



m | C | A

Automated Property-Based Testing for OCaml Modules

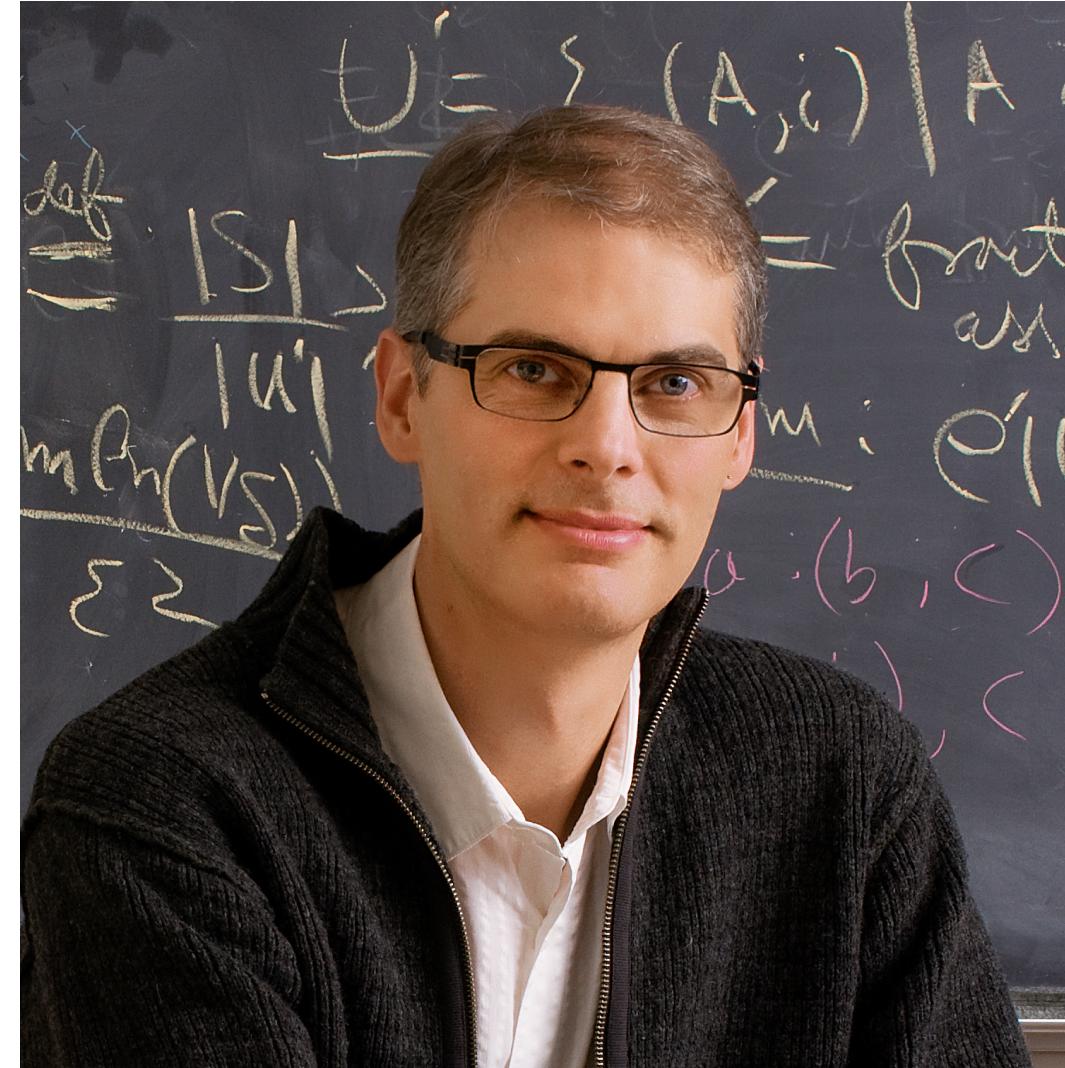
Ernest Ng
University of Pennsylvania
Advised by Harry Goldstein & Benjamin C. Pierce

