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

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

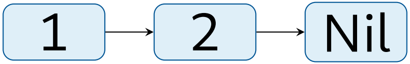
OCaml's `'a Seq.t` datatype

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

A *thunked* tail

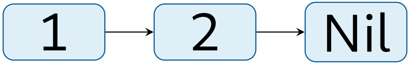
OCaml’s `'a Seq.t` datatype

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



OCaml's `'a Seq.t` datatype

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



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

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’s write
some code!

```
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 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.

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

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

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

Q: Convert a sequence to an array.

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

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

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

Q: Convert a sequence to an array.

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

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

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

Q: Convert a sequence to an array.

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

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

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

Q: Convert a sequence to an array.

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

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

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

“a” **points-to** an array

Q: Convert a sequence to an array.

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

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

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

Let's write
some 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
    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.
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 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 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 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 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 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 $[| \; h \; tl]$
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec $(\lambda t \to$
    $a \mapsto$ Array (
    $t \mathbin{++} \mathit{drop}\,(\mathit{length}\; t)$
      $(\mathit{make}\,(\mathit{length}\;\ell)\; h))$
).
  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 $(\lambda t \to$

$a \mapsto$ Array (

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

$(\mathit{make}\,(\mathit{length}\,\ell)\,h))$

).

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 $\ell$ as [| $h$ $tl$]
   – xvalemptyarr.
   –
    xapp.
    xalloc.
    xapp (iteri_spec ($\lambda t \to$
     $a \mapsto$ Array (
     $t$ $++$ $drop\ (length\ t)$
       $(make\ (length\ \ell)\ h)$)
    ).
   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 $\ell$ as [| $h$ $tl$]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec ($\lambda t \to$
    $a \mapsto$ Array (
     $t \mathbin{++} drop\ (length\ t)$
       $(make\ (length\ \ell)\ h))$
   ).
  xval.
```

```
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.
```

```
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.
```

```
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 $\ell$ as [| $h$ $tl$]
  – xvalemptyarr.
  –
   xapp.
   xalloc.
   xapp (iteri_spec ($\lambda t \to$
    $a \mapsto$ Array (
    $t \mathbin{++} drop\ (length\ t)$
      $(make\ (length\ \ell)\ h))$
   ).
  xval.
```

```
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.
```

prefix

```
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.
```

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

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

  | 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]

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

```
let to_array s =

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

        a.(i) <- vl

    ) s;
    a
```

```
xcf.

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

$$a \mapsto \mathrm{Array}(t \mathbin{++} \mathit{drop}\,(\mathit{length}\;t)\;(\mathit{make}\,(\mathit{length}\;\ell)\;h))$$

```
let to_array s =

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

    ) s;
    a
```

$$
\begin{array}{l}
\text{xcf.} \\
\\
\quad - \\
\quad \text{xapp.} \\
\quad \text{xalloc.} \\
\quad \text{xapp (iteri\_spec } (\lambda t \to \\
\qquad a \mapsto \text{Array (} \\
\qquad\ t \mathbin{++} \mathit{drop}\ (\mathit{length}\ t) \\
\qquad\quad (\mathit{make}\ (\mathit{length}\ \ell)\ h)) \\
\quad \text{)).} \\
\text{xval.}
\end{array}
$$

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

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

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

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

```
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
```

```
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.
```

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

       a.(i) <- vl

    ) s;
    a
```

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

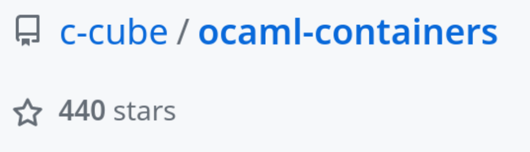
Writing verified code is hard

Writing verified code is hard

Writing verified code is hard…

A problem arises…

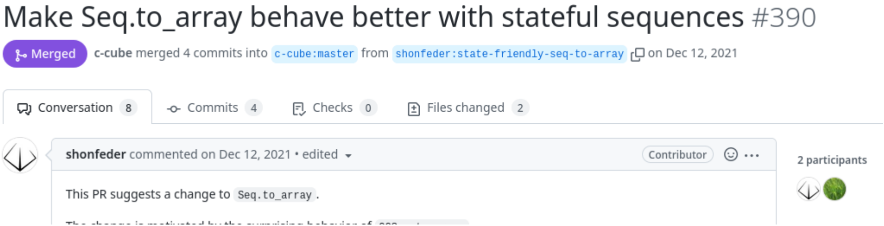
A problem arises...



c-cube / **ocaml-containers**

440 stars

A problem arises…



Make Seq.to_array behave better with stateful sequences #390

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

shonfeder commented on Dec 12, 2021 • edited

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

The change is motivated by the surprising behavior of

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Old

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

New

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

Completely different implementation...

Old

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

New

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

Completely different implementation…

…proof must be redone.

Writing verified code is hard...

Maintaining

~~Writing~~ verified code is hard…

Maintaining

~~Writing~~ verified code is hard...

Proof Repair

Maintaining ~~Writing~~ verified code is hard...

Proof Repair

Ringer (2021)

Maintaining

~~Writing~~ verified code is hard...

...can we make it easier?

Maintaining ~~Writing~~ verified code is hard…

…can we make it easier?

Yes!

This Work

Sisyphus

Mostly automated *proof-repair* for OCaml programs

Outline

1. Motivation

2. Key Challenges & Solutions

3. Evaluation

Outline

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

New Program:

```
let to_array s =

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

New Program:

```
let to_array s =

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

How to generate a proof script?

New Program:

```
let to_array s =

  let sz, ls = fold_left

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

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

How to generate a proof script?

Observation: proofs are syntax-directed

```
let to_array s =

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

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

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

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

xcf.

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

xcf.

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

xcf.

