make (length ℓ) h))
        )).
      xval.
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 |]
    — xvalempyarr.
    —
    xapp.
    xalloc.
    xapp (iteri_spec (λt →
      a → Array (t ++ drop (length t)
        (make (length ℓ) h)))
    ).
xval.
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
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]]
  — xvalemptryarr.
  —
    xapp.
    xalloc.
    xapp (iteri_spec (λt →
      a ↦ Array (t ++ drop (length t)
          (make (length ℓ) h))
    ).
    xval.
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] − xvalempytarr.

−

xapp.

xalloc.

xapp (iteri_spec (λ t ->
  a → Array (t ++ drop (length t) (make (length ℓ) h)))
).

xval.
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

a \mapsto \text{Array}(t \mapsto \text{drop}(\text{length } t) \text{(make } (\text{length } \ell) 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;
a
a

\[
\begin{align*}
a & \mapsto \text{Array}(t + \text{drop}(\text{length } t)(\text{make}(\text{length } \ell) h)) \\
\mid \text{Cons } (h, \_ ) & \rightarrow \quad - \\
\text{let } sz = \text{length } s \text{ in} & \quad \text{xapp}. \\
\text{let } a = \text{make } sz \text{ h in} & \quad \text{xalloc}. \\
\text{iteri } (\text{fun } i \text{ vl } \rightarrow & \quad \text{xapp}(\text{iteri_spec}(\lambda t \rightarrow \\
& a \mapsto \text{Array}(t + \text{drop}(\text{length } t)(\text{make}(\text{length } \ell) h)) \\
& ) \text{ s}; \\
& a
\end{align*}
\]
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
a

\( a \mapsto \text{Array}(t \quad \text{drop} \quad (\text{length } t) \quad (\text{make} \quad (\text{length } \ell) \quad h)) \)
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;
    a

a ─→ Array(t ++ drop (length t) (make (length ℓ) 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;
    a

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

fork c-cube / ocaml-containers

star 440 stars
A problem arises...

Make Seq.to_array behave better with stateful sequences #390

shonfeder commented on Dec 12, 2021 • edited

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

The change is motivated by the need to handle stateful sequences.
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
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

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
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
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

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
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

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
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

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
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

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
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

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

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

New

```ocaml
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...
Writing verified code is hard...
Writing verified code is hard...

Proof Repair
Maintaining verified code is hard...

Proof Repair

Ringer (2021)
Writing verified code is hard...

...can we make it easier?
Writing verified code is hard...

...can we make it easier?

Yes!
This Work

Mostly automated proof-repair for OCaml programs
Outline

1. Motivation
2. Key Challenges & Solutions
3. Evaluation
New Program:

```ml
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:

```ocaml
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:

```ocaml
let to_array s =

  let sz, ls = fold_left
  ! (fun (i, acc) x ->
  !   (i+1, x::acc))
  ! (0, []) l

  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?

**Observation:** proofs are syntax-directed

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

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

xf.

case
  l
as
  [; init rest].
  - xmatch 0.
  - xvalemptyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
           (fun acc t =>
             (??)
           )).
    xvals.
    /two.tf/zero.tf//five.tf/four.tf
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

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

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. xvalempyarr.
  - xmatch 1.
    xalloc a data Hdata.
    xlet idx.
    xapp (fold_left_spec idx rest
      (fun acc t =>
        (??))).
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 t =>
            (??)))

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 t =>
        (??)))).

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. xvalempyarr.
  - xmatch 1.
    xalloc a data Hdata.
xlet idx.
xapp (fold_left_spec idx rest
      (fun acc t =>
         (??))).
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

\[2\\text{xcf.} \quad \text{xapp} \ldots\] \[2\\text{case l as [ init rest].} \]
\[- xmatch 0. xvalemptyarr. \]
\[- xmatch 1. \]
\[- xalloc a data Hdata. \]
\[2\\text{xlet idx.} \quad \text{xapp (fold_left_spec idx rest}
\[2\\text{(fun acc t =>
\[2\\quad (??))}). \]
\[2\\text{xvals.} \]
Key challenges for proof repair

1. Generating candidate invariants

2. Choosing valid invariants
Key challenges for proof repair

1. Generating candidate invariants

2. Choosing valid invariants
How to generate invariants?
How to generate invariants?

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—

```ocaml
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
```

```ocaml
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—

```ocaml
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
```

---

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ocaml
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
```

```ocaml
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

```ml
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

```ocaml
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
```

```ocaml
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

```ocaml
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

Similar behaviour...

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

```ocaml
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
```

```ocaml
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
```

Similar behaviour...

..similar invariants?
Q: How to discover these similarities automatically?
Generating candidate invariants

**Q:** How to discover these similarities automatically?

**A:** Instrumentation based dynamic analysis
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
  | 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
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 =

  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

Seq.of_list [1; 2; 3]
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 =

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 =
  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 =

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 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
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
  | [] -> []
  | x::rest ->
    let a = make sz x in
    let idx = sz - 2 in
    let _ =
      List.fold_left
        (fun i x -> a.(i) <- x; i - 1)
      idx rest in
    a

Seq.of_list [1; 2; 3]
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 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

Seq.of_list [1; 2; 3]
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
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

Seq.of_list [1; 2; 3]
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
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

Seq.of_list [1; 2; 3]
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

Seq.of_list [1; 2; 3]
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
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;

Seq.of_list [1; 2; 3]
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
```ocaml
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
```

```ocaml
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
```
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
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 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
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
```ocaml
let to_array l = 
  match l () with
  | Nil -> []
  | Cons (x, _) ->
    let len = length' l in
    let a = make len x in
    let _ = List.fold_left (fun i x -> a.(i) <- x; i - 1) idx rest in
    a
```

```ocaml
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
```