OPLSS 2023

Advisors



Harry Goldstein
4th Year PhD

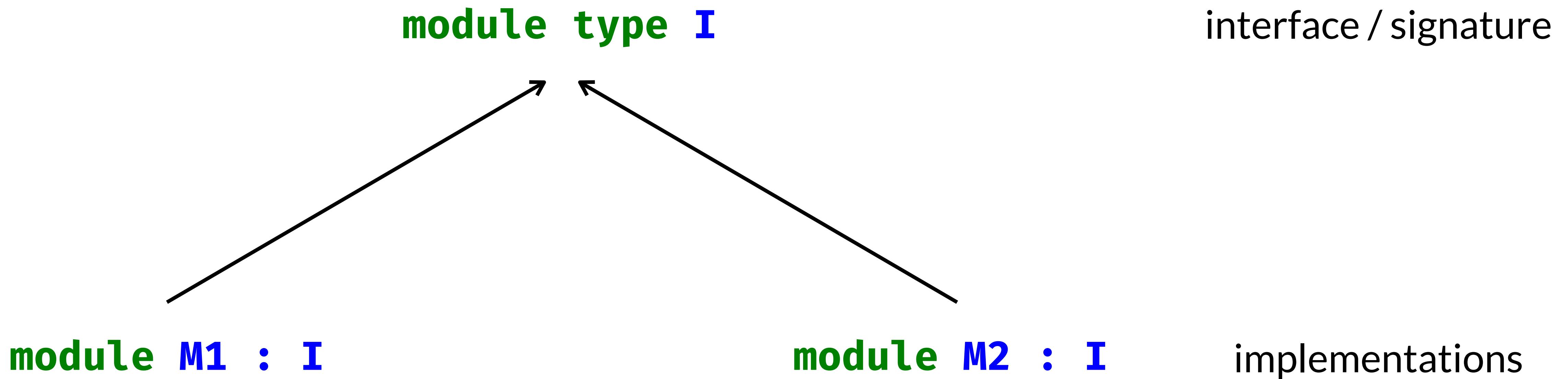


Benjamin C. Pierce
Professor

MOTIVATION

**MODULES
MATTER
MOST**

- Bob Harper (2011)



