



m | C | A

Automated Property-Based Testing for OCaml Modules

Ernest Ng

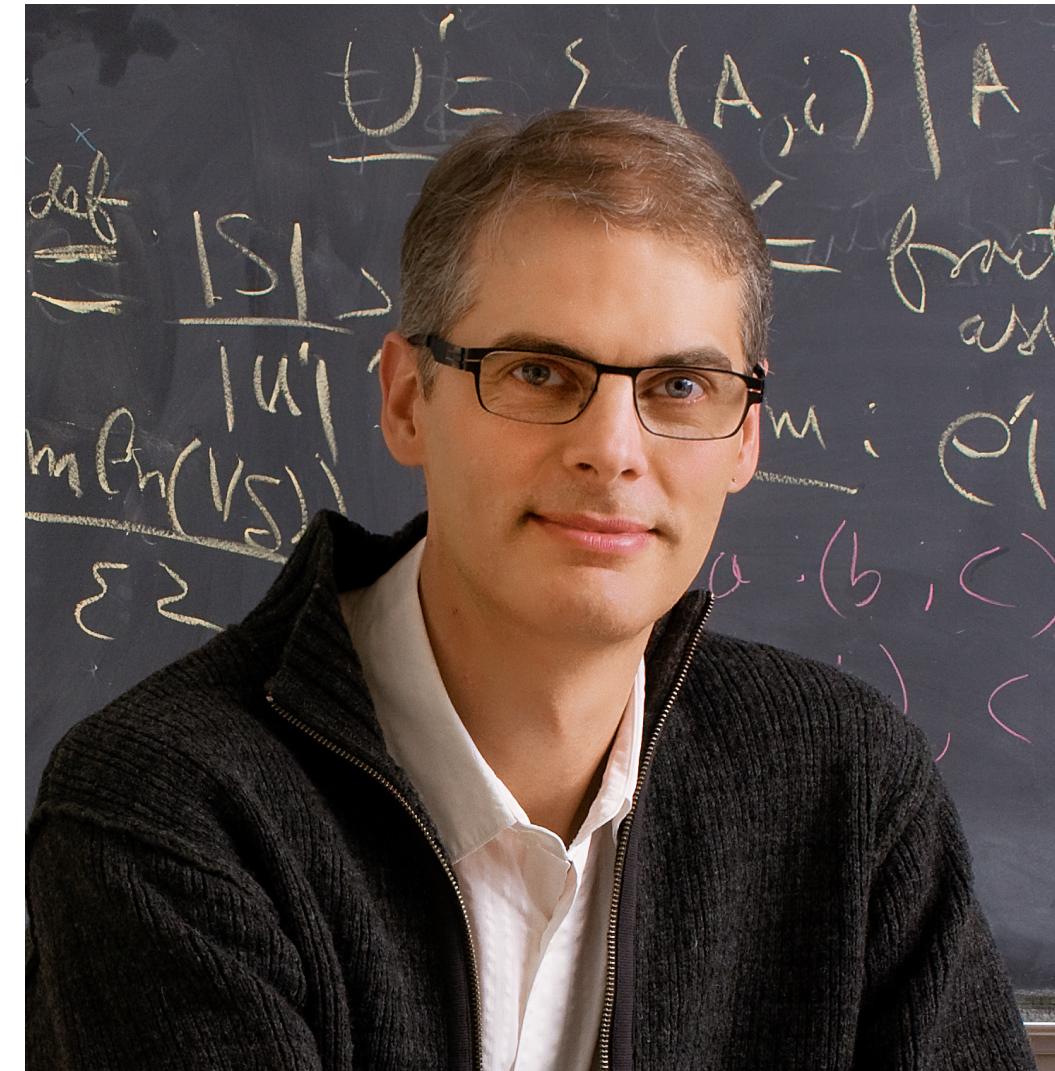
Advised by Harry Goldstein & Benjamin C. Pierce

PLClub, July 28 2023

Acknowledgements



Harry Goldstein



Benjamin C. Pierce

- Carl Eastlund (Jane Street)
- Jan Midgaard (Tarides)

Other Penn PBT projects at ICFP



Reflecting on Random Generation

HARRISON GOLDSTEIN, University of Pennsylvania, USA
SAMANTHA FROHLICH, University of Bristol, UK
MENG WANG, University of Bristol, UK
BENJAMIN C. PIERCE, University of Pennsylvania, USA

ETNA: An Evaluation Platform for Property-Based Testing (Experience Report)

JESSICA SHI, University of Pennsylvania, USA
ALPEREN KELES, University of Maryland, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Motivation

**modules
matter
most**

- Bob Harper (2011)



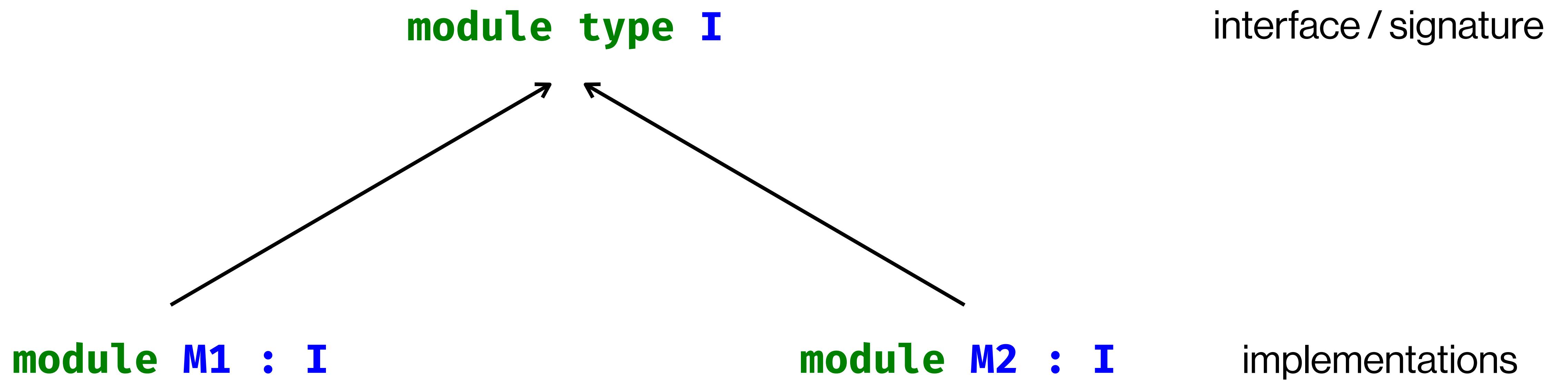
Barbara Liskov,

Programming with Abstract Data Types (1974):

- **Abstract data types** are completely characterised by operations over the ADT
- **Encapsulation**
(implementation details should be hidden from clients)



Representation independence



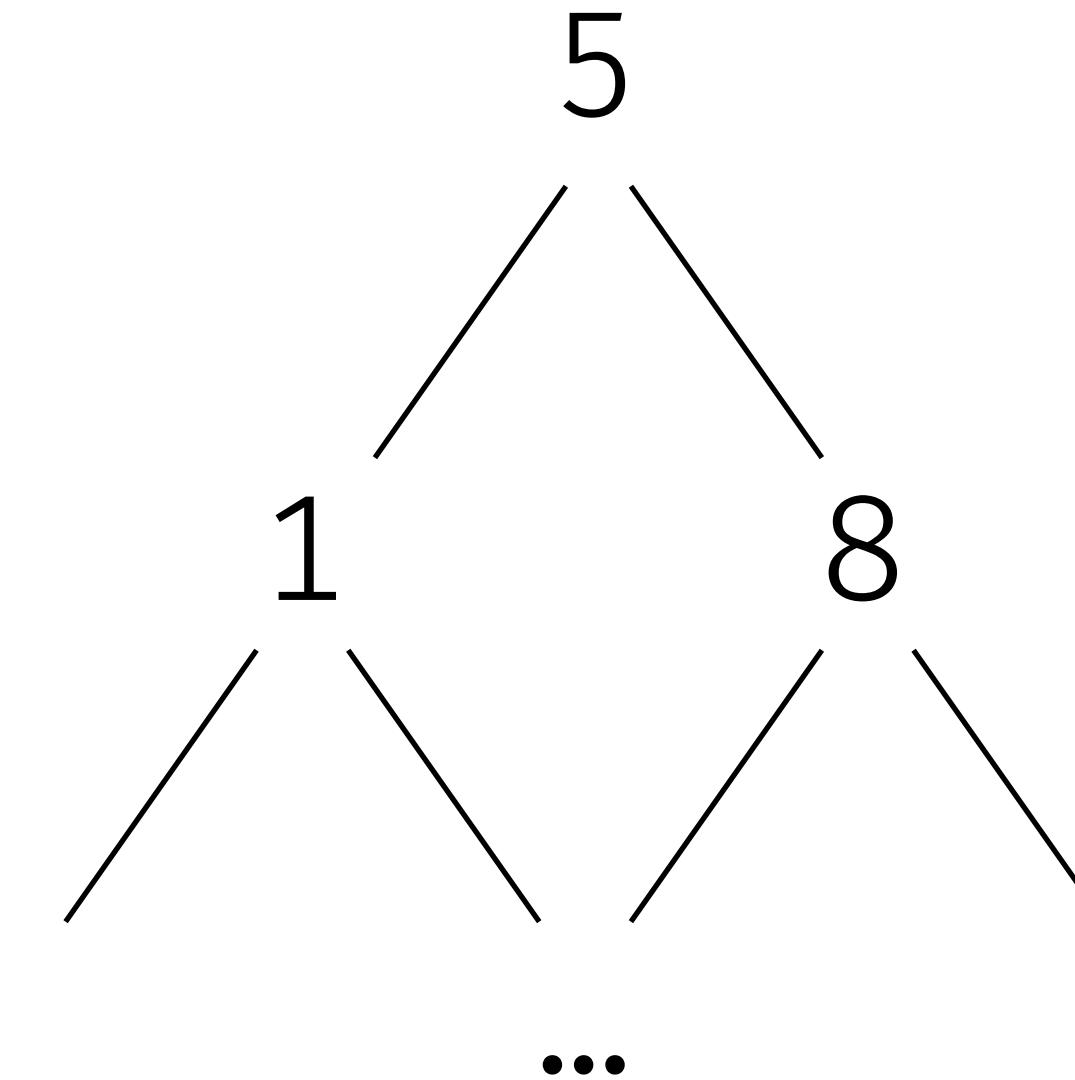
Example: Finite Sets

```
module type SetIntf = sig
  type 'a t
  val empty      : 'a t
  val add        : 'a → 'a t → 'a t
  val intersect : 'a t → 'a t → 'a t
  ...
  val invariant : 'a t → bool
end
```