Seq.of_list [1; 2; 3]

Seq.of_list [1; 2; 3]
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 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

Seq.of_list [1;2;3]
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 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
```ocaml
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
```

```
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
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```

```
Seq.of_list [; ;]
```
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
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
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 =
    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

Seq.of_list [1;2;3]
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 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
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
Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

Identify “similar” program points through traces.
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.
Old Invariant:

\[ a \mapsto \text{Array}(t \mapsto \text{drop}(\text{length } t) \ (\text{make} \ (\text{length } \ell) \ h)) \]
Old Invariant:

\[ a \mapsto Array(t \mapsto drop(length t)(make(length \ell) h)) \]

New Invariant:
Old Invariant:

$$a \mapsto \text{Array}(t +\!\!+ \text{drop}(\text{length } t) (\text{make}(\text{length } \ell) h))$$

New Invariant:

$$(\text{fun } \text{acc } t \Rightarrow a \mapsto \text{Array}(\text{repeat}(\text{acc } + 1) \text{ init } ++$$
$$\text{drop}(\text{acc } + 1) \ell))$$

(1)
Old Invariant:

\[ a \mapsto \text{Array}(t \mathbin{\mathit{++}} \text{drop}(\text{length } t) \ (\text{make}(\text{length } \ell) \ h)) \]

New Invariant:

\[
\begin{align*}
(\text{fun} \ acc \ t &= a \mapsto \\
& \text{Array} \ (\text{repeat} \ (acc + 1) \ init \ ++ \ \cdots \ \\
& \quad \text{drop} \ (acc + 1) \ \ell))
\end{align*}
\]

(1)
Old Invariant:

\[ a \mapsto \text{Array}(t \mathbin{++} \text{drop}(\text{length } t) (\text{make}(\text{length } \ell) h)) \]

New Invariant:

\[
(\text{fun } \text{acc } t => \ a \mapsto \\
\quad \text{Array}(\text{repeat}(\text{acc + 1}) \text{ init } \mathbin{++} \\
\quad \quad \quad \text{drop}(\text{acc + 1}) \ell))
\]

(1)

\[
\cdots 
\quad \text{Array}([\ ] \mathbin{++} \text{drop } 0
\quad \quad \text{(repeat}(\text{length } t) \text{ init}))
\]

(n)
Key challenges for proof repair

1. Generating candidate invariants

2. Choosing valid invariants
Key challenges for proof repair

1. **Generating candidate invariants**
2. **Choosing valid invariants**
Choosing an Invariant
Choosing an Invariant

...  
xlet idx.  
xapp (fold_left_spec idx rest  
 (fun acc t =>  
  (??)  )).  

...
Choosing an Invariant

...  
xlet idx.  
xapp (fold_left_spec idx rest  
(fun acc t =>  
  (??)  )).  
...

(fun acc t => a ↦→  
  Array (repeat (acc + 1) init ++  
  drop (acc + 1) ℓ))  

(fun acc t => a ↦→  
  Array ([] ++ drop 0  
  (repeat (length t) init)))
Choosing an Invariant

... xlet idx.
  xapp (fold_left_spec idx rest
  (fun acc t =>
  (??) ))).
...

(fun acc t => a ↦
  Array (repeat (acc + 1) init ++
  drop (acc + 1) ℓ))

(fun acc t => a ↦
  Array ([]) ++ drop 0
  (repeat (length t) init)))
Choosing an Invariant

\[
\begin{align*}
&\text{let idx.} \\
&\text{xapp (fold_left_spec idx rest} \\
&\text{(fun acc t =>} \\
&\text{(??) )))}.
\end{align*}
\]

\[
\begin{align*}
&\text{(fun acc t => a \mapsto} \\
&\text{Array (repeat (acc + 1) init ++} \\
&\text{drop (acc + 1) \ell})
\end{align*}
\]
Choosing an Invariant

... 
\begin{align*}
\text{xlet } idx. \\
\text{xapp } (\text{fold\_left\_spec } idx \text{ rest } \\
\text{(fun } acc \text{ t }=\rightarrow \\
\quad (\text{??}))) \\
\end{align*}

... 

\begin{align*}
\text{(fun } acc \text{ t }=\rightarrow a \mapsto \\
\quad \text{Array (repeat (acc + 1) init ++ drop (acc + 1) } \ell))) \\
\end{align*}

\begin{align*}
\text{(fun } acc \text{ t }=\rightarrow a \mapsto \\
\quad \text{Array ([] ++ drop 0 (repeat (length t) init)))}
\end{align*}
Choosing an Invariant

... xlet idx.
xapp (fold_left_spec idx rest
  (fun acc t =>
   (??))
).
...

Which one?

(fun acc t => a ➞
 Array (repeat (acc + 1) init ++
 drop (acc + 1) ℓ))

(fun acc t => a ➞
 Array ([] ++ drop 0
 (repeat (length t) init)))
Choosing an Invariant

\[
\begin{align*}
\text{xlet idx.} \\
\text{xapp (fold_left_spec idx rest} \\
(\text{fun acc t =>} \\
(??)) \text{)).}
\end{align*}
\]

Which one? Validate using SMT?

\[
(\text{fun acc t => a } \mapsto \text{Array (repeat (acc + 1) init ++} \\
\text{drop (acc + 1) } \ell))
\]

\[
(\text{fun acc t => a } \mapsto \text{Array ([]} ++ \text{drop 0} \\
\text{(repeat (length t) init))})
\]
Choosing an Invariant

... xlet idx.
   xapp (fold_left_spec idx rest
   (fun acc t =>
      (??) ))).
...

Which one?

Validate using SMT?

Too slow!

(fun acc t => a ↦
   Array (repeat (acc + 1) init ++
         drop (acc + 1) ℓ))

(fun acc t => a ↦
   Array ([] ++ drop 0
          (repeat (length t) init)))
Choosing an Invariant

... xlet idx.
xapp (fold_left_spec idx rest
(fun acc t =>
  (??))).

Which one?

If only we could test...

(fun acc t => a \mapsto
  Array (repeat (acc + 1) init ++
  drop (acc + 1) ℓ))

(fun acc t => a \mapsto
  Array ([] ++ drop 0
  (repeat (length t) init)))
Choosing an Invariant

... xlet idx.
  xapp (fold_left_spec idx rest
    (fun acc => (?)(?))).
...

Which one?

(fun acc t => a ↦
  Array (repeat (acc + 1) init ++
    drop (acc + 1) ℓ))

(fun acc t => a ↦
  Array ([] ++ drop 0
    (repeat (length t) init)))

Depends on logical parameters!

If only we could test...
Choosing an Invariant

... xlet idx.
xapp (fold_left_spec idx rest (fun acc => (??))).

Which one?

Depends on logical parameters!

(fun acc t => a ↦→
 Array (repeat (acc + 1) init ++
      drop (acc + 1) ℓ))

(fun acc t => a ↦→
 Array ([] ++ drop 0
      (repeat (length t) init)))
Choosing an Invariant

... 
**xlet** idx.
**xapp** (**fold_left_spec** idx rest
  (**fun** acc t =>
    (??) ))).
...

Choosing an Invariant

... 
\textbf{xlet} \texttt{idx}.
\textbf{xapp} (\texttt{fold_left_spec} \texttt{idx} \texttt{rest} 
\textbf{fun} \texttt{acc} \texttt{t} =>
  (?)))).

...
Verified Fold Left