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

```
xcf.

xapp (...).
```

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

```
xcf.

xapp (...).
```

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

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

```
xcf.

xapp (...).

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

                     idx rest
    (fun acc t =>
     (??))).

    xvals.
```

How to fill in holes?

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—

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

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

Generating candidate invariants

Programs are different—

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

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

—but have similarities.

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

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

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

Generating candidate invariants

```
let to_array l =

  match l () with

  | Nil          -> [| |]

  | Cons (x, _) ->

      let len = length' l in

      let a = make len x in

      iteri

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

        l;

       a
```

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

Generating candidate invariants

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

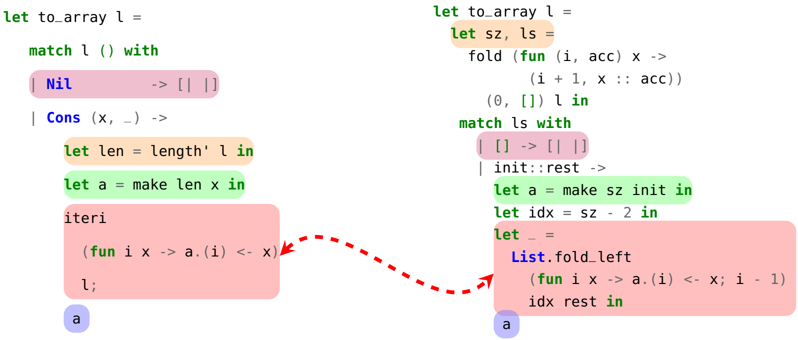
```
let to_array l =
  let sz, ls =
    fold (fun (i, acc) x ->
           (i + 1, x :: acc))
      (0, []) l in
   match ls with
     | [] -> [| |]
     | init::rest ->
       let a = make sz init in
       let idx = sz - 2 in
       let _ =
         List.fold_left
           (fun i x -> a.(i) <- x;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

Generating candidate invariants

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

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

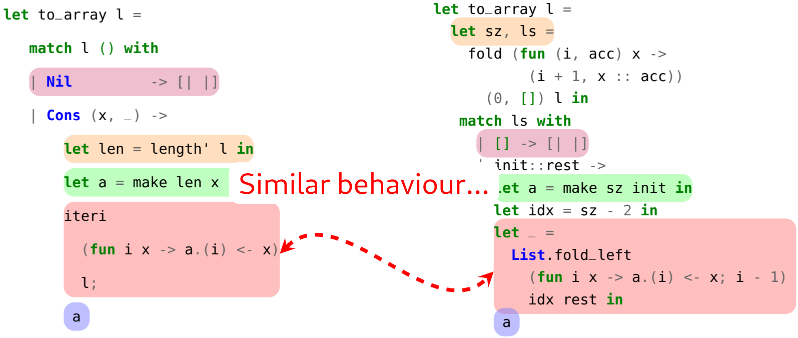
Generating candidate invariants



```
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
    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 ->
      et 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
    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 ->
     et 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?

Generating candidate invariants

Q: How to discover these similarities automatically?

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

..similar invariants?

Generating candidate invariants

Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

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

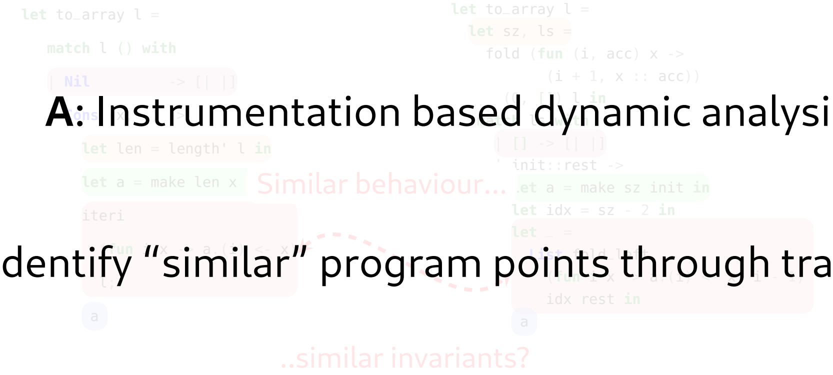
Similar behaviour...

```
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
```

..similar invariants?

Generating candidate invariants

Q: How to discover these similarities automatically?



A: Instrumentation based dynamic analysis

Identify “similar” program points through traces.

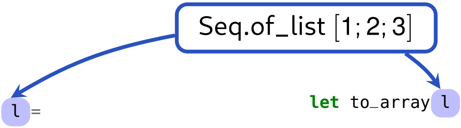
```
let to_array l =

   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
```

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
```

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
```

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
```

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]

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]

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]

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]

3

[|1; 1; 1|]

```
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]

3

[|1; 1; 1|]

```
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]

3

[|1; 1; 1|]

```
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
```

3

[|1; 1; 1|]

```
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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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]

3

[|1; 1; 1|]

[|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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 3|]

[|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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 3|]

[|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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 3|]

[|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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

[|3; 3; 3|]

[|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
```

3

[|1; 1; 1|]

[|1; 2; 3|]