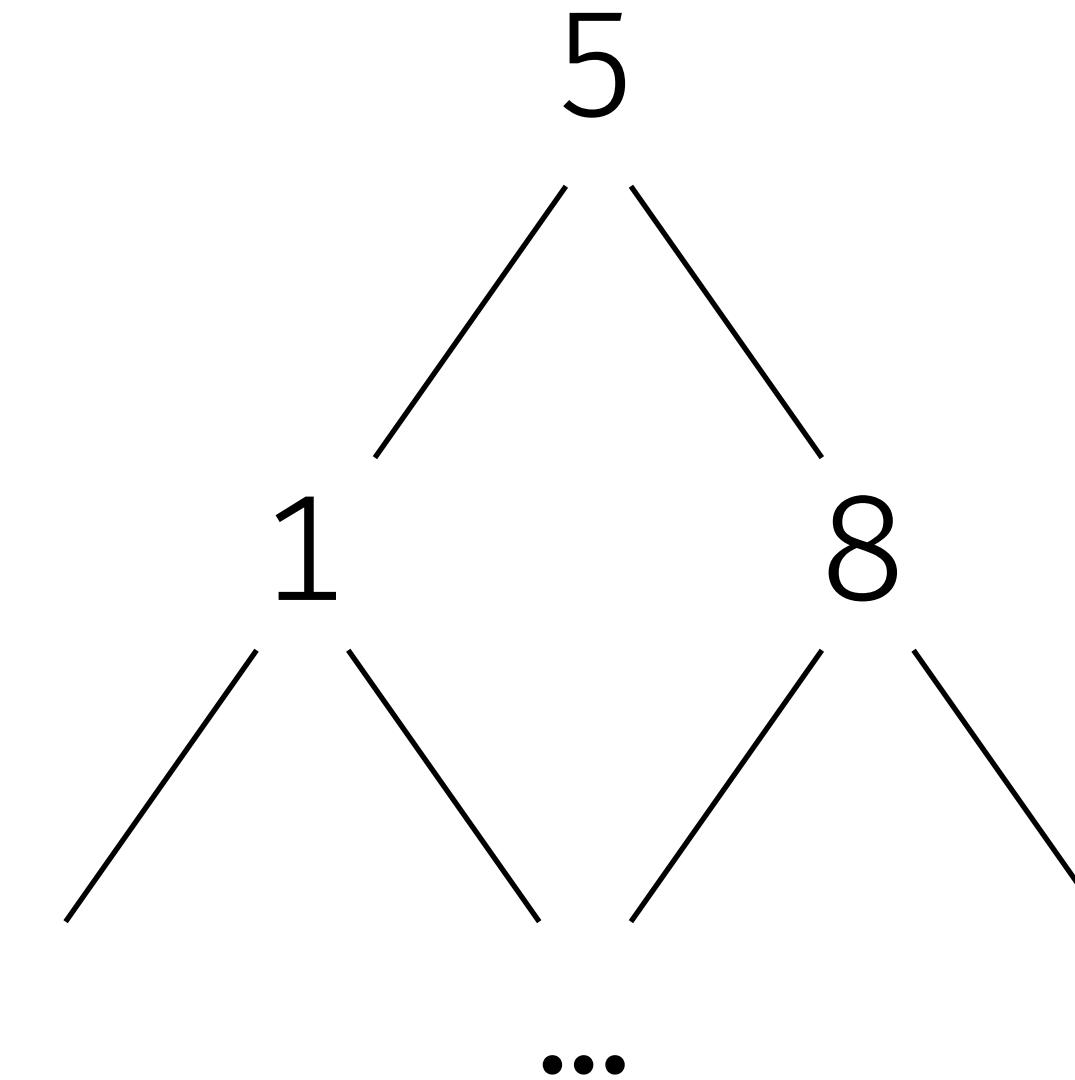
Example: Finite Sets

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

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



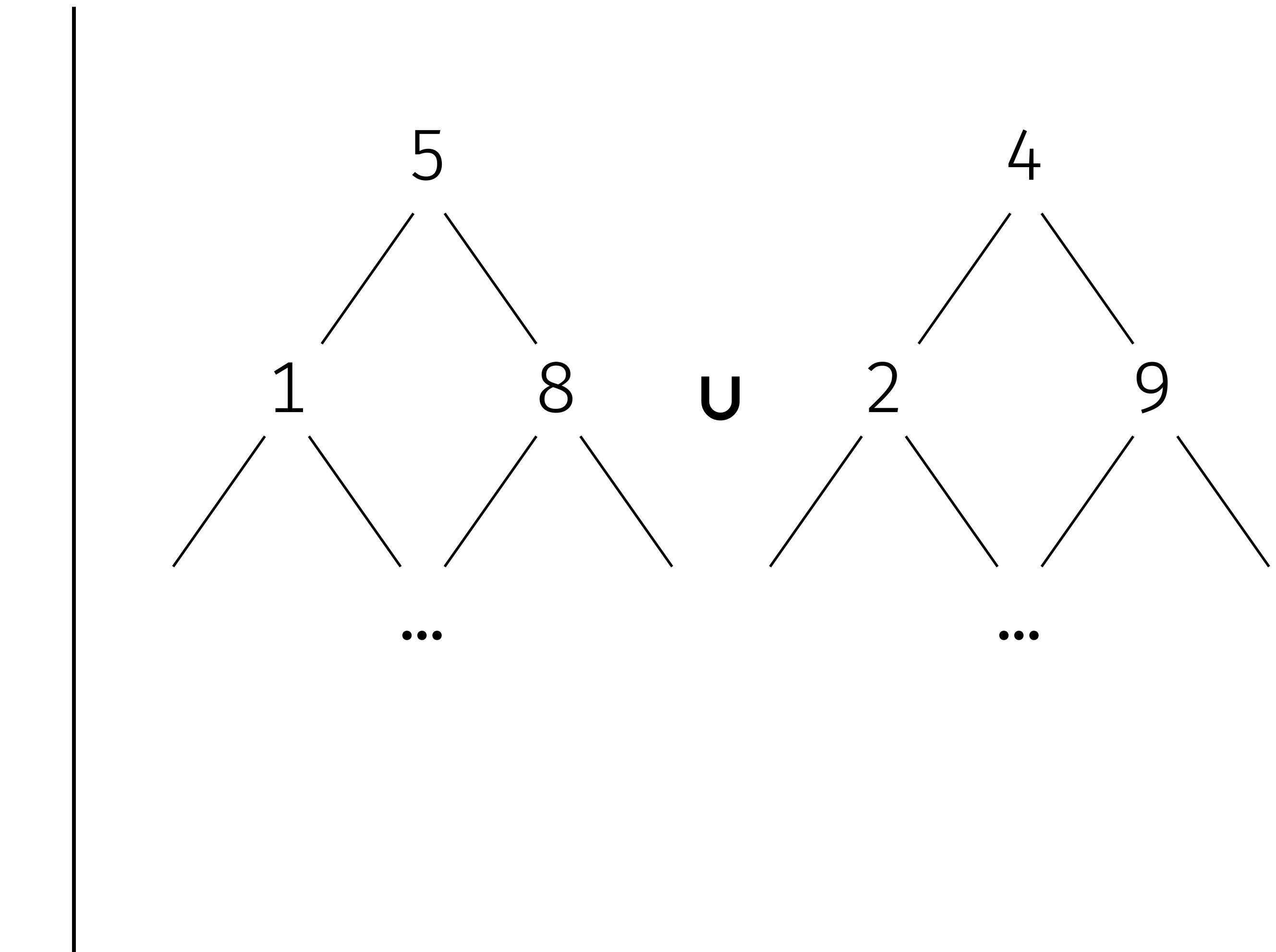
```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