```ocaml
define fold_left f acc ls
  =

  match ls with
  | [] ->
    acc
  | h :: tl ->

    let acc' = f acc h
    in
    fold_left f acc' tl
```
Verified Fold Left

\[
\text{let } \text{fold\_left } f \text{ acc ls } t = \\]

\[
\text{match ls with} \]
\[
| [] -> \]
\[
\text{acc} \]
\[
| h :: tl -> \]
\[
\text{let } \text{acc'} = f \text{ acc h in} \]
\[
\text{fold\_left } f \text{ acc'} tl \]
let fold_left f acc ls t /

match ls with
| [] -> acc
| h :: tl ->

let acc' = f acc h in
fold_left f acc' tl
Verified Fold Left

```ocaml
let fold_left f acc ls t / =
{ /
  acc t }
match ls with
| [] ->

  acc
| h :: tl ->

  let acc' = f acc h in

  fold_left f acc' tl
```
Verified Fold Left

\[
\text{let } \text{fold\_left } f \text{ acc } ls \text{ t / } = \\
\{l \text{ acc } t\} \\
\text{match } ls \text{ with} \\
| [] \rightarrow \\
\{l \text{ acc } t\} \\
\text{acc} \\
| h :: tl \rightarrow \\
\text{let acc' } = f \text{ acc } h \text{ in} \\
\text{fold\_left } f \text{ acc'} \text{ tl}
\]
Verified Fold Left

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

Verified Fold Left

```ocaml
let fold_left f acc ls t l =
{! acc t}
match ls with
| [] ->
{! acc t}
acc
| h :: tl ->
{! acc t}
let acc' = f acc h in
{! acc' (t ++ [h])}
fold_left f acc' tl
```
let fold_left f acc ls t l =
{l acc t}
match ls with
| [] ->
{l acc t}
acc
| h :: tl ->
{l acc t}
let acc' = f acc h in
{l acc' (t ++ [h])}
fold_left f acc' tl
{l acc'' (t ++ [h] ++ tl)}
Verified Fold Left

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

Describes exactly how \( I \) is maintained
Verified Fold Left

```ocaml
define fold_left (f: 'a -> 'b -> 'a) (acc: 'a) (ls: 'a list) (t: 'b) : 'a =
  {l acc t}
match ls with
| [] ->
  {l acc t}
  acc
| h :: tl ->
  {l acc t}
  let acc' = f acc h in
  {l acc' (t ++ [h])}
  fold_left f acc' tl
  {l acc'' (t ++ [h] ++ tl)}
```
Verified Fold Left

```ocaml
let fold_left f acc ls t l =
  assert { l acc t }
  match ls with
  | [] ->
    assert { l acc t }
    acc
  | h :: tl ->
    assert { l acc t }
    let acc' = f acc h in
    assert { l acc' (t ++ [h]) }
    fold_left f acc' tl
    assert { l acc'' (t ++ [h] ++ tl) }
```
 Verified Fold Left

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

Proof-Driven Testing
Proof-Driven Testing

fold_left_spec
Proof-Driven Testing

fold_left_spec ?/ f 2 [2;1]
Proof-Driven Testing

*Instantiate with concrete arguments..*

```
fold_left_spec ?/ f 2 [2;1]
```
Proof-Driven Testing

*Instantiate with concrete arguments..*

\[\text{fold\_left\_spec \ ?/ } f \ 2 \ [2;1]\]
Proof-Driven Testing

*Instantiate with concrete arguments*..