$\{1, 5, 8, \dots\} \rightsquigarrow [1; 5; 8; \dots]$

```
module ListSet : Set_intf = struct
  type 'a t = 'a list
  (* No duplicates in list *)
  let invariant s = ...
  ...
end
```

{1, 5, 8, ...} \rightsquigarrow



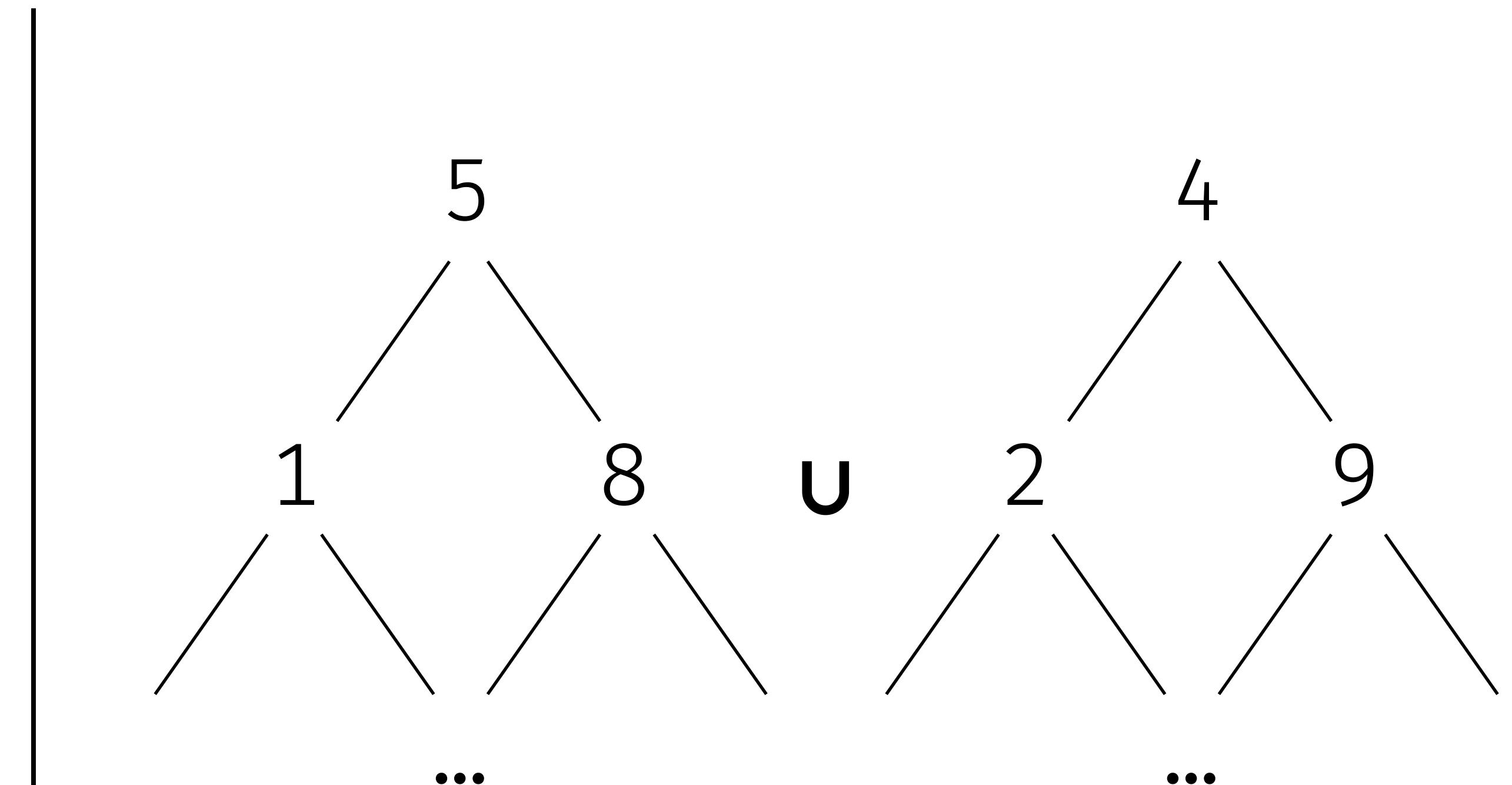
```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