```
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

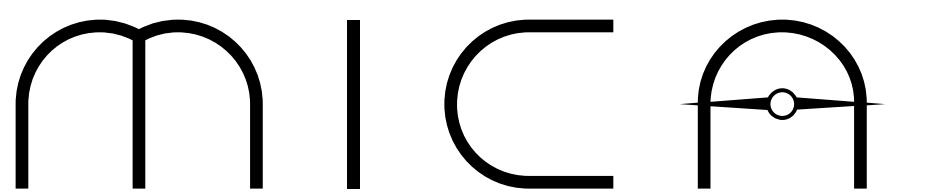
`rev (rev lst) = lst`

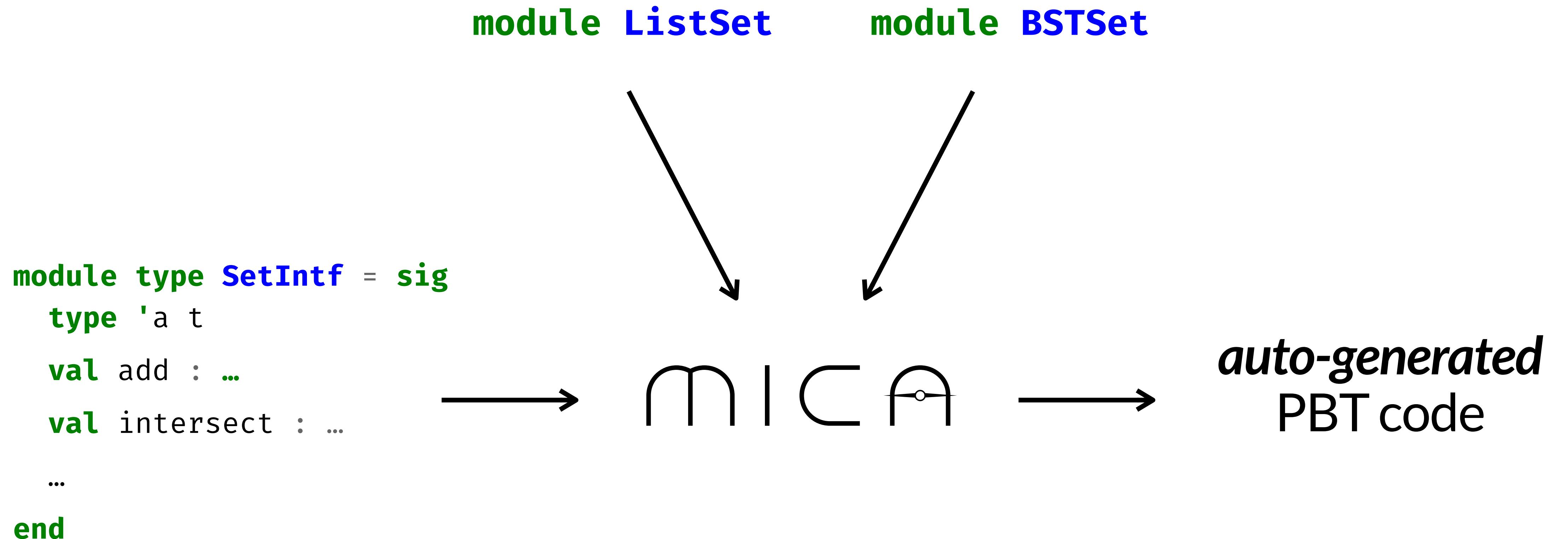
[]
[3; 1; 2]
[4.39; 0.00; 234.5; ...]
['o'; 'p'; 'l'; 's'; 's'; ...]
...
...