```
fold_left_spec ?/ f 2 [2; 1]
```

*...with existentials for proof arguments*
Proof-Driven Testing

fold_left_spec ?/ f 2 [2; 1]
Proof-Driven Testing

\texttt{fold_left_spec \texttt{?/ f 2 [2;1]}}
Proof-Driven Testing

fold_left_spec ?/ f 2 [2; 1]
| reduce proof term
↓
Proof-Driven Testing

```ocaml
let fold_left f acc ls t =
  {l acc t}
match ls with
  | [] ->
    {l acc t}
    acc
  | h :: tl ->
    {l acc t}
    let acc' = f acc h in
    {l acc' (t ++ [h])}
    fold_left f acc' tl
    {l acc'' (t ++ [h] ++ tl)}
```
Proof-Driven Testing

```
let fold_left f acc ls t =
  (/ 2 [])
match ls with
  | [] ->
    (/ 2 [])
    acc
  | 2 :: [1] ->
    (/ 2 [])
    let acc' = f 2 2 in
    (/ acc' [2])
    fold_left f acc' tl
    (/ acc'' [2,1])
```
Proof-Driven Testing

\[
\text{fold_left_spec} \ ?/ f \ 2 \ [2;1] \\
\downarrow \\
\text{reduce proof term} \\
\downarrow \\
(* \ ...\text{reduced proof term}... \ *)
\]
Proof-Driven Testing

\[
\text{fold\_left\_spec} ?/ f 2 [2;1] \\
\downarrow \\
\text{reduce proof term} \\
\downarrow \\
(* \ldots \text{reduced proof term} \ldots *) \\
\downarrow \\
\text{custom proof extraction}
\]
Proof-Driven Testing

```ocaml
let fold_left f acc ls =
  /
  /
  | 2 []
match ls with
  | [] ->
    /
    /
    | 2 []
    /
    /
    acc
  | 2 :: [1] ->
    /
    /
    | 2 []
    /
    /
    let acc' = f 2 2 in
    /
    /
    | acc' [2]
    /
    /
    fold_left f acc' tl
    /
    /
    | acc'' [2,1]
```
Proof-Driven Testing

```
assert (/ len []);
let acc = f len 2 in
assert (/ acc [2]);
let acc = f acc 1 in
assert (/ acc [2; 1]);
()
Proof-Driven Testing

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

Simulates concrete run of `List.fold_left`
Proof-Driven Testing

*Instantiate I with embedding of candidate invariant...*

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

/three.tf/nine.tf//five.tf/four.tf
Proof-Driven Testing

Instantiate $I$ with embedding of candidate invariant...

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

...prune candidate if assertion raised.
Testing Candidate Invariants

```
... xlet idx.
xapp (fold_left_spec idx rest
  (fun acc t =>
    (??))).
...
```

```
(fun acc t => a ↦
  Array (repeat (acc + 1) init ++
    drop (acc + 1) ℓ))
```

```
(fun acc t => a ↦
  Array (length t) init))
```

Testing Candidate Invariants

\[
\begin{aligned}
&\quad xlet \ idx. \\
&\ xapp (\text{fold_left_spec idx rest} \\
&\quad (\text{fun acc t =>}) \\
&\quad (\text{??})). \\
&\quad ...
\end{aligned}
\]

\[
\begin{aligned}
&\quad (\text{fun acc t => a \mapsto}) \\
&\quad \text{Array (repeat (acc + 1) init ++ drop (acc + 1) l))}
\end{aligned}
\]

\[
\begin{aligned}
&\quad (\text{fun acc t => a \mapsto}) \\
&\quad \text{Array ([] ++ drop 0 (repeat (length t) init)))}
\end{aligned}
\]
Testing Candidate Invariants

```
... xlet idx.
xapp (fold_left_spec idx rest
  (fun acc t =>
    (??))).
...
```

- **Correct**
  ```
  (fun acc t => a ↦
    Array (repeat (acc + 1) init ++
      drop (acc + 1) ℓ))
  ```

- **Incorrect**
  ```
  (fun acc t => a ↦
    Array ([] ++ drop 0
      (repeat (length t) init)))
  ```
Proof-Driven Testing

Proofs are proofs

Proofs are programs

Proofs are tests

Proof are programs...
Proof-Driven Testing

Proofs are proofs

Proofs are programs

Proofs are tests

Proof are programs...

but, what do they compute?
Proof-Driven Testing

Proof are programs...
but, what do they compute?

Curry Howard: They establish logical facts.
Proof-Driven Testing

Proofs are proofs
Proofs are programs
Proofs are tests

Proof are programs...
but, what do they compute?

Curry Howard: They establish logical facts.

PDT: HO-proofs describe tests!
Sisyphus

Old Proof
- Specification
- Skeleton
- Invariants

New Program
- Old Program

New Proof
- Specification
- Skeleton
- Invariants
- Obligations

Skeleton Generation
Invariant Synthesis
Invariant Testing

Output
Old Proof

- Specification
- Skeleton
- Invariants

New Program

- Old Program

Sisyphus

- Skeleton Generation
- Invariant Synthesis
- Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations
Sisyphus

Old Proof

- Specification
- Skeleton
- Invariants

New Proof

- Specification
- Skeleton
- Invariants
- Obligations

Skeleton Generation

Invariant Synthesis

Invariant Testing

Old Program

New Program

(1) Skeleton Generation
Old Proof

- Specification
- Skeleton
- Invariants

New Program

Old Program

New Proof

- Specification
- Skeleton
- Invariants

Obligations

(1) Skeleton Generation
(2) Invariant Synthesis
(3) Invariant Testing
Outline

1. Motivation
2. Key Challenges & Solutions
3. Evaluation
Outline

1. Motivation
2. Key Challenges & Solutions
3. Evaluation
Pragmatic Concerns

1. Is Sisyphus effective at repairing proofs?

2. Does Sisyphus repair proofs in reasonable time?
Benchmark Programs

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

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

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Data Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array exists</td>
<td>Array</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>
# Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Data Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array exists</td>
<td>Array</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array exists</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>
# Benchmark Programs