```
module BSTSet : Set_intf = struct
  type 'a t = 'a tree
  (* BST invariant *)
  let invariant s = ...
  ...
end
```

Are these equivalent?

$$\{1, 5, 8\} \cup \{2, 4, 9\}$$

$$[1; 5; 8] + [2; 4; 9]$$



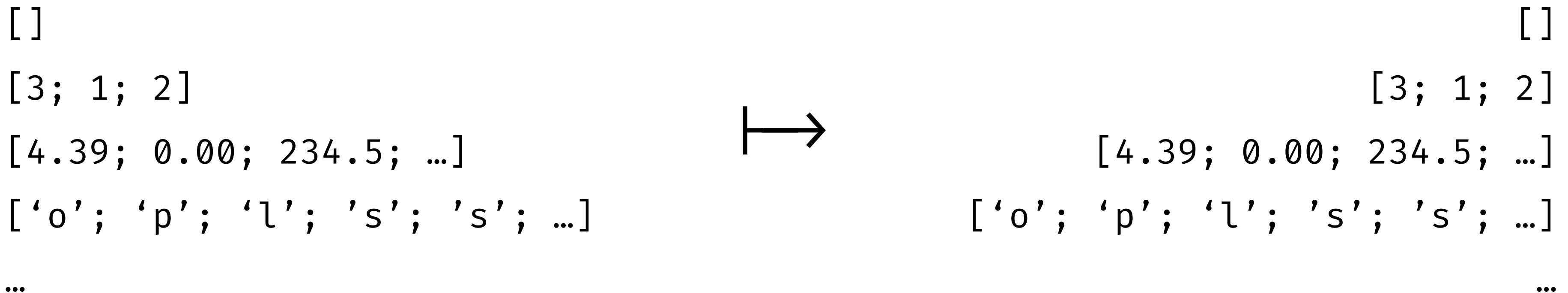
Observational equivalence

equivalent
inputs \mapsto equivalent
outputs

How do we test for
observational equivalence ?

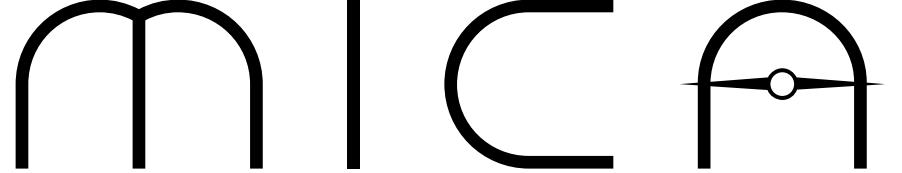
Property-based testing

$\text{rev}(\text{rev lst}) = \text{lst}$

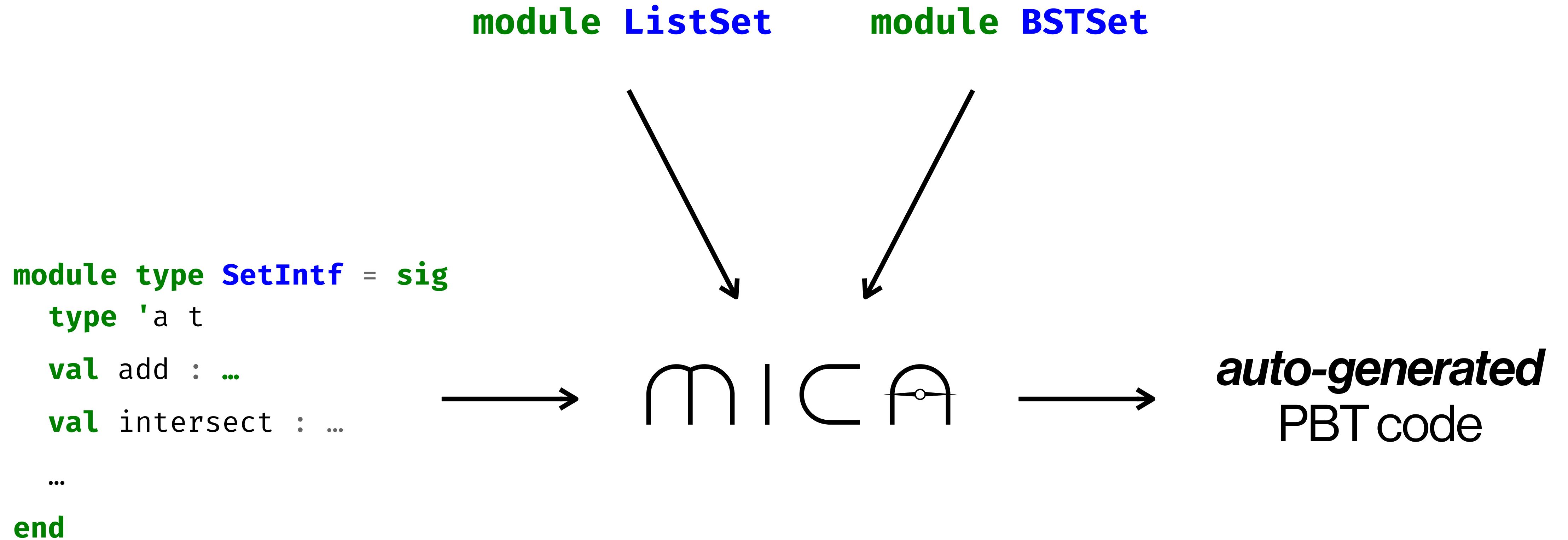


Problem:

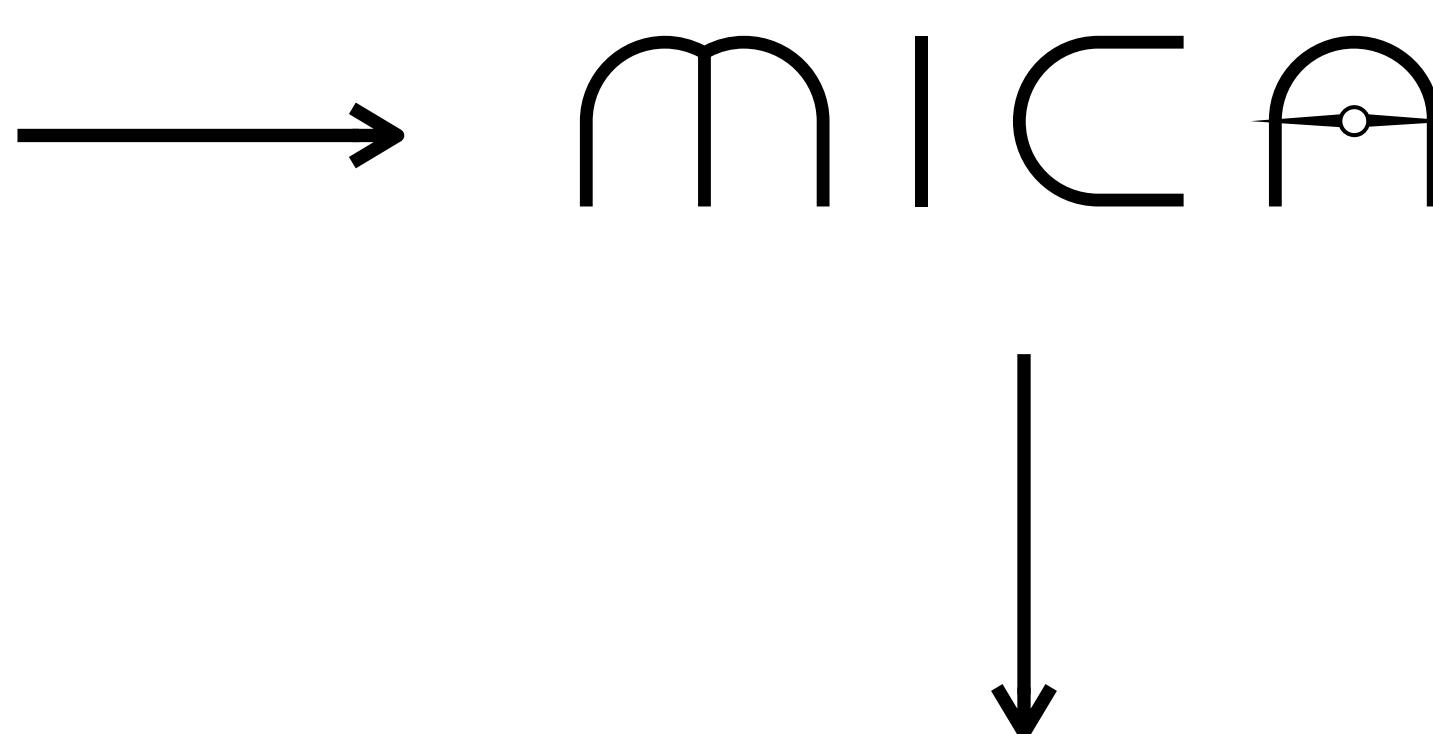
writing PBT code for different modules is time-consuming!

Solution: The logo consists of the lowercase letters 'm', 'i', 'c', and 'a' arranged vertically. A small circle is positioned at the top of the vertical stroke of the letter 'c'.

m | C A overview



```
module type SetIntf = sig
  type 'a t
  val empty      : ...
  val add        : ...
  val intersect  : ...
  ...
end
```

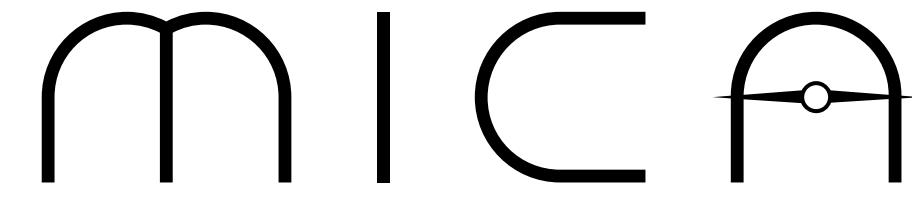


```
type expr =
| Empty
| Add of int * expr
| Intersect of expr * expr
...
type ty     = Bool | Int | T
type value =
| ValBool of bool
| ValInt of int
| ValT of int M.t

val gen_expr : ty → expr Generator.t
val interp   : expr → value
```

*auto-generated
PBT code*

Demo



automatically produces:

```
type expr =
| Empty
| Add of int * expr
| Intersect of expr * expr
...
type ty    = Bool | Int | T
type value =
| ValBool of bool
| ValInt of int
| ValT of int M.t
```

datatype definitions
representing
symbolic commands





MICΑ automatically produces:

Generator for **well-typed** sequences
of symbolic commands

val gen_expr : ty → expr **Generator**.t

`gen_expr` **T**

well-typed command sequences
that return type **T**

`gen_expr T`

well-typed command sequences
that return type **T**

Intersect (Add 2 Empty) Empty



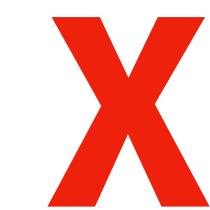
`gen_expr T`

well-typed command sequences
that return type **T**

Intersect (Add 2 Empty) Empty



Is_empty (Size Empty)





MICAH automatically produces:

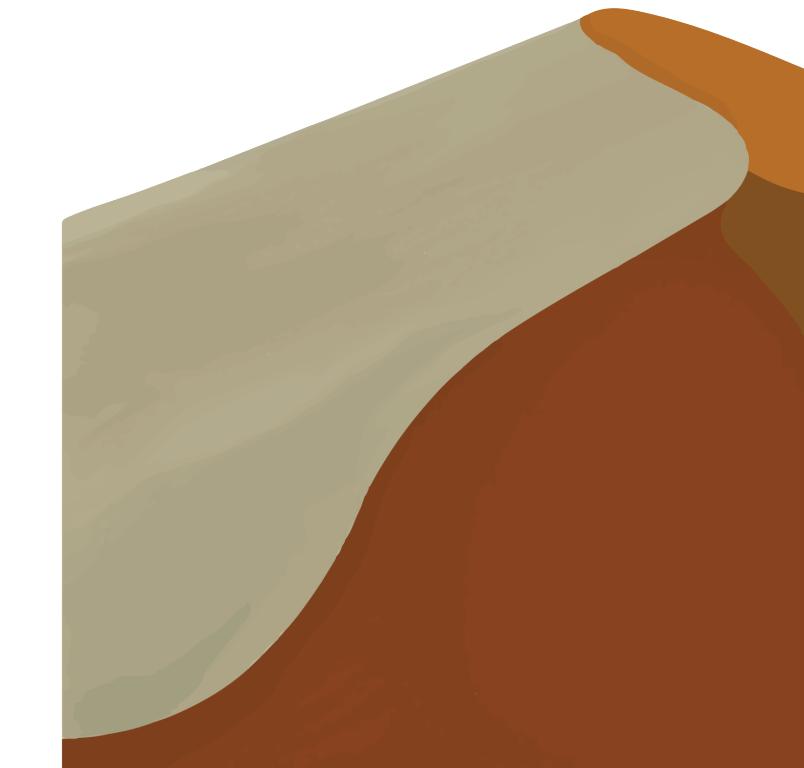
Interpreter for symbolic commands

val interp : expr → value



automatically produces:

Executable for testing observational equivalence



DUNE

Generator

generate **random**
command sequences