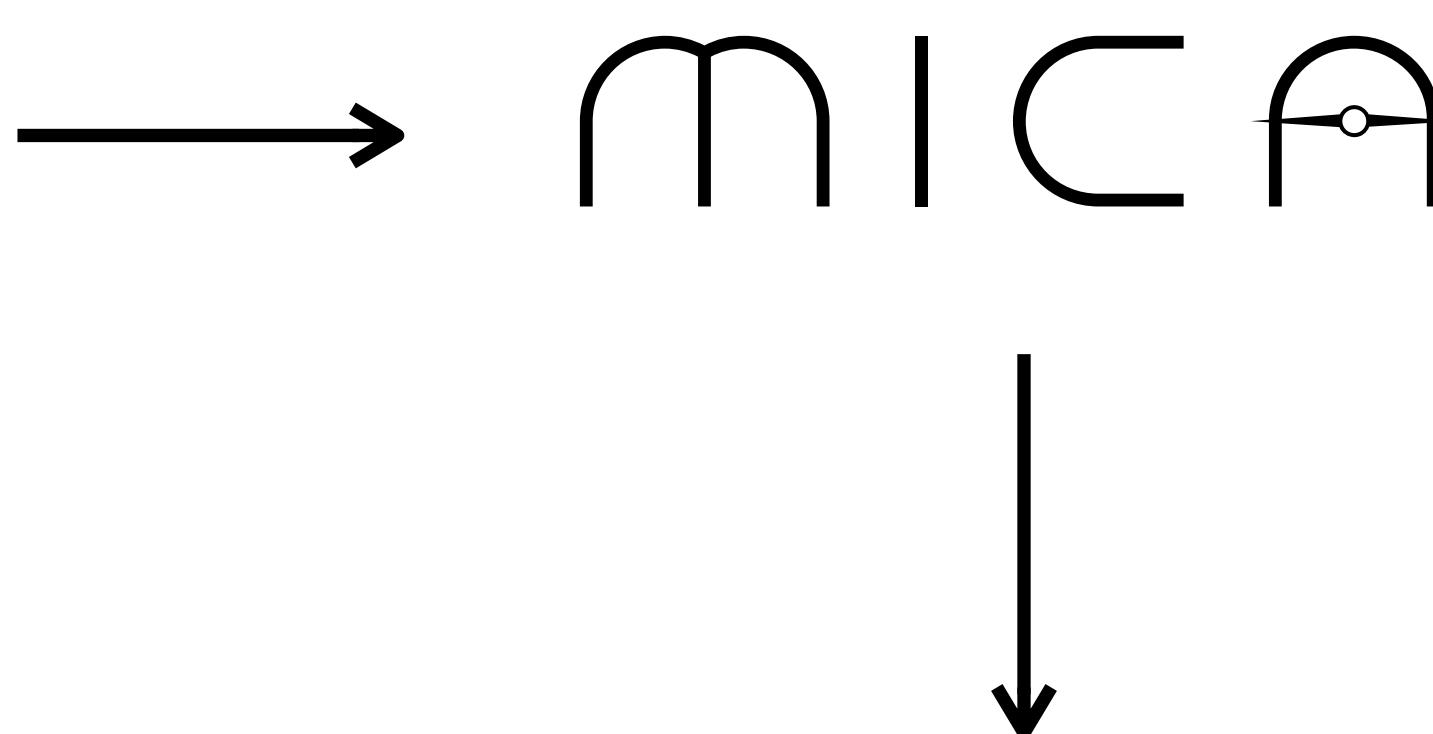
[]
[3; 1; 2]
[4.39; 0.00; 234.5; ...]
['o'; 'p'; 'l'; 's'; 's'; ...]
...
...