## Array

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Stacks and mapi</td>
<td>Array</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>

## Seq

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sllofarray†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>

† indicates marked-up examples.
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>DataStr</td>
</tr>
</tbody>
</table>

---

### Seq

### Array

### Stack
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>
# Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Array Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>

Diagram:

- **Tree**
- **Array**
- **Seq**
- **Stack**
- **SLL**
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref</td>
<td>Pure/Immutable</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Immutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Immutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>SLL</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>DataStr</td>
</tr>
</tbody>
</table>

*Tree, Seq, Array, Stack, Queue, SLL*
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Array Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list †</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Tree to array †</td>
<td>Array, Tree</td>
<td>DataStr</td>
</tr>
<tr>
<td>Tree to array</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array foldi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter ‡</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse ‡</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sll partition ‡</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array ‡</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>

**Data Structures:**
- Array
- Stack
- Queue
- SLL

**Trees and Sequences:**
- Tree
- Seq

**Stack and Queues:**
- Stack
- Queue
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Structure</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq Ref</td>
<td>IterOrd, DataStr</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Array, Seq Ref</td>
<td>Mutable/Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree IterOrd</td>
<td>DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Array, Tree IterOrd</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array exists</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array findmapi</td>
<td>Array, Ref</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Array issorted</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array</td>
<td>Pure/Mutable</td>
</tr>
<tr>
<td>Arraypartition</td>
<td>Array</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack</td>
<td>DataStr</td>
</tr>
<tr>
<td>Sllpartition†</td>
<td>SLL</td>
<td>Mutable/Pure, IterOrd</td>
</tr>
<tr>
<td>Sll of array†</td>
<td>Array, SLL</td>
<td>IterOrd</td>
</tr>
</tbody>
</table>
## Benchmark Programs

<table>
<thead>
<tr>
<th>Example</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>Array, Seq, IterOrd</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Array, Ref, Pure</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Tree, IterOrd, Pure</td>
</tr>
<tr>
<td>Array exists</td>
<td>Array, DataStr</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array, DataStr</td>
</tr>
<tr>
<td>Array filter</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack, DataStr</td>
</tr>
<tr>
<td>Stack reverse†</td>
<td>Stack, DataStr</td>
</tr>
<tr>
<td>Sll partition†</td>
<td>Sll, DataStr</td>
</tr>
<tr>
<td>Arrayfindi</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Arrayofrevlist</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Arrayfoldi</td>
<td>Array, Pure</td>
</tr>
<tr>
<td>Arraypartition</td>
<td>Array, DataStr</td>
</tr>
<tr>
<td>Stackfilter</td>
<td>Stack, DataStr</td>
</tr>
<tr>
<td>Stackreverse</td>
<td>Stack, DataStr</td>
</tr>
<tr>
<td>Sllpartition</td>
<td>Sll, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Tree, Pure</td>
</tr>
<tr>
<td>Seq to array</td>
<td>Seq, IterOrd</td>
</tr>
<tr>
<td>Make rev list†</td>
<td>Ref, Pure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Order</th>
<th>Pure to Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Array</td>
</tr>
<tr>
<td>Tree</td>
<td>Stack</td>
</tr>
<tr>
<td>Sll</td>
<td>Queue</td>
</tr>
</tbody>
</table>

Note: † indicates refactoring tasks that are not pure or mutable.
Benchmark Programs

- **Tree**
- **Seq**
- **Array**
- **Stack**
- **Queue**
- **SLL**

**Iteration Order**

**Pure to Mutable**

**Mutable to Pure**
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Refactoring</th>
<th>Iteration Order</th>
<th>Pure to Mutable</th>
<th>Mutable to Pure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Tree</td>
<td>Array</td>
<td>Stack</td>
<td>SLL</td>
</tr>
<tr>
<td>Seq to array</td>
<td>Make rev list†</td>
<td>Array, Seq, Ref, IterOrd, DataStr</td>
<td>Array, Ref, IterOrd, DataStr</td>
<td>Stack, DataStr</td>
</tr>
<tr>
<td>Tree to array†</td>
<td>Tree to array†</td>
<td>Array, Array, Tree</td>
<td>Array, Array, ref, Tree</td>
<td>Stack, Tree</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>Array is sorted</td>
<td>Array, ref, Sorted</td>
<td>Array, Array, Sorted</td>
<td>Stack, Sorted</td>
</tr>
<tr>
<td>Array findi</td>
<td>Array findi</td>
<td>Array, ref, Findi</td>
<td>Array, ref, Findi</td>
<td>Stack, Findi</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>Array of rev list</td>
<td>Array, ref, of rev list</td>
<td>Array, ref, of rev list</td>
<td>Stack, of rev list</td>
</tr>
<tr>
<td>Array partition</td>
<td>Array partition</td>
<td>Array, ref, partition</td>
<td>Array, ref, partition</td>
<td>Stack, partition</td>
</tr>
<tr>
<td>Stack filter†</td>
<td>Stack reverse†</td>
<td>Stack, Array, Stack</td>
<td>Stack, Array, Stack</td>
<td>SLL, Stack</td>
</tr>
<tr>
<td>SLL partition†</td>
<td>SLL partition†</td>
<td>SLL, Array, SLL</td>
<td>SLL, Array, SLL</td>
<td>Stack, IterOrd, SLL</td>
</tr>
</tbody>
</table>