```
(Is_empty  
  (Intersect (Rem 8 (Add 7 Empty))  
    (Union (Add 2 Empty)  
      (Union Empty Empty)))
```

Generator

generate **random**
command sequences

```
(Is_empty  
  (Intersect (Rem 8 (Add 7 Empty))  
            (Union (Add 2 Empty)  
                  (Union Empty Empty)))
```

Interpreter

interpret commands
over modules

... → **module BSTSet** → ...
... → **module ListSet** → ...

Generator

generate **random**
command sequences

```
(Is_empty  
  (Intersect (Rem 8 (Add 7 Empty))  
            (Union (Add 2 Empty)  
                  (Union Empty Empty)))
```

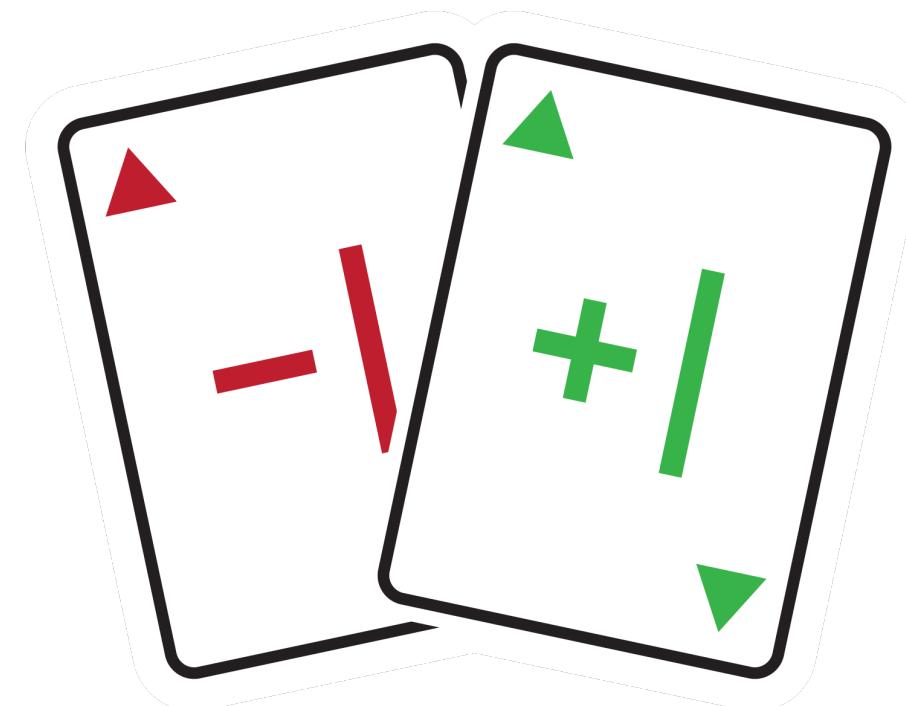
Interpreter

interpret commands
over modules

... → **module BSTSet** → ...
... → **module ListSet** → ...

Executable

test for
observational
equivalence





Jane Street



BASE_QUICKCHECK



CORE

Parser

inhabitedtype/ angstrom



Parser combinators built for speed and memory efficiency

Code generator

fpottier/pprint

A pretty-printing combinator library for OCaml



Examining the *automatically generated* PBT code

The **expr** type

expressions = sequences of symbolic commands

```
module type SetIntf = sig
```

```
  type 'a t
```

```
  val empty : 'a t
```

```
  val add : 'a t → 'a t → 'a t
```

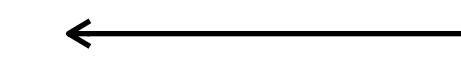
```
  val union: 'a t → 'a t → 'a t
```

```
  val size: 'a t → int
```

```
  val is_empty : 'a t → bool
```

```
  ...
```

```
end
```



```
type expr =  
| Empty
```



```
| Add of int * expr
```



```
| Union of expr * expr
```



```
| Size of expr
```



```
| Is_empty of expr
```

```
...
```

Possible **types** & **values** that can be returned by **exprs**

```
type ty =  
| Bool  
| Int  
| T
```

```
type value =  
| ValBool of bool  
| ValInt of int  
| ValT of int M.t
```

Generator for expr's

```
let rec gen_expr (ty : ty) : expr Generator.t =
  let%bind k = QC.size in
  match ty, k with
  | (T, 0) → return Empty
  | (T, _) →
    let intersect =
      let%bind e1 = QC.with_size ~size:(k / 2) (gen_expr T) in
      and e2 = QC.with_size ~size:(k / 2) (gen_expr T) in
      QC.return @@ Intersect(e1, e2) in
    ...
    QC.union [ intersect; ... ]
  ...
  ...
```

Invoking QC generators for opaque types

```
module type MapIntf = sig
  type t
  val from_list : AssocList.t → t
  ...
end

let rec gen_expr (ty : ty) : expr Generator.t =
  match ty, QC.size with
  | (T, _) → ...
    let from_list =
      let%bind xs = [%quickcheck.generator: AssocList.t] in
      G.return @@ From_list xs
      in G.union [ from_list; ... ]
  ...

```

Interpreter for expr's