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'.



```
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*

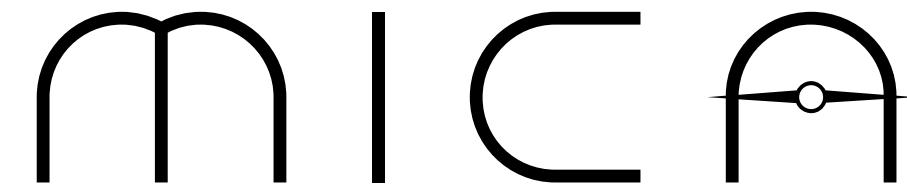
DEMO

(time permitting)

MICAI 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
```

ADT definitions
representing
symbolic commands

 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)



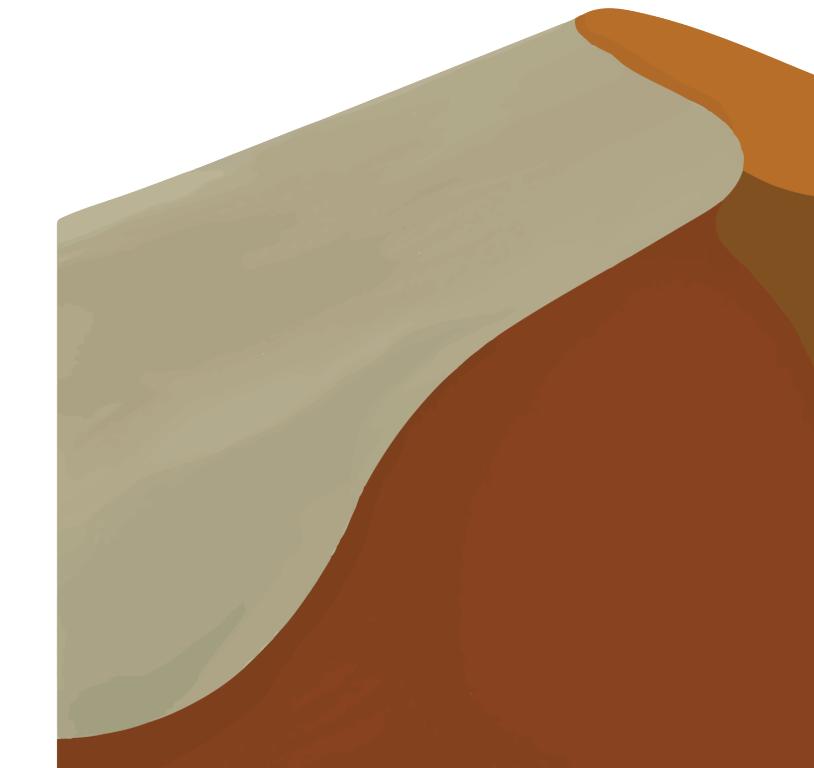
 automatically produces:

Interpreter for symbolic commands

val interp : expr → value

MICA 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)))
```

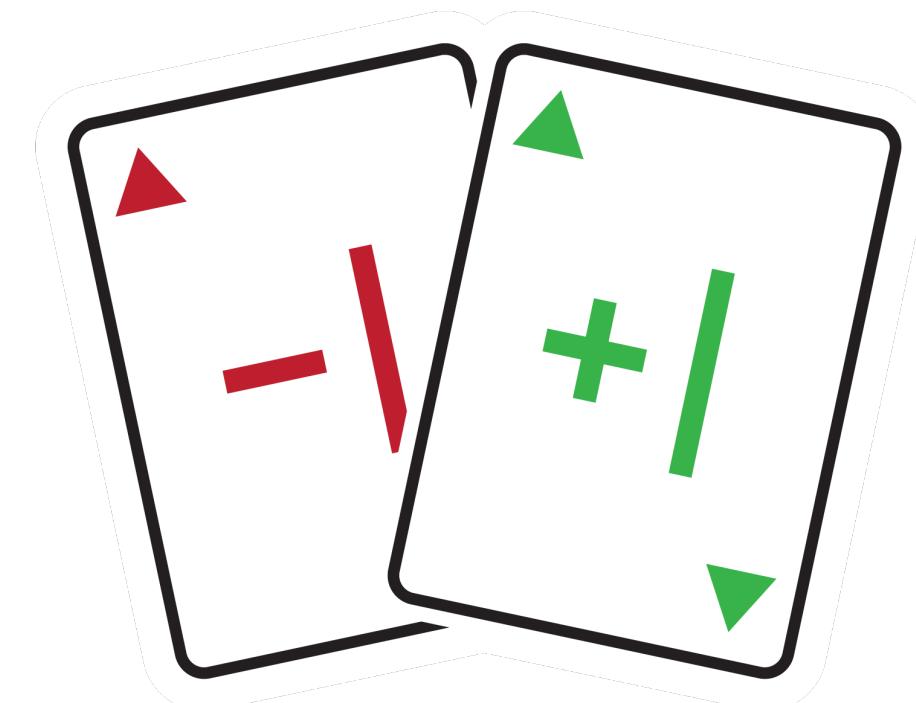
Interpreter

interpret commands
over modules

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

Executable

test for
observational
equivalence





Jane Street



BASE_QUICKCHECK



CORE

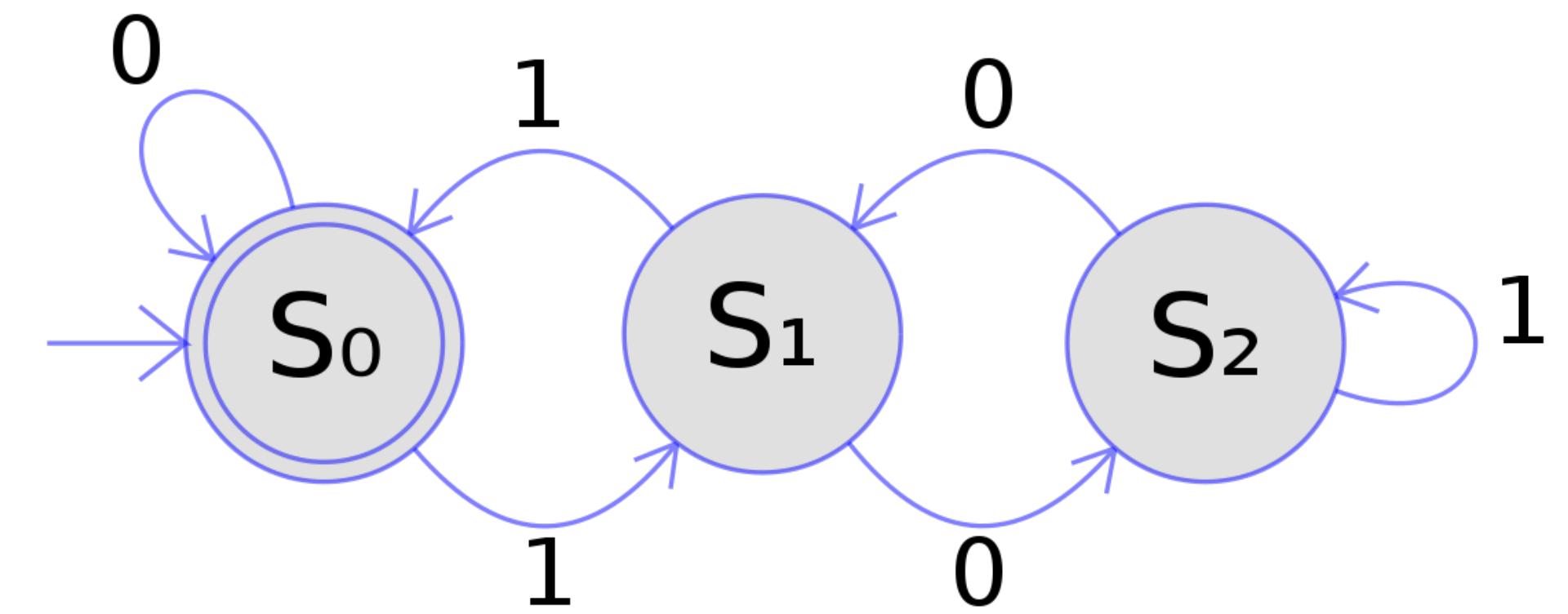
CASE STUDIES

Regex matching

Brzozowski derivatives