- † - indicates new refactorings.
**RQ1: Effectiveness of proof repair**

<table>
<thead>
<tr>
<th>Name</th>
<th># Admits / # Obligations</th>
<th>Time (old)</th>
<th>Time (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>3 / 5</td>
<td>2hrs</td>
<td>17m</td>
</tr>
<tr>
<td>Make rev list</td>
<td>0 / 2</td>
<td>10m</td>
<td>-</td>
</tr>
<tr>
<td>Tree to array</td>
<td>2 / 4</td>
<td>5hrs</td>
<td>18m</td>
</tr>
<tr>
<td>Array exists</td>
<td>2 / 4</td>
<td>30m</td>
<td>12m</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>2 / 5</td>
<td>1.5hrs</td>
<td>12m</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>2 / 5</td>
<td>4hrs</td>
<td>2m</td>
</tr>
<tr>
<td>Array findi</td>
<td>3 / 7</td>
<td>1.5hrs</td>
<td>9m</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>2 / 3</td>
<td>1hr</td>
<td>3m</td>
</tr>
<tr>
<td>Array foldi</td>
<td>0 / 1</td>
<td>15m</td>
<td>-</td>
</tr>
<tr>
<td>Array partition</td>
<td>3 / 3</td>
<td>2.5hrs</td>
<td>5m</td>
</tr>
<tr>
<td>Stack filter</td>
<td>3 / 3</td>
<td>1.5hrs</td>
<td>11m</td>
</tr>
<tr>
<td>Stack reverse</td>
<td>1 / 1</td>
<td>2hrs</td>
<td>30s</td>
</tr>
<tr>
<td>SLL partition</td>
<td>0 / 2</td>
<td>2hrs</td>
<td>-</td>
</tr>
<tr>
<td>SLL of array</td>
<td>0 / 1</td>
<td>2hrs</td>
<td>-</td>
</tr>
</tbody>
</table>
## RQ1: Effectiveness of proof repair

<table>
<thead>
<tr>
<th>Name</th>
<th># Admits / # Obligations</th>
<th>Time (old)</th>
<th>Time (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>3 / 5</td>
<td>2hrs</td>
<td>17m</td>
</tr>
<tr>
<td>Make rev list</td>
<td>0 / 2</td>
<td>10m</td>
<td>-</td>
</tr>
<tr>
<td>Tree to array</td>
<td>2 / 4</td>
<td>5hrs</td>
<td>18m</td>
</tr>
<tr>
<td>Array exists</td>
<td>2 / 4</td>
<td>30m</td>
<td>12m</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>2 / 5</td>
<td>1.5hrs</td>
<td>12m</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>2 / 5</td>
<td>4hrs</td>
<td>2m</td>
</tr>
<tr>
<td>Array findi</td>
<td>3 / 7</td>
<td>1.5hrs</td>
<td>9m</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>2 / 3</td>
<td>1hr</td>
<td>3m</td>
</tr>
<tr>
<td>Array foldi</td>
<td>0 / 1</td>
<td>15m</td>
<td>-</td>
</tr>
<tr>
<td>Array partition</td>
<td>3 / 3</td>
<td>2.5hrs</td>
<td>5m</td>
</tr>
<tr>
<td>Stack filter</td>
<td>3 / 3</td>
<td>1.5hrs</td>
<td>11m</td>
</tr>
<tr>
<td>Stack reverse</td>
<td>1 / 1</td>
<td>2hrs</td>
<td>30s</td>
</tr>
<tr>
<td>SLL partition</td>
<td>0 / 2</td>
<td>2hrs</td>
<td>-</td>
</tr>
<tr>
<td>SLL of array</td>
<td>0 / 1</td>
<td>2hrs</td>
<td>-</td>
</tr>
</tbody>
</table>
RQ1: Effectiveness of proof repair

<table>
<thead>
<tr>
<th>Name</th>
<th># Admits / # Obligations</th>
<th>Time (old)</th>
<th>Time (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>3 / 5</td>
<td>2hrs</td>
<td>17m</td>
</tr>
<tr>
<td>Make rev list</td>
<td>0 / 2</td>
<td>10m</td>
<td>-</td>
</tr>
<tr>
<td>Tree to array</td>
<td>2 / 4</td>
<td>5hrs</td>
<td>18m</td>
</tr>
<tr>
<td>Array exists</td>
<td>2 / 4</td>
<td>30m</td>
<td>12m</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>2 / 5</td>
<td>1.5hrs</td>
<td>12m</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>2 / 5</td>
<td>4hrs</td>
<td>2m</td>
</tr>
<tr>
<td>Array findi</td>
<td>3 / 7</td>
<td>1.5hrs</td>
<td>9m</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>2 / 3</td>
<td>1hr</td>
<td>3m</td>
</tr>
<tr>
<td>Array foldi</td>
<td>0 / 1</td>
<td>15m</td>
<td>-</td>
</tr>
<tr>
<td>Array partition</td>
<td>3 / 3</td>
<td>2.5hrs</td>
<td>5m</td>
</tr>
<tr>
<td>Stack filter</td>
<td>3 / 3</td>
<td>1.5hrs</td>
<td>11m</td>
</tr>
<tr>
<td>Stack reverse</td>
<td>1 / 1</td>
<td>2hrs</td>
<td>30s</td>
</tr>
<tr>
<td>SLL partition</td>
<td>0 / 2</td>
<td>2hrs</td>
<td>-</td>
</tr>
<tr>
<td>SLL of array</td>
<td>0 / 1</td>
<td>2hrs</td>
<td>-</td>
</tr>
</tbody>
</table>
## RQ1: Effectiveness of proof repair