```
module ExprToImpl (M : SetInterface) = struct

let rec interp (expr : expr) : value =
  match expr with
  | Empty → ValT (M.empty)
  | Add(x1, e2) → match interp e2 with
    | ValT e' → ValT (M.add x1 e')
    ...
  | Union(e1, e2) →
    match (interp e1, interp e2) with
    | (ValT e1', ValT e2') → ValT (M.union e1' e2')
    ...
  ...
end
```

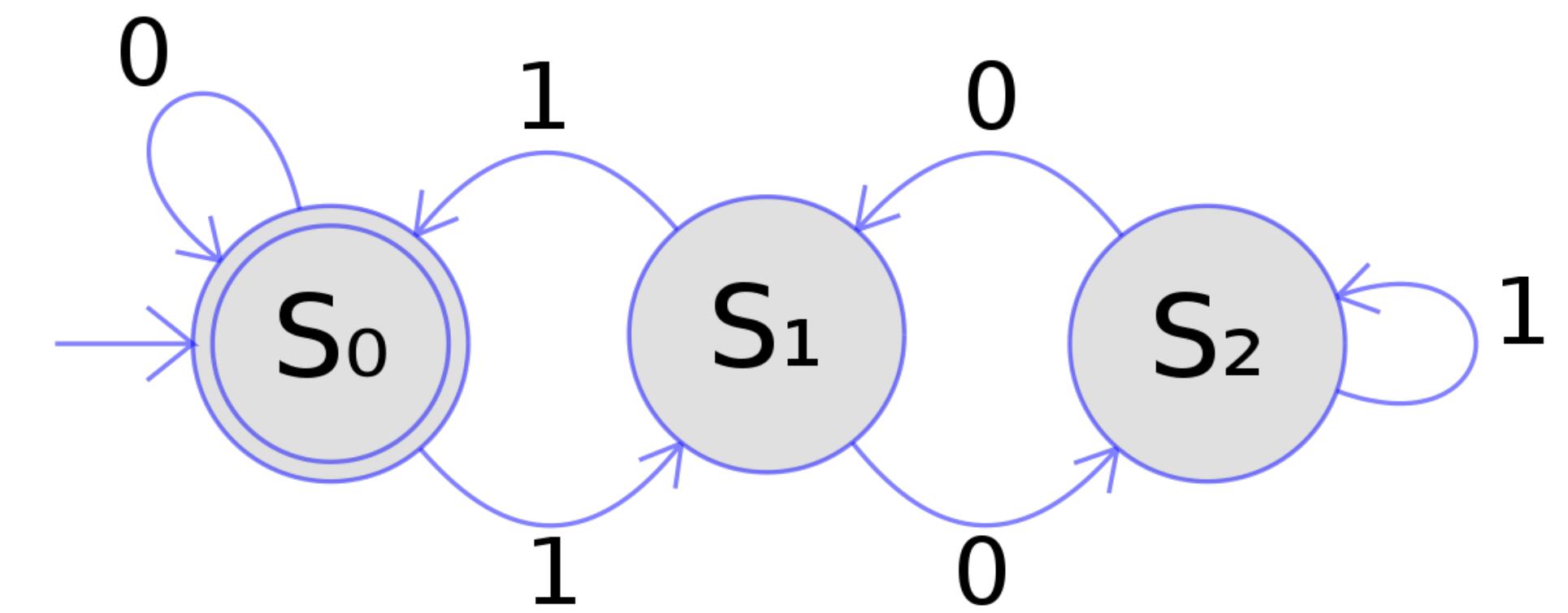
Case studies

Regex matching

Brzozowski derivatives

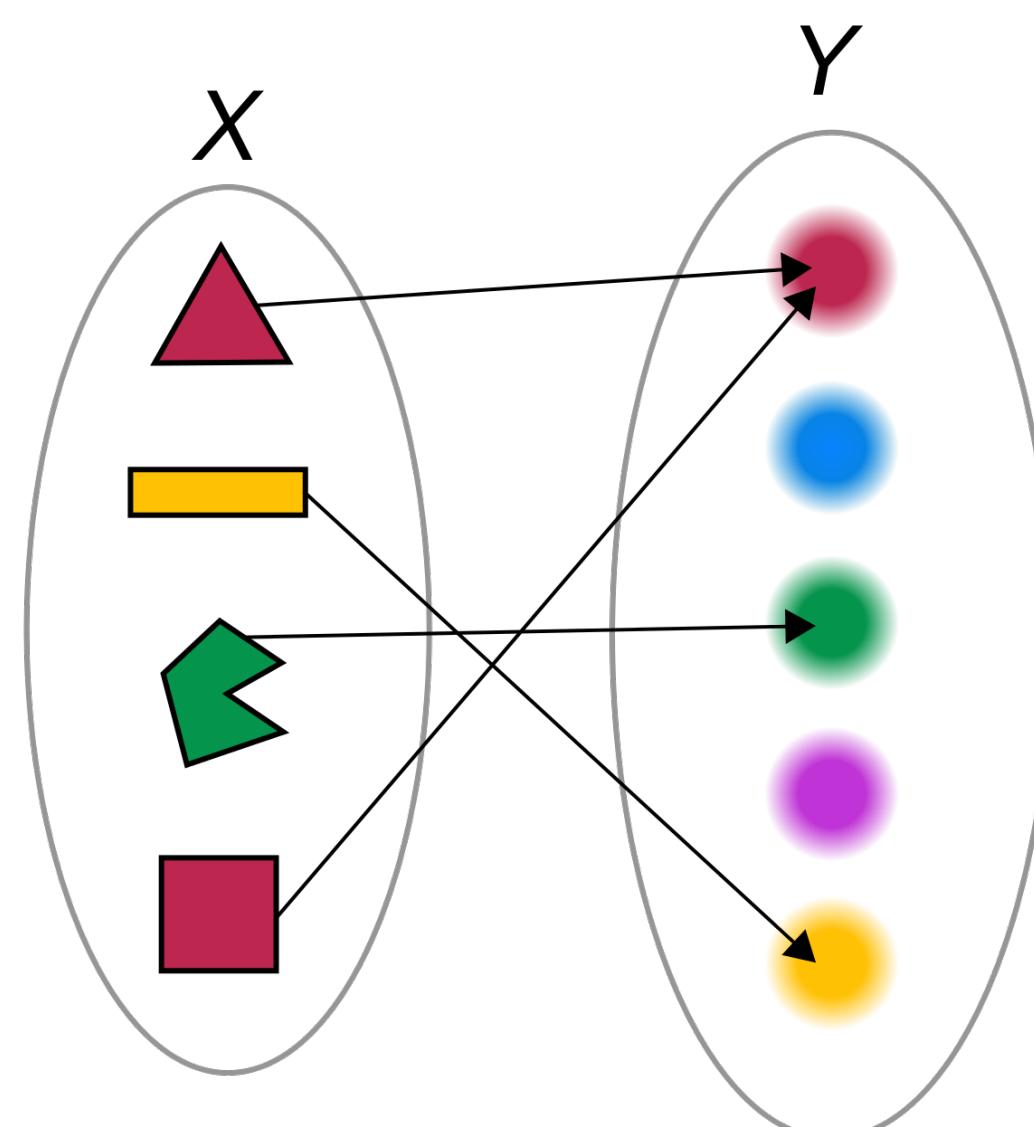
$$u^{-1}S = \{v \in \Sigma^* \mid uv \in S\}$$

DFA

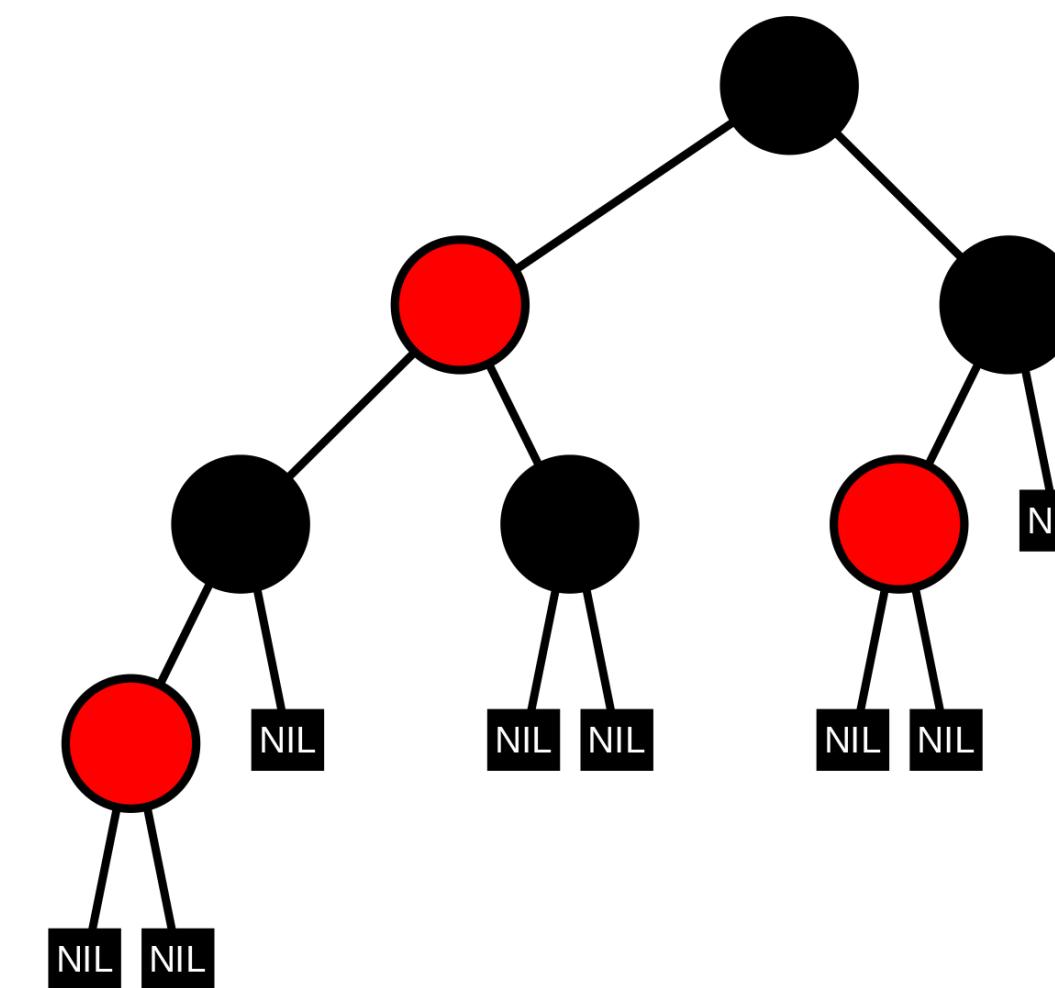


Functional maps

Association lists



Red-Black Trees

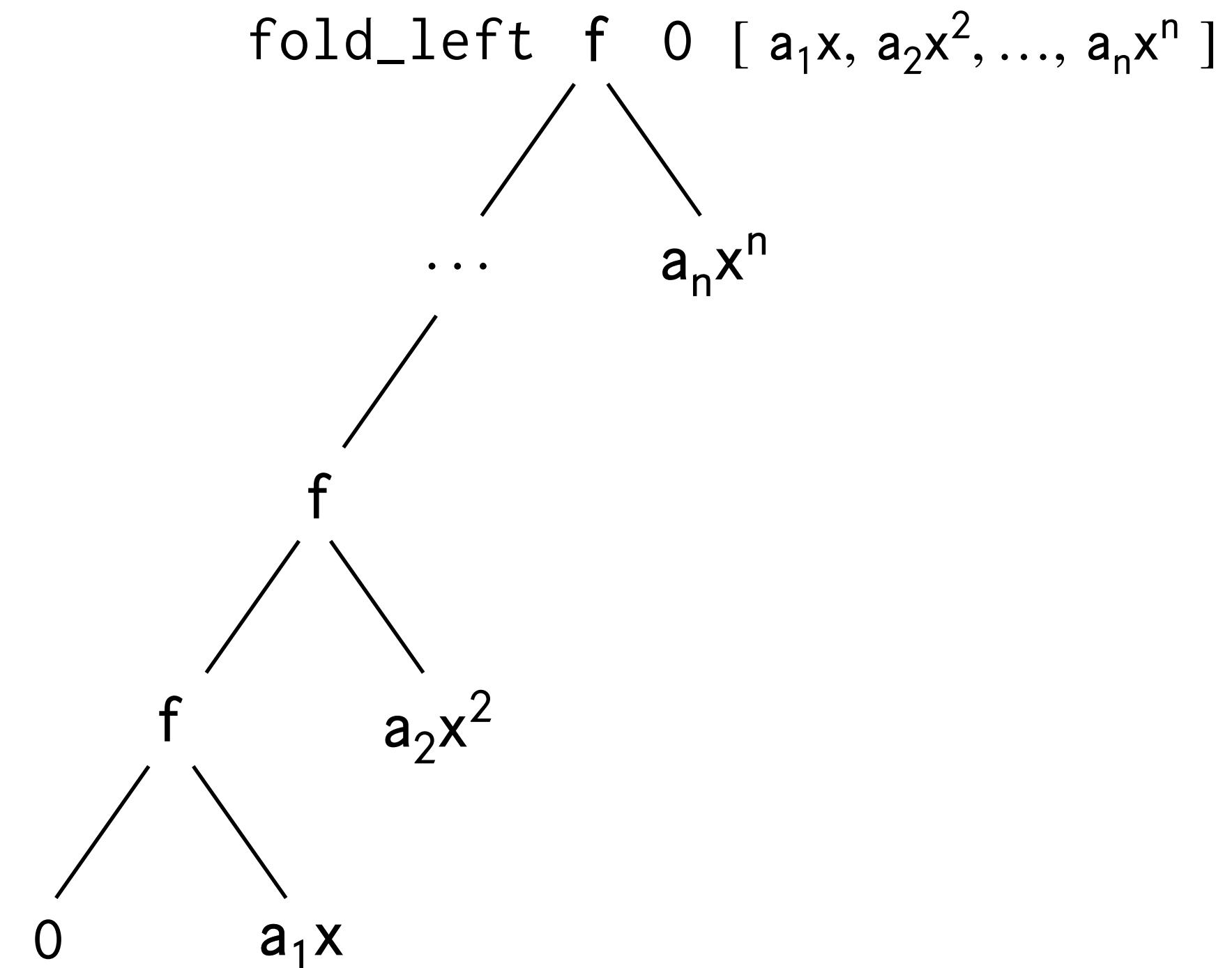


Polynomials

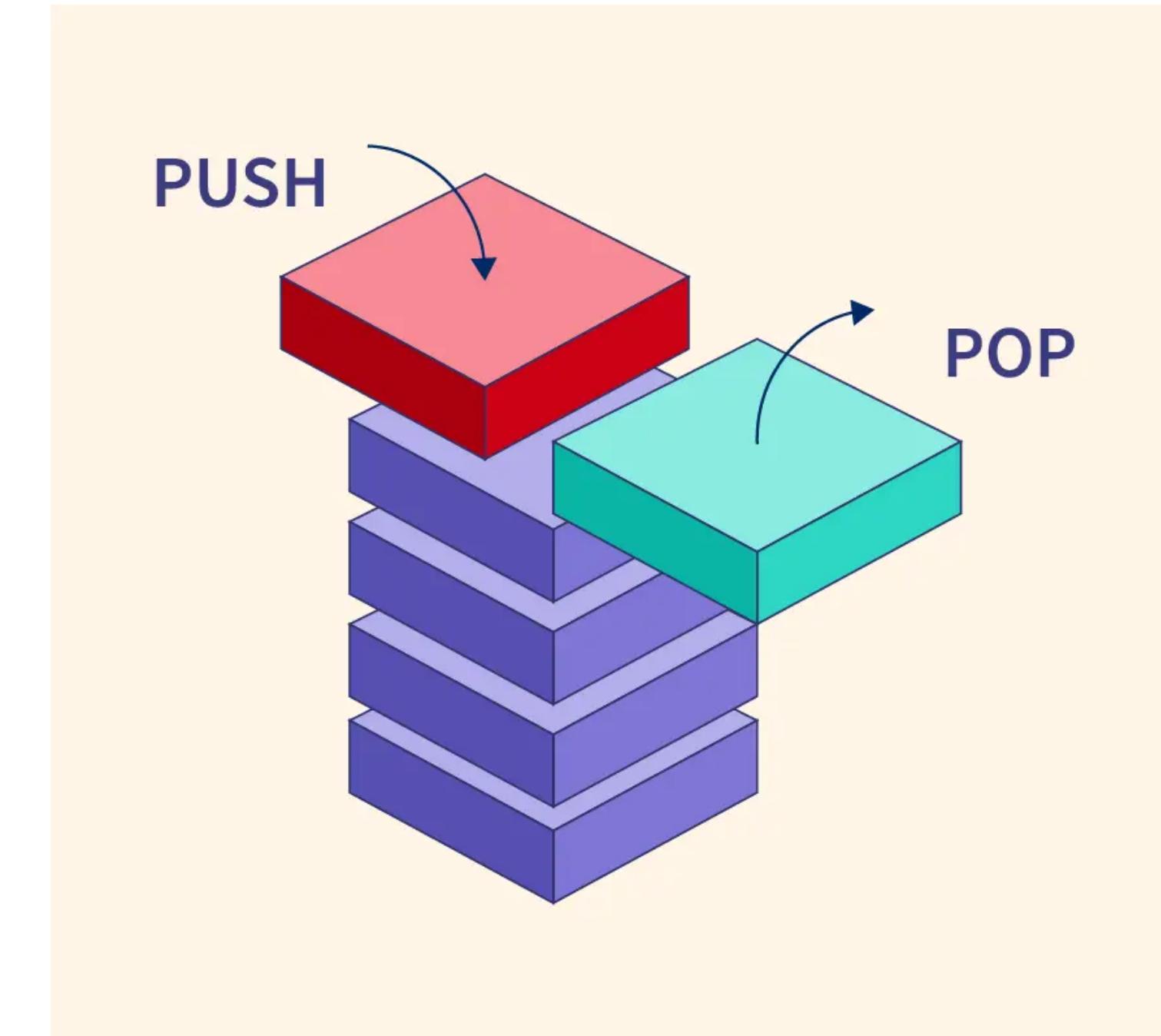
Horner's algorithm

$$\begin{aligned} p(x_0) &= a_0 + x_0 \left(a_1 + x_0 \left(a_2 + \cdots + x_0 (a_{n-1} + b_n x_0) \cdots \right) \right) \\ &= a_0 + x_0 \left(a_1 + x_0 \left(a_2 + \cdots + x_0 b_{n-1} \right) \right) \\ &\vdots \\ &= a_0 + x_0 b_1 \\ &= b_0. \end{aligned}$$

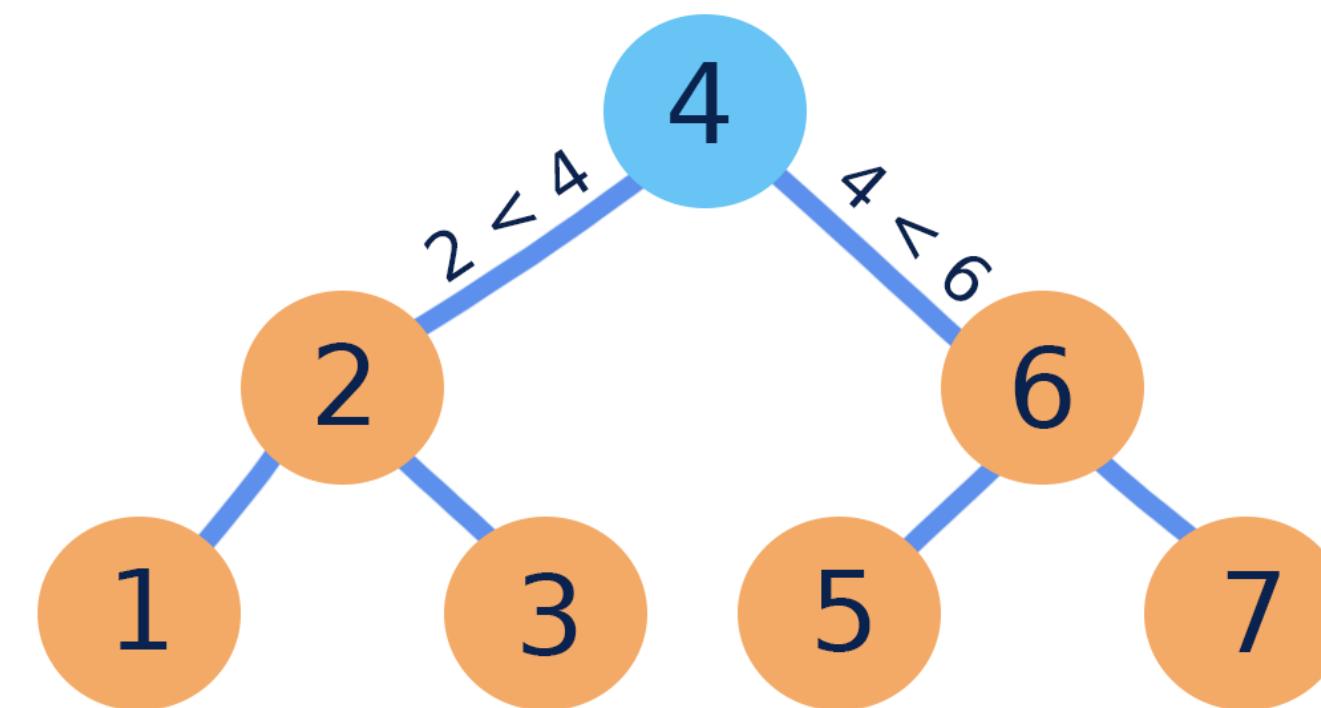
Fold over list of monomials



Stacks

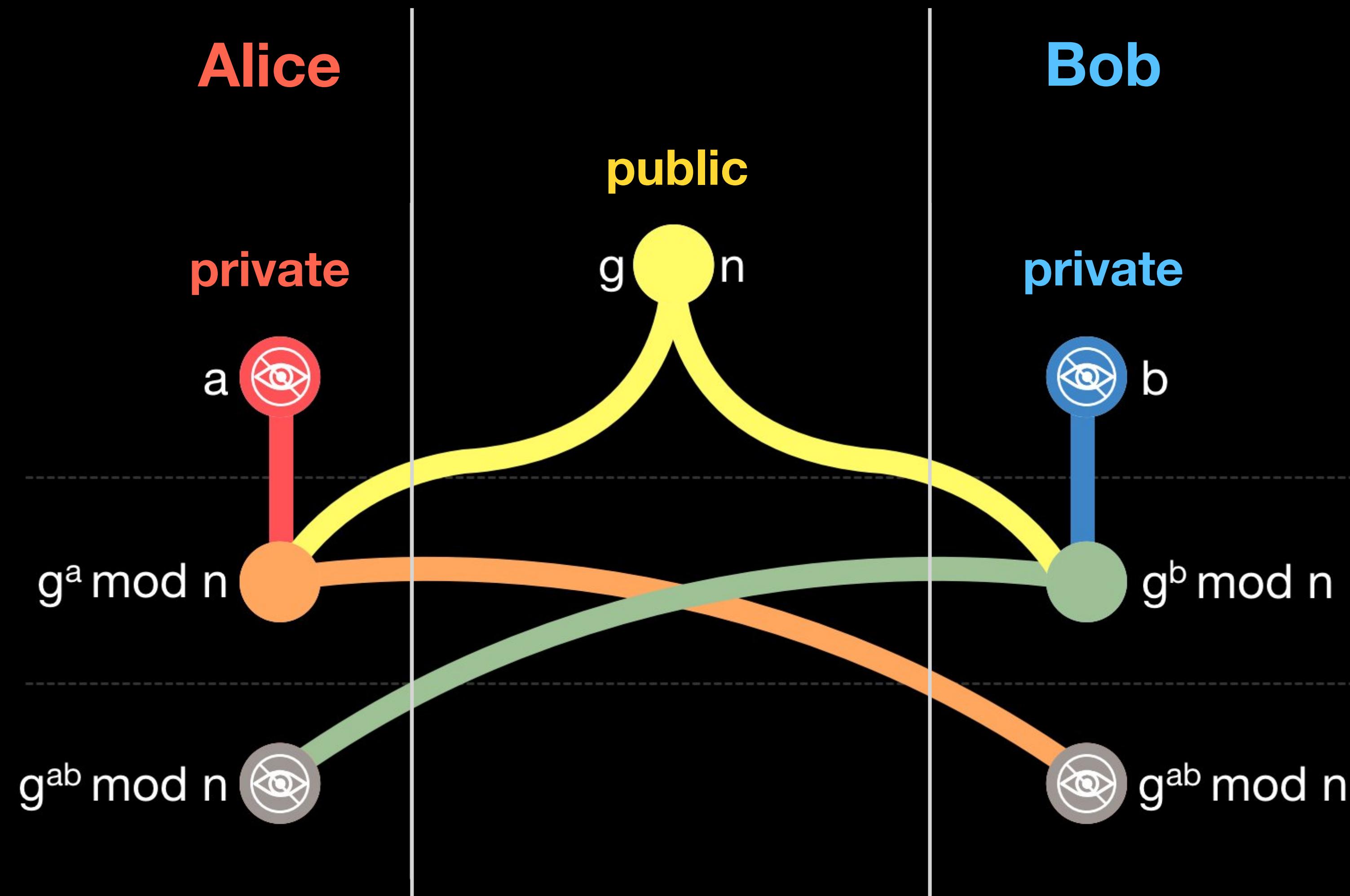