```
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
```

3

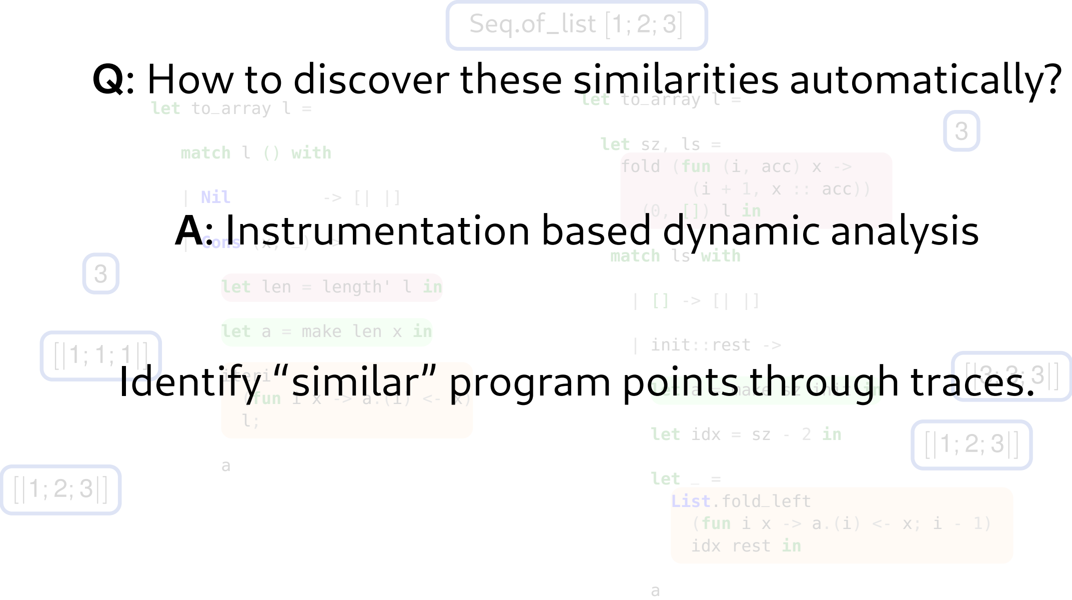
[|3; 3; 3|]

[|1; 2; 3|]

Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

Identify "similar" program points through traces.

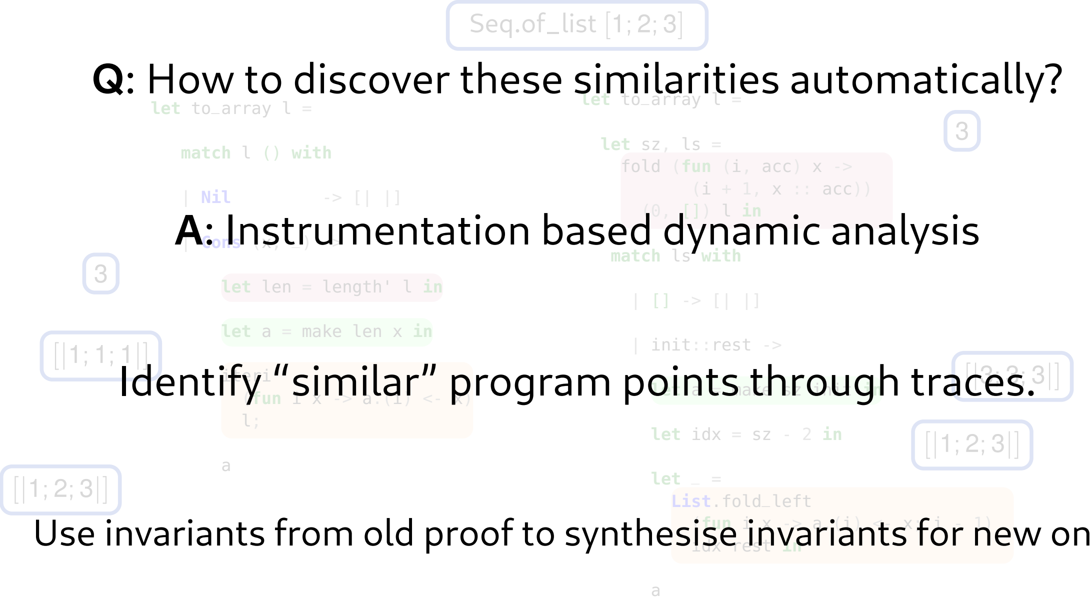


Q: How to discover these similarities automatically?

A: Instrumentation based dynamic analysis

Identify “similar” program points through traces.

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



```
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
```

Old Invariant:

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

Old Invariant:

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

New Invariant:

Old Invariant:

$$a \mapsto \texttt{Array}(t \mathbin{++} \mathit{drop}\ (\mathit{length}\ t)\ (\mathit{make}\ (\mathit{length}\ \ell)\ h))$$

New Invariant:

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

(1)

Old Invariant:

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

New Invariant:

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

(1)

Old Invariant:

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

New Invariant:

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

(1)

• • •

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

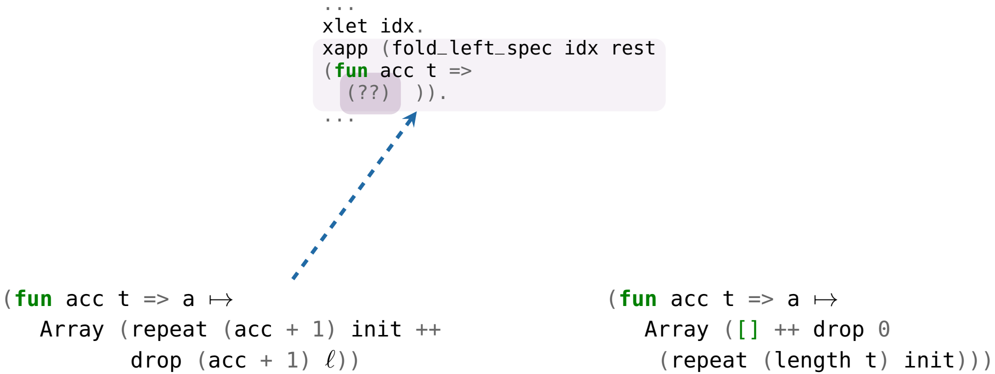
Choosing an Invariant

```
...
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



```
...
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

```
...
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

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

Validate using SMT?

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

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

Validate using SMT?

Which one?