<table>
<thead>
<tr>
<th>Name</th>
<th># Admits / # Obligations</th>
<th>Time (old)</th>
<th>Time (new)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq to array</td>
<td>3 / 5</td>
<td>2hrs</td>
<td>17m</td>
</tr>
<tr>
<td>Make rev list</td>
<td>0 / 2</td>
<td>10m</td>
<td>-</td>
</tr>
<tr>
<td>Tree to array</td>
<td>2 / 4</td>
<td>5hrs</td>
<td>18m</td>
</tr>
<tr>
<td>Array exists</td>
<td>2 / 4</td>
<td>30m</td>
<td>12m</td>
</tr>
<tr>
<td>Array find mapi</td>
<td>2 / 5</td>
<td>1.5hrs</td>
<td>12m</td>
</tr>
<tr>
<td>Array is sorted</td>
<td>2 / 5</td>
<td>4hrs</td>
<td>2m</td>
</tr>
<tr>
<td>Array findi</td>
<td>3 / 7</td>
<td>1.5hrs</td>
<td>9m</td>
</tr>
<tr>
<td>Array of rev list</td>
<td>2 / 3</td>
<td>1hr</td>
<td>3m</td>
</tr>
<tr>
<td>Array foldi</td>
<td>0 / 1</td>
<td>15m</td>
<td>-</td>
</tr>
<tr>
<td>Array partition</td>
<td>3 / 3</td>
<td>2.5hrs</td>
<td>5m</td>
</tr>
<tr>
<td>Stack filter</td>
<td>3 / 3</td>
<td>1.5hrs</td>
<td>11m</td>
</tr>
<tr>
<td>Stack reverse</td>
<td>1 / 1</td>
<td>2hrs</td>
<td>30s</td>
</tr>
<tr>
<td>SLL partition</td>
<td>0 / 2</td>
<td>2hrs</td>
<td>-</td>
</tr>
<tr>
<td>SLL of array</td>
<td>0 / 1</td>
<td>2hrs</td>
<td>-</td>
</tr>
</tbody>
</table>
RQ2: Efficiency of proof repair

![Bar chart showing the time taken for different proof repair tasks. The chart includes tasks such as Seq to array, Tree to array, Array exists, Array find mapi, Array is sorted, Array find i, Array of rev list, Array foldi, Array partition, Stack filter, Stack reverse, Sll partition, Sll of array. The y-axis represents the time taken in seconds, ranging from 0 to 400. The x-axis lists the tasks. The chart highlights that some tasks take significantly longer than 2 minutes.](image-url)
## RQ2: Efficiency of proof repair

<table>
<thead>
<tr>
<th>Example</th>
<th>Time (s)</th>
<th></th>
<th></th>
<th></th>
<th>Total (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Extraction</td>
<td>Testing</td>
<td>Remaining</td>
<td></td>
</tr>
<tr>
<td>seq_to_array</td>
<td>28.57</td>
<td>1.95</td>
<td>20.36</td>
<td>5.28</td>
<td>58</td>
</tr>
<tr>
<td>make_rev_list</td>
<td>≤ 10 ms</td>
<td>3.36</td>
<td>≤ 10 ms</td>
<td>11.95</td>
<td>15</td>
</tr>
<tr>
<td>tree_to_array</td>
<td>6.75</td>
<td>1.95</td>
<td>2.98</td>
<td>13.32</td>
<td>25</td>
</tr>
<tr>
<td>array_exists</td>
<td>≤ 10 ms</td>
<td>3.30</td>
<td>≤ 10 ms</td>
<td>13.23</td>
<td>17</td>
</tr>
<tr>
<td>array_find_mapi</td>
<td>≤ 10 ms</td>
<td>2.13</td>
<td>≤ 10 ms</td>
<td>13.95</td>
<td>17</td>
</tr>
<tr>
<td>array_is_sorted</td>
<td>≤ 10 ms</td>
<td>2.04</td>
<td>≤ 10 ms</td>
<td>15.38</td>
<td>18</td>
</tr>
<tr>
<td>array_findi</td>
<td>≤ 10 ms</td>
<td>2.13</td>
<td>≤ 10 ms</td>
<td>19.07</td>
<td>22</td>
</tr>
<tr>
<td>array_of_rev_list</td>
<td>1.72</td>
<td>2.82</td>
<td>0.96</td>
<td>15.62</td>
<td>21</td>
</tr>
<tr>
<td>array_foldi</td>
<td>≤ 10 ms</td>
<td>488.89</td>
<td>≤ 10 ms</td>
<td>15.00</td>
<td>504</td>
</tr>
<tr>
<td>array_partition</td>
<td>3.51</td>
<td>69.73</td>
<td>2.62</td>
<td>17.53</td>
<td>95</td>
</tr>
<tr>
<td>stack_filter</td>
<td>≤ 10 ms</td>
<td>81.88</td>
<td>≤ 10 ms</td>
<td>21.53</td>
<td>104</td>
</tr>
<tr>
<td>stack_reverse</td>
<td>≤ 10 ms</td>
<td>88.42</td>
<td>≤ 10 ms</td>
<td>16.94</td>
<td>105</td>
</tr>
<tr>
<td>sll_partition</td>
<td>≤ 10 ms</td>
<td>426.62</td>
<td>≤ 10 ms</td>
<td>16.43</td>
<td>443</td>
</tr>
<tr>
<td>sll_of_array</td>
<td>0.02</td>
<td>55.98</td>
<td>0.01</td>
<td>13.33</td>
<td>69</td>
</tr>
</tbody>
</table>
## RQ2: Efficiency of proof repair