Sets (BSTs, lists)



Diffie-Hellman key exchange

(work in progress)



Related work

History of model-based PBT



Monadic QuickCheck

[Claessen & Hughes 2002]



QuviQ QuickCheck

[Hughes 2016]

Model-based PBT



QCSTM

[Midgaard 2020]

Model_quickcheck

[Dumont 2020]

- Algebraic data types for representing symbolic commands
- Mica adds support for invariants + binary operations on abstract types

Random testing of ML modules



Monolith

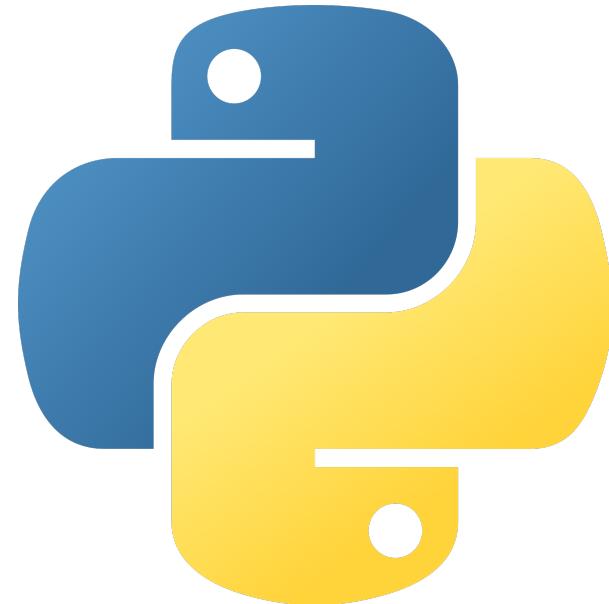
[Pottier 2021]

Articheck

[Braibant et al. 2014]

- GADT-based DSLs for testing ML modules
- Mutation-based fuzzing
- Mica automatically derives the requisite PBT code

Automatic generation of PBT code



Hypothesis
Ghostwriter

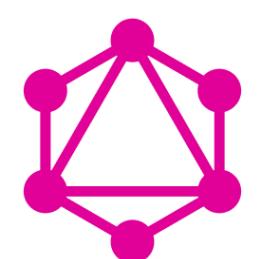
[Hatfield-Dodds et al. 2020]



Clojure

QuickREST

[Karlsson et al. 2020, 2019]



GraphQL

Future work

Future work

Handle imperative code



Graphic from [Ahrefs](#)

Future work

Encode dependencies in
generated command sequences

Rem **2** (Add **2** Empty)

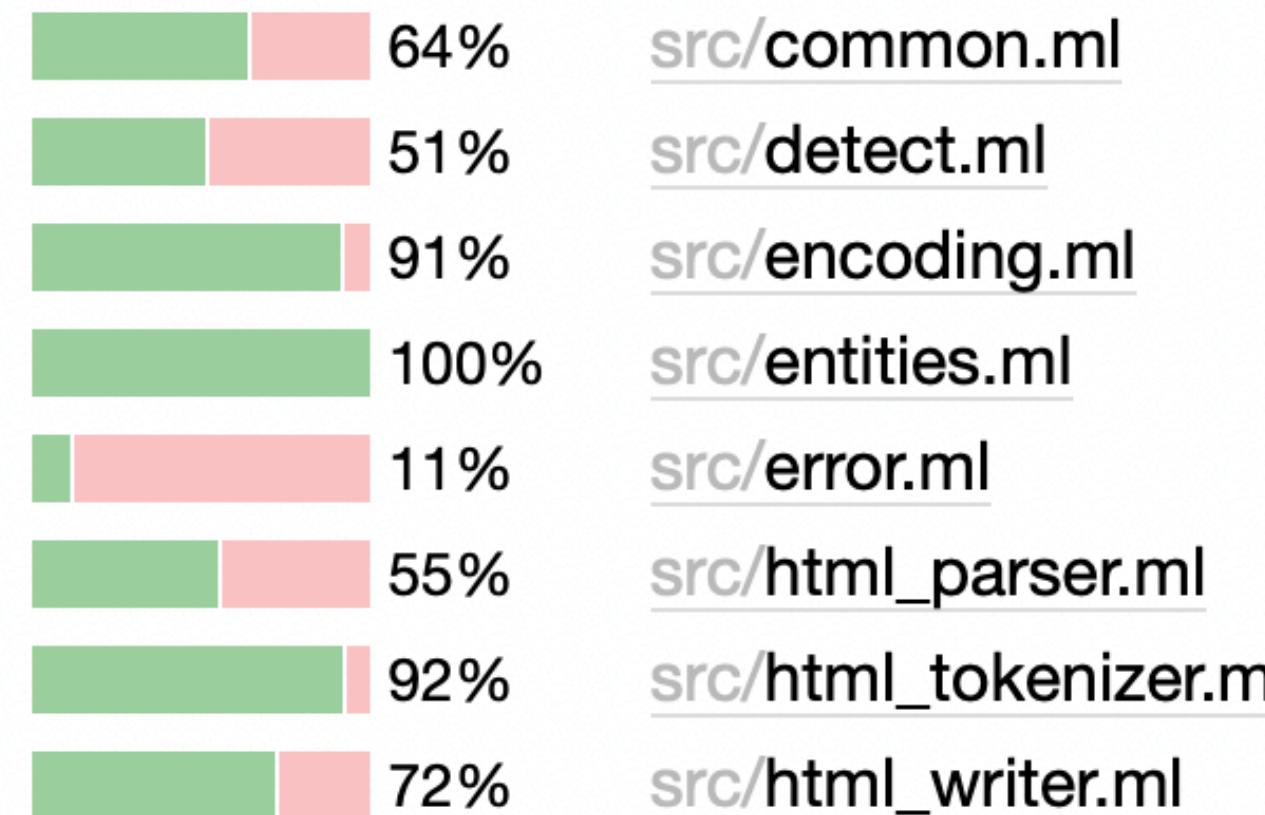


Future work

Empirical evaluation

ppx_bisect

Coverage report 72.15%



Haskell QuickCheck