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 =>
  (??)  )).
...
```

If only we could *test*...

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

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

Depends on
logical parameters!

If only we could *test*...

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

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

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

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

Verified Fold Left

```
let 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

```
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

```
let fold_left f acc ls t l =

  match ls with
  | [] ->

    acc
  | h :: tl ->

    let acc' = f acc h in

    fold_left f acc' tl
```

Verified Fold Left

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

    acc
  | h :: tl ->

    let acc' = f acc h in

    fold_left f acc' tl
```

Verified Fold Left

```
let fold_left f acc ls t l =
  {l  acc  t}
  match ls with
  | [] ->
    {l  acc  t}
    acc
  | h :: tl ->

    let acc' = f acc h in

    fold_left f acc' tl
```

Verified Fold Left

```
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

```
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
```

Verified Fold Left

```
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

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

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

Verified Fold Left

```
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

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

Verified Fold Left

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

Proof-Driven Testing

Proof-Driven Testing

`fold_left_spec`

Proof-Driven Testing

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

Proof-Driven Testing

Instantiate with concrete arguments..

$\texttt{fold\_left\_spec}\ ?l\ \boxed{f\ \ 2\ \ [2;1]}$

Proof-Driven Testing

Instantiate with concrete arguments..

`fold_left_spec` $?l\ f\ 2\ [2; 1]$

Proof-Driven Testing

Instantiate with concrete arguments..

`fold_left_spec` $?l \; f \; 2 \; [2;1]$

...with existentials for proof arguments

Proof-Driven Testing

`fold_left_spec` $?l \ f \ 2 \ [2; 1]$

Proof-Driven Testing

`fold_left_spec` $?l\ \ f\ \ 2\ \ [2; 1]$

Proof-Driven Testing

`fold_left_spec` $?l\ \ f\ \ 2\ \ [2;1]$

reduce proof term

↓

Proof-Driven Testing

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

Proof-Driven Testing

```
let fold_left f acc ls t =
  {I 2 []}
  match ls with
  | [] ->
    {I 2 []}
    acc
  | 2 :: [1] ->
    {I 2 []}
    let acc' = f 2 2 in
    {I acc' [2]}
    fold_left f acc' tl
    {I acc'' [2, 1]}
```

Proof-Driven Testing

`fold_left_spec` $?l\ f\ 2\ [2;1]$

|

reduce proof term

↓

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

Proof-Driven Testing

`fold_left_spec` $?l\ f\ 2\ [2;1]$

reduce proof term

↓

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

custom proof extraction

↓

Proof-Driven Testing

```
let fold_left f acc ls =
  {l 2 []}
  match ls with
  | [] ->
    {l 2 []}
    acc
  | 2 :: [1] ->
    {l 2 []}
    let acc' = f 2 2 in
    {l acc' [2]}
    fold_left f acc' tl
    {l 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

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

Simulates concrete run of `List.fold_left`

Proof-Driven Testing

Instantiate I with embedding of candidate invariant...

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

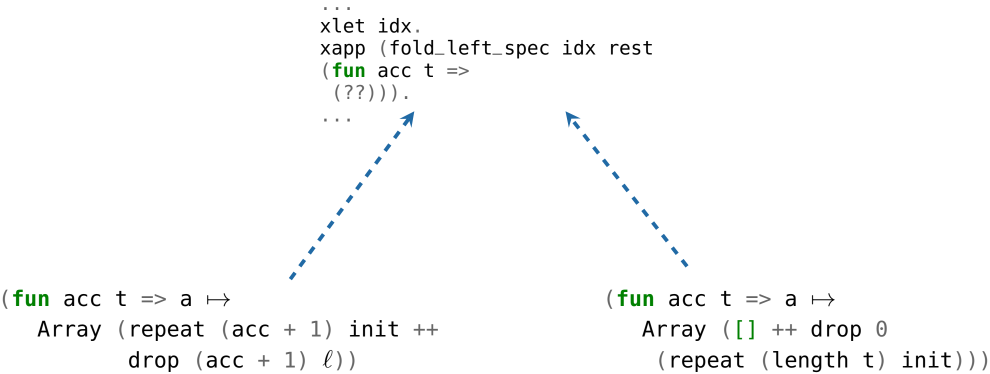
Proof-Driven Testing

Instantiate I with embedding of candidate invariant...

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

...prune candidate if assertion raised.

Testing Candidate Invariants

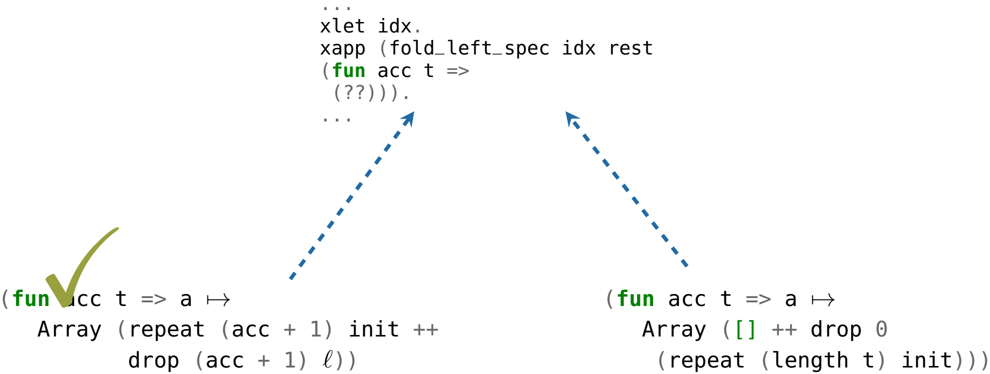


```
...
xlet idx.
xapp (fold_left_spec idx rest
(fun acc t =>
 (??))).
...
```

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

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

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 ([] ++ drop 0
    (repeat (length t) init)))
```

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 ([] ++ drop 0
    (repeat (length t) init)))
```

Proof-Driven Testing

Proofs are proofs

Proofs are programs

Proofs are tests

Proof are *programs*…

Proof-Driven Testing

Proof are *programs*...

but, what do they *compute*?

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.

Proof-Driven Testing

Proofs are proofs

Proofs are programs

Proofs are tests

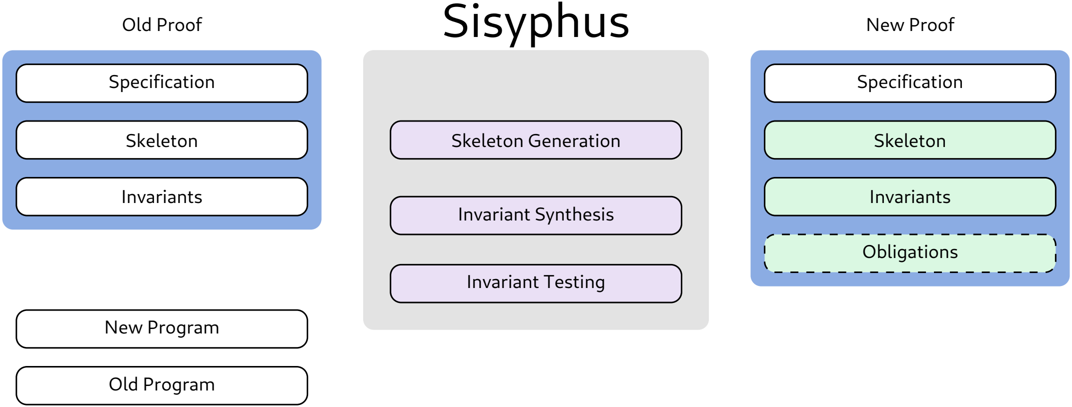
Proof are *programs*...

but, what do they *compute*?

Curry Howard: They establish logical facts.

PDT: HO-proofs describe tests!

Sisyphus



Old Proof

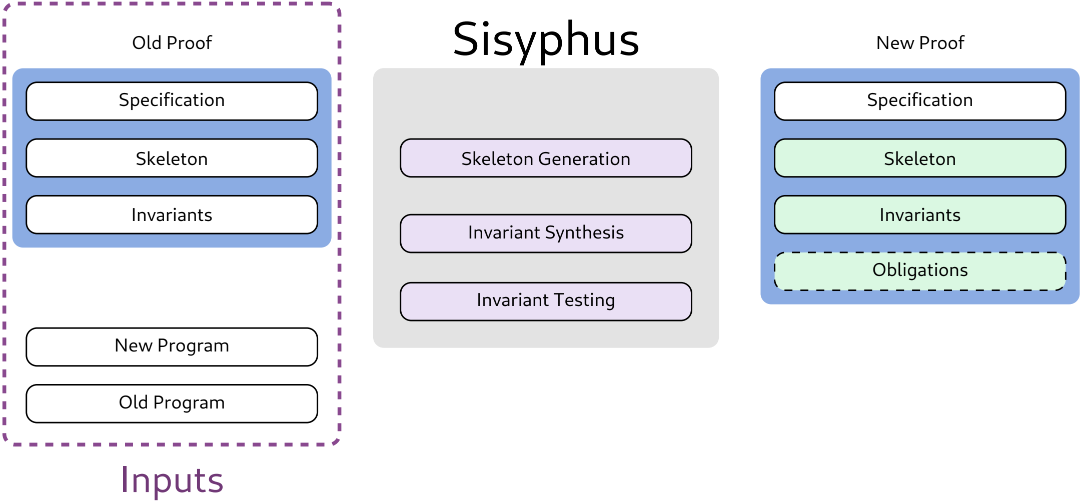
- Specification
- Skeleton
- Invariants

- New Program
- Old Program

- Skeleton Generation
- Invariant Synthesis
- Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations



Old Proof

Specification

Skeleton

Invariants

New Program

Old Program

Inputs

Sisyphus

Skeleton Generation

Invariant Synthesis

Invariant Testing

New Proof

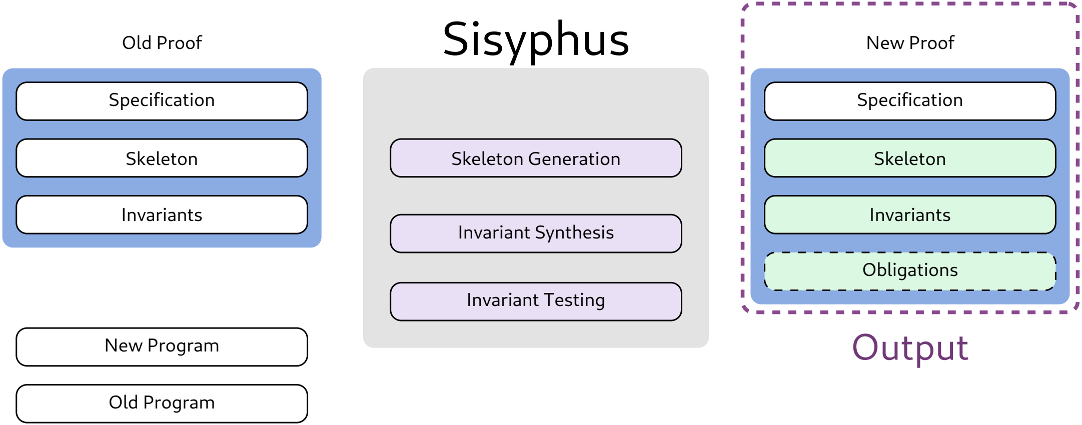
Specification

Skeleton

Invariants

Obligations

Sisyphus



Old Proof

- Specification
- Skeleton
- Invariants

New Program

Old Program

Skeleton Generation

Invariant Synthesis

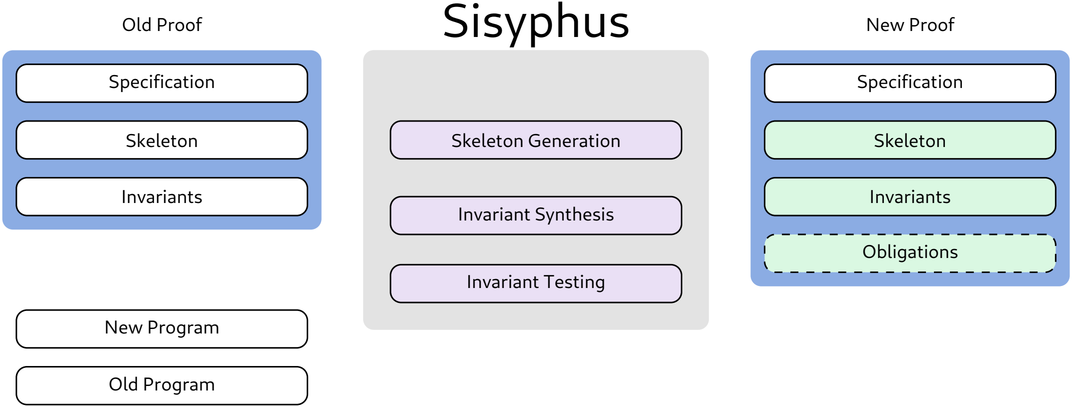
Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations

Output

Sisyphus



Old Proof

Specification

Skeleton

Invariants

New Program

Old Program

Skeleton Generation

Invariant Synthesis

Invariant Testing

New Proof

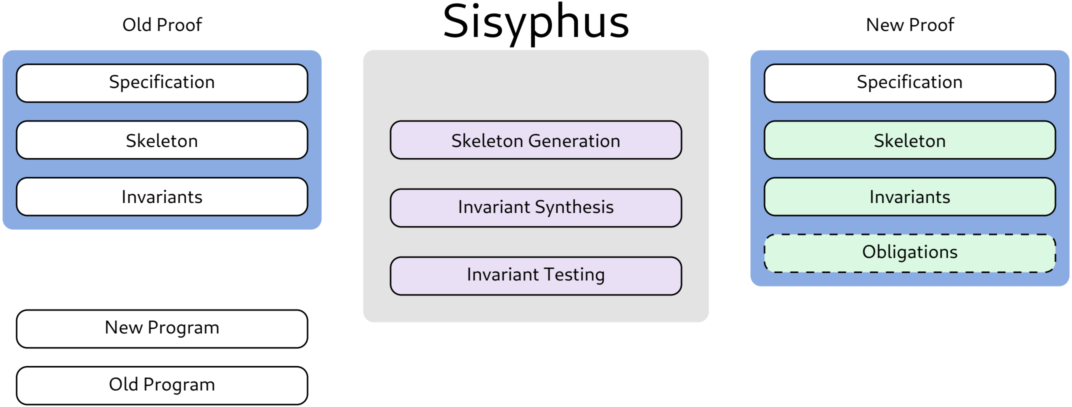
Specification

Skeleton

Invariants

Obligations

Sisyphus



Old Proof

- Specification
- Skeleton
- Invariants

New Program

Old Program

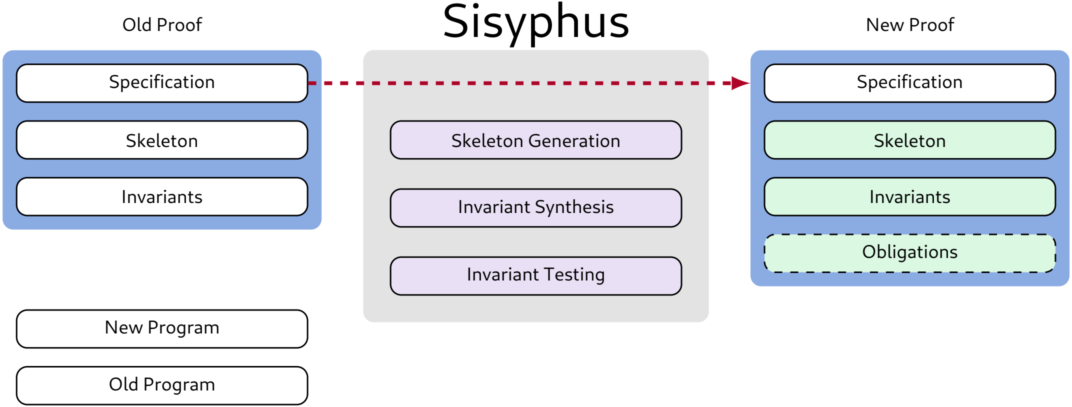
Sisyphus

- Skeleton Generation
- Invariant Synthesis
- Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations

Sisyphus



Old Proof

- Specification
- Skeleton
- Invariants

New Program

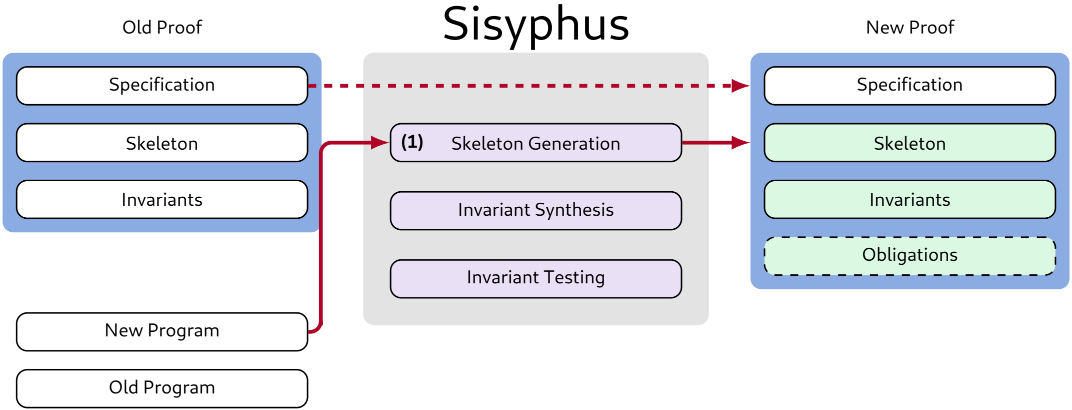
Old Program

Sisyphus

- Skeleton Generation
- Invariant Synthesis
- Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations



Sisyphus

Old Proof

- Specification
- Skeleton
- Invariants

New Program

Old Program

(1) Skeleton Generation

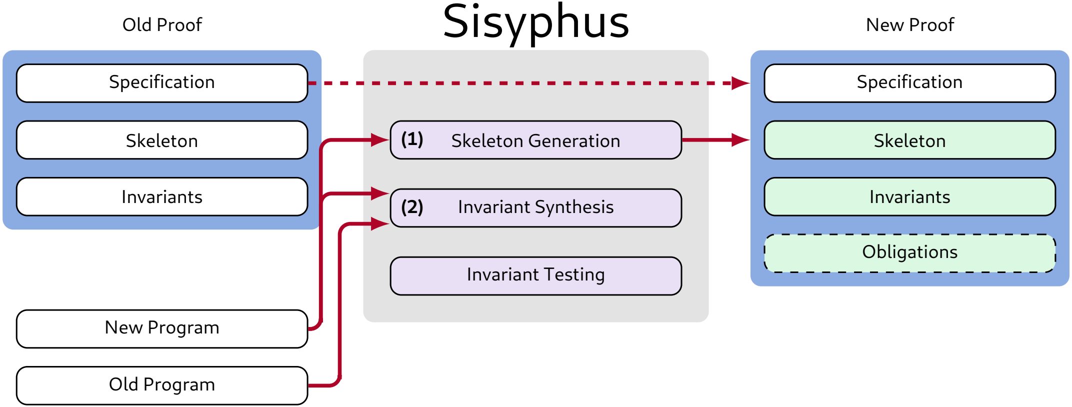
Invariant Synthesis

Invariant Testing

New Proof

- Specification
- Skeleton
- Invariants
- Obligations

Sisyphus



Old Proof

Specification

Skeleton

Invariants

New Program

Old Program

(1) Skeleton Generation

(2) Invariant Synthesis

Invariant Testing

New Proof

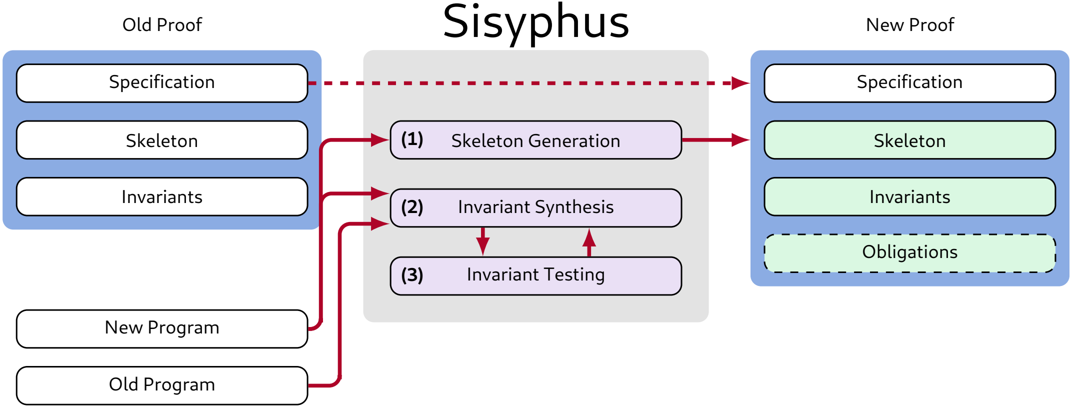
Specification

Skeleton

Invariants

Obligations

Sisyphus



Old Proof

Specification

Skeleton

Invariants

New Program

Old Program

(1) Skeleton Generation

(2) Invariant Synthesis

(3) Invariant Testing

New Proof

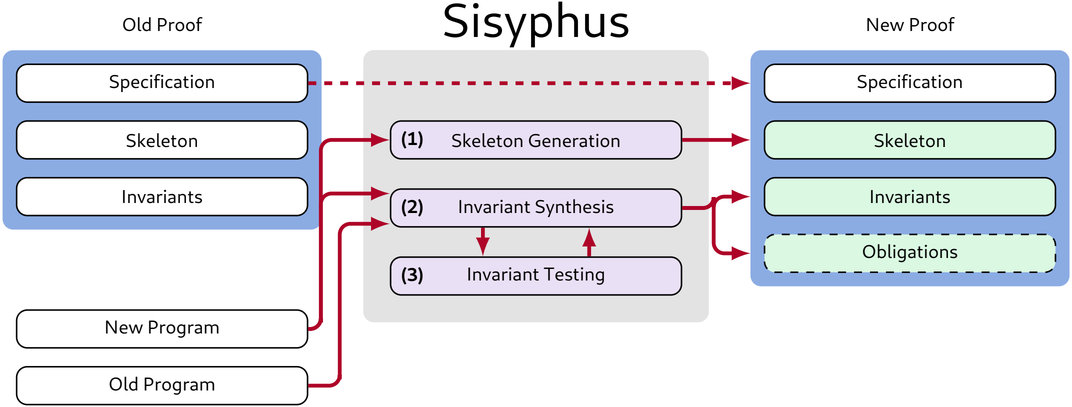
Specification

Skeleton

Invariants

Obligations

Sisyphus



Old Proof

Specification

Skeleton

Invariants

New Program

Old Program

(1) Skeleton Generation

(2) Invariant Synthesis

(3) Invariant Testing

New Proof

Specification

Skeleton

Invariants

Obligations

Outline

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

Outline

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

Pragmatic Concerns

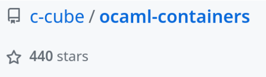
① Is Sisyphus effective at repairing proofs?