<table>
<thead>
<tr>
<th>Example</th>
<th>Time (s)</th>
<th></th>
<th></th>
<th></th>
<th>Total (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Extraction</td>
<td>Testing</td>
<td>Remaining</td>
<td></td>
</tr>
<tr>
<td>seq_to_array</td>
<td>28.57</td>
<td>1.95</td>
<td>20.36</td>
<td>5.28</td>
<td>58</td>
</tr>
<tr>
<td>make_rev_list</td>
<td>(\leq 10ms)</td>
<td>3.36</td>
<td>(\leq 10ms)</td>
<td>11.95</td>
<td>15</td>
</tr>
<tr>
<td>tree_to_array</td>
<td>6.75</td>
<td>1.95</td>
<td>2.98</td>
<td>13.32</td>
<td>25</td>
</tr>
<tr>
<td>array_exists</td>
<td>(\leq 10ms)</td>
<td>3.30</td>
<td>(\leq 10ms)</td>
<td>13.23</td>
<td>17</td>
</tr>
<tr>
<td>array_find_mapi</td>
<td>(\leq 10ms)</td>
<td>2.13</td>
<td>(\leq 10ms)</td>
<td>13.95</td>
<td>17</td>
</tr>
<tr>
<td>array_is_sorted</td>
<td>(\leq 10ms)</td>
<td>2.04</td>
<td>(\leq 10ms)</td>
<td>15.38</td>
<td>18</td>
</tr>
<tr>
<td>array_findi</td>
<td>(\leq 10ms)</td>
<td>2.13</td>
<td>(\leq 10ms)</td>
<td>19.07</td>
<td>22</td>
</tr>
<tr>
<td>array_of_rev_list</td>
<td>1.72</td>
<td>2.82</td>
<td>0.96</td>
<td>15.62</td>
<td>21</td>
</tr>
<tr>
<td>array_foldi</td>
<td>(\leq 10ms)</td>
<td>488.89</td>
<td>(\leq 10ms)</td>
<td>15.00</td>
<td>504</td>
</tr>
<tr>
<td>array_partition</td>
<td>3.51</td>
<td>69.73</td>
<td>2.62</td>
<td>17.53</td>
<td>95</td>
</tr>
<tr>
<td>stack_filter</td>
<td>(\leq 10ms)</td>
<td>81.88</td>
<td>(\leq 10ms)</td>
<td>21.53</td>
<td>104</td>
</tr>
<tr>
<td>stack_reverse</td>
<td>(\leq 10ms)</td>
<td>88.42</td>
<td>(\leq 10ms)</td>
<td>16.94</td>
<td>105</td>
</tr>
<tr>
<td>sll_partition</td>
<td>(\leq 10ms)</td>
<td>426.62</td>
<td>(\leq 10ms)</td>
<td>16.43</td>
<td>443</td>
</tr>
<tr>
<td>sll_of_array</td>
<td>0.02</td>
<td>55.98</td>
<td>0.01</td>
<td>13.33</td>
<td>69</td>
</tr>
</tbody>
</table>
## RQ2: Efficiency of proof repair

<table>
<thead>
<tr>
<th>Example</th>
<th>Time (s)</th>
<th>Total (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generation</td>
<td>Extraction</td>
</tr>
<tr>
<td>seq_to_array</td>
<td>28.57</td>
<td>1.95</td>
</tr>
<tr>
<td>make_rev_list</td>
<td>≤ 10ms</td>
<td>3.36</td>
</tr>
<tr>
<td>tree_to_array</td>
<td>6.75</td>
<td>1.95</td>
</tr>
<tr>
<td>array_exists</td>
<td>≤ 10ms</td>
<td>3.30</td>
</tr>
<tr>
<td>array_find_mapi</td>
<td>≤ 10ms</td>
<td>2.13</td>
</tr>
<tr>
<td>array_is_sorted</td>
<td>≤ 10ms</td>
<td>2.04</td>
</tr>
<tr>
<td>array_findi</td>
<td>≤ 10ms</td>
<td>2.13</td>
</tr>
<tr>
<td>array_of_rev_list</td>
<td>1.72</td>
<td>2.82</td>
</tr>
<tr>
<td>array_foldi</td>
<td>≤ 10ms</td>
<td>488.89</td>
</tr>
<tr>
<td>array_partition</td>
<td>3.51</td>
<td>69.73</td>
</tr>
<tr>
<td>stack_filter</td>
<td>≤ 10ms</td>
<td>81.88</td>
</tr>
<tr>
<td>stack_reverse</td>
<td>≤ 10ms</td>
<td>88.42</td>
</tr>
<tr>
<td>sll_partition</td>
<td>≤ 10ms</td>
<td>426.62</td>
</tr>
<tr>
<td>sll_of_array</td>
<td>0.02</td>
<td>55.98</td>
</tr>
</tbody>
</table>
Mostly Automated Proof Repair for Verified Libraries

To Take Away

1. Building blocks for new proof found in old proof

2. Picking a correct invariant is hard!

3. A new take on Curry-Howard: Proof-Driven Testing
To Take Away

1. Building blocks for new proof found in old proof

2. Picking a correct invariant is hard!

3. A new take on Curry-Howard: Proof-Driven Testing

Thanks!
Mostly Automated Proof Repair for Verified Libraries

To Take Away

1. Building blocks for new proof found in old proof

2. Picking a correct invariant is hard!

3. A new take on Curry-Howard: Proof-Driven Testing

Thanks!
**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

  $$\text{filter } p (\text{filter } p \ell) = \text{filter } p \ell$$

  not present in original proof.
RQ3: Failure Modes

```ocaml
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...

```ocaml
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...

```plaintext
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.
```