```
>>> quickCheck prop_sorted_sort
+++ OK, passed 100 tests; 1684 discarded.
```

List elements (109 in total):

3.7%	0
3.7%	17
3.7%	2
3.7%	6
2.8%	-6
2.8%	-7

Future engineering work

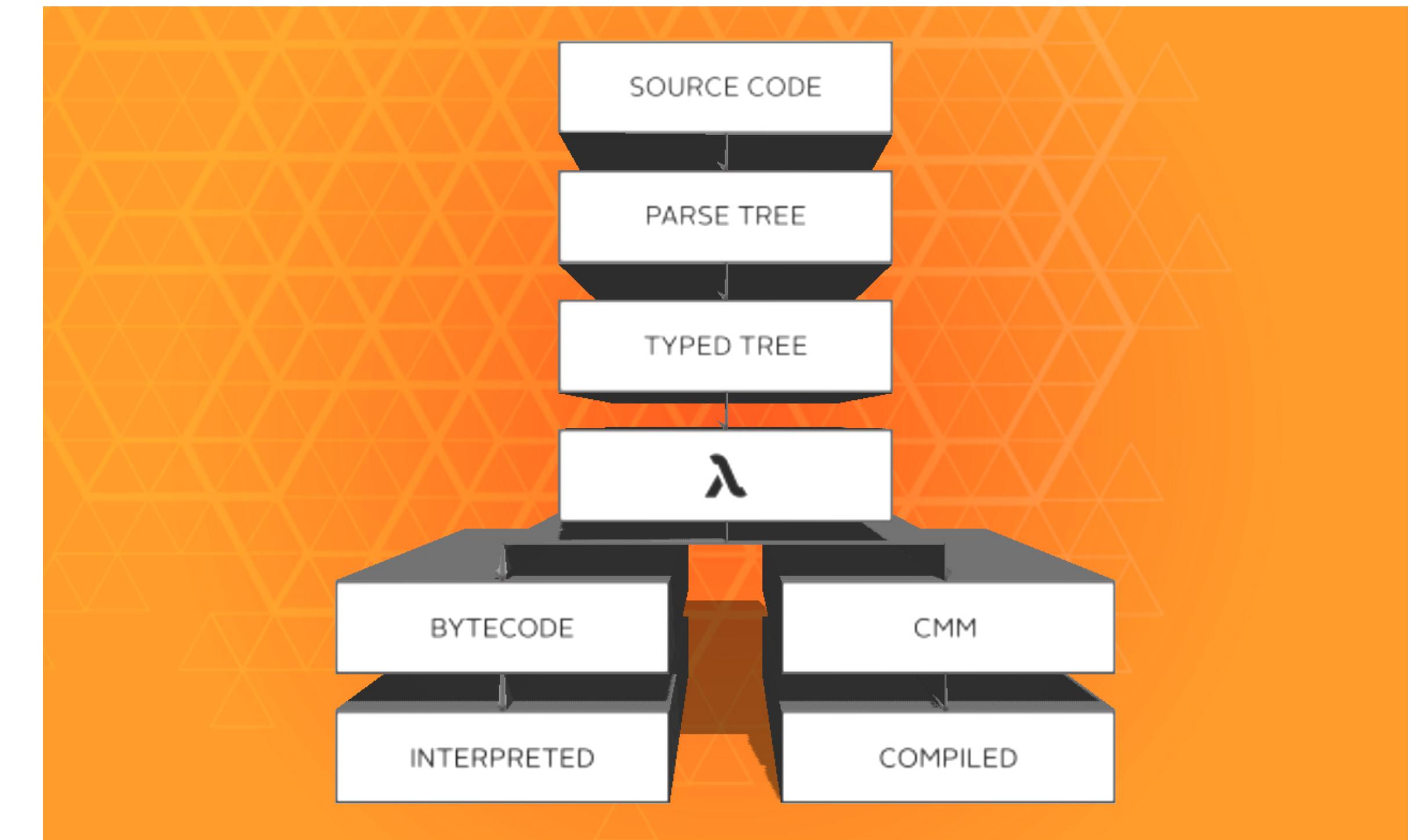
Shrinking

Add **5** (Rem **2** (Add **2** Empty))



Add **5** Empty

Integrate with OCaml compiler



Graphic from [Jane Street](#)

Takeaways



Graphic from *Real World OCaml*

Takeaways

1. Checking observational equivalence for ML modules is time-consuming



Graphic from *Real World OCaml*

Takeaways

1. Checking observational equivalence for ML modules is time-consuming
2. $M \subset A$ can automate this process via PBT!



Graphic from *Real World OCaml*

Thank you!

(Questions?)

nbernest@seas.upenn.edu

github.com/nbernest/module_pbt

Appendix

Appendix 1: Bugs caught

“Bug Bounty” Overview

- **Total of 15 bugs artificially inserted across all three pairs of modules**
 - (the bugs are detailed in the following slides)
- **14 bugs caught**
- **Experimental limitations:**
 - **Bugs were introduced one at a time (no cascading bugs)**
 - **In each trial, the bug was only introduced in one module, with the other module remaining correct**

Artificial bugs (Stack & Set examples)

Artificially introduced bug	Failing test case	Correct result	Erroneous result
Is_empty empty = false	Is_empty Empty	TRUE	FALSE
Peek s = None (forall s)	Peek (Push -1 Empty)	-1	None
push x stack = stack @ [x]	Peek (Push -8 (Push 3 Empty))	-8	3
length (h :: t) = length t	Length (Push -1 Empty)	1	0
Wrong invariant (List implementation of Sets)	Invariant Empty	TRUE	FALSE
let invariant (s : 'a list) : bool = (* not @@ *) List.contains_dup ~compare:(=) s	Invariant (Union (Add 5 (Union (Union Empty Empty) (Add -3 Empty))) (Rem -10 (Add 5 (Add 9 Empty)))) = ... = Invariant [-3; 5; 5; 9]	Invariant [-3; 5; 9] = TRUE	Invariant [-3; 5; 5; 9] = FALSE
Not enforcing Set invariant (Union for ListSet)			
let union s1 s2 = s1 @ s2 (* ▷ dedup *)			