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`

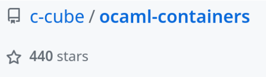
Benchmark Programs


c-cube / **ocaml-containers**

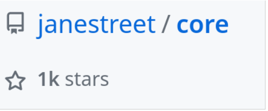
☆ 440 stars

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

Benchmark Programs


c-cube / **ocaml-containers**

☆ 440 stars


janestreet / **core**

☆ 1k stars

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

Benchmark Programs

Example	Data Structure	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tree	IterOrd, DataStr
Array exists	Array	Mutable/Pure
Array find mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Example	Data Structure	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list[†]	Ref	Mutable/Pure
Tree to array[†]	Array, Tree	IterOrd, DataStr
Array exists	Array	Mutable/Pure
Array find mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter[‡]	Stack	DataStr
Stack reverse[‡]	Stack	DataStr
Sll partition[‡]	SLL	Mutable/Pure, IterOrd
Sll of array[‡]	Array, SLL	IterOrd

Benchmark Programs

Array

Example	cture	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tree	IterOrd, DataStr
Array exists	Array	Mutable/Pure
Array find mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Seq

Example	cture	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tree	IterOrd, DataStr
sts	Array	Mutable/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Seq

Example	cture	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tree	IterOrd, DataStr
sts	Array	Mutable/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

SLL

Seq

Example	cture	Refactoring
Seq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tree	IterOrd, DataStr
sts	Array	Mutable/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

SLL

Seq

Example	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list[†]	Ref	Mutable/Pure
Tree to array[†]	Array, Tree	IterOrd, DataStr
sts	Array	Mutable/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter[‡]	Stack	DataStr
Stack reverse[‡]	Stack	DataStr
Sll partition[‡]	SLL	Mutable/Pure, IterOrd
Sll of array[‡]	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

Queue

SLL

Seq

xample	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
sts	Array	ble/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

Queue

SLL

Seq

Example	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
sts	Array	table/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
Array foldi	Array	Pure/Mutable
Array partition	Array	DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

Queue

SLL

Seq

Iteration Order

Example	...cture	Refactoring
...eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
...sts	Array	...ble/Pure
...d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
A		Pure/Mutable
A		DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
Sll of array‡	Array, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

Queue

SLL

Seq

xample	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
sts	Array	le/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
A		Pure/Mutable
A		DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
	ay, SLL	IterOrd

Iteration Order

Pure to Mutable

Benchmark Programs

Tree

Array

Stack

Queue

SLL

Seq

Iteration Order

Pure to Mutable

Mutable to Pure

Example	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
sts	Array	Mutable/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
A		Pure/Mutable
A		DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
	ay, SLL	IterOrd

Benchmark Programs

Array

Stack

Tree

Queue

SLL

Seq

Data Structure

Iteration Order

Pure to Mutable

Mutable to Pure

xample	cture	Refactoring
eq to array	Array, Seq	IterOrd, DataStr
Make rev list†	Ref	Mutable/Pure
Tree to array†	Array, Tr	DataStr
sts	Array	ble/Pure
d mapi	Array, Ref	Pure/Mutable
Array is sorted	Array	Pure/Mutable
Array findi	Array	Pure/Mutable
Array of rev list	Array	DataStr
A		Pure/Mutabl
A		DataStr
Stack filter‡	Stack	DataStr
Stack reverse‡	Stack	DataStr
Sll partition‡	SLL	Mutable/Pure, IterOrd
	ay, SLL	IterOrd

RQ1: Effectiveness of proof repair

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

RQ1: Effectiveness of proof repair

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

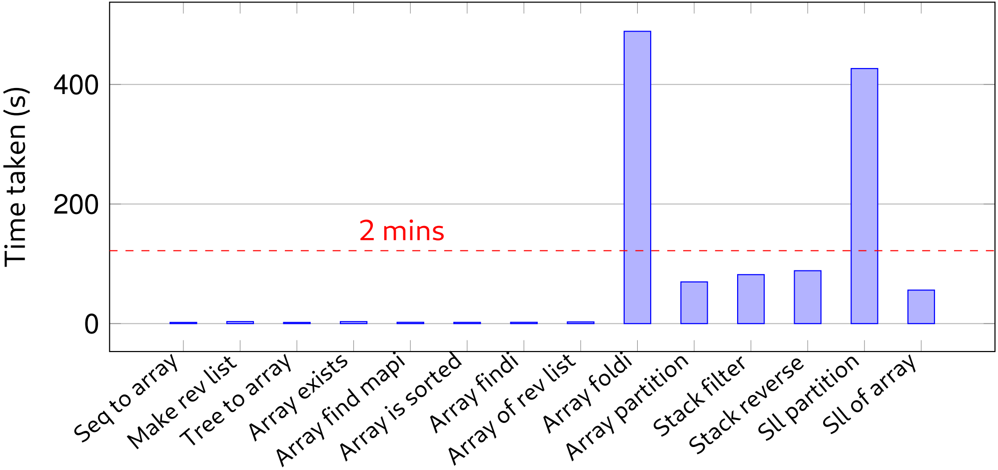
RQ1: Effectiveness of proof repair

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

RQ1: Effectiveness of proof repair

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

RQ2: Efficiency of proof repair



RQ2: Efficiency of proof repair

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

RQ2: Efficiency of proof repair

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

RQ2: Efficiency of proof repair

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

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!

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!

Backup Slides

RQ3: Failure Modes

- Repair assumes components from old proof are sufficient for new one.
- Quality of repair degrades when this fails to hold.

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

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

not present in original proof.

RQ3: Failure Modes

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

RQ3: Failure Modes

Invariant requires *flattening* operation...

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

RQ3: Failure Modes

Invariant requires *flattening* operation...

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

...not present in old proof.