```

type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

```

Artificially introduced bug	Failing test case	Correct	Erroneous
Spurious recursive call (BST insertion) <pre> let rec add x t = match t with ... Node (lt, n, rt) -> if x < n then Node (add x lt, n, add x rt) ... </pre>	Fails BST invariant Invariant $\begin{aligned} & (\text{Rem } 2 \text{ (Add } -7 \\ & \quad (\text{Add } -3 \text{ (Intersect Empty Empty)))) \\ & = \dots \\ & = \text{Invariant } (\dots \text{ (Node } (-7, 3, -7))) \end{aligned}$	TRUE	FALSE
Spurious negation (Intersection of two BSTs) <pre> let intersect t1 t2 = let commonElts = filter ~f:(fun x -> not mem mem x t1) (inorderTraversal t2) in List.fold commonElts ~init:empty ~f:(fun acc x -> add x acc) </pre>	$\begin{aligned} & (\text{Is_empty} \\ & \quad (\text{Intersect} \text{ (Rem } 8 \text{ (Rem } -7 \text{ (Rem } 7 \text{ Empty)))) \\ & \quad (\text{Intersect} \text{ (Union} (\text{Add } 2 \text{ Empty}) \\ & \quad \quad (\text{Union} \text{ Empty Empty})) \\ & \quad (\text{Add } 8 \text{ (Add } 4 \text{ Empty})) \\ & = \dots \\ & = \text{Is_empty } \text{Empty} \end{aligned}$	TRUE	FALSE
Changing the base case of the BST invariant function to False instead of True (details omitted)	Invariant $\begin{aligned} & (\text{Union} \text{ (Add } 9 \text{ (Add } -4 \text{ (Rem } 6 \text{ Empty)))) \\ & \quad (\text{Intersect} \text{ (Add } -10 \text{ (Intersect Empty Empty)))} \\ & \quad (\text{Union} \text{ (Add } -6 \text{ Empty}) \\ & \quad \quad (\text{Union} \text{ Empty Empty}))) \\ & = \text{Invariant } \{-6, -6, -4, 9\} \end{aligned}$	TRUE	FALSE

```

type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree

```

BST implementation of Sets: removing an element

```

let rec rem (x : 'a) (t : 'a tree) : 'a tree =
  match t with ...
| Node (lt, n, rt) → ...

```

```

if x > n then Node (rem x lt, n, rem x rt)

```

Spurious recursive call

```

if x > n then Node (lt, n, rt)

```

Missing recursive call

Bug not caught

Technically, [rem x lt] doesn't do anything here

Bug caught

Failing test case	Correct result	Erroneous result
Size (Rem -2 (Add -2 (Add -3 (Add 4 Empty))))	2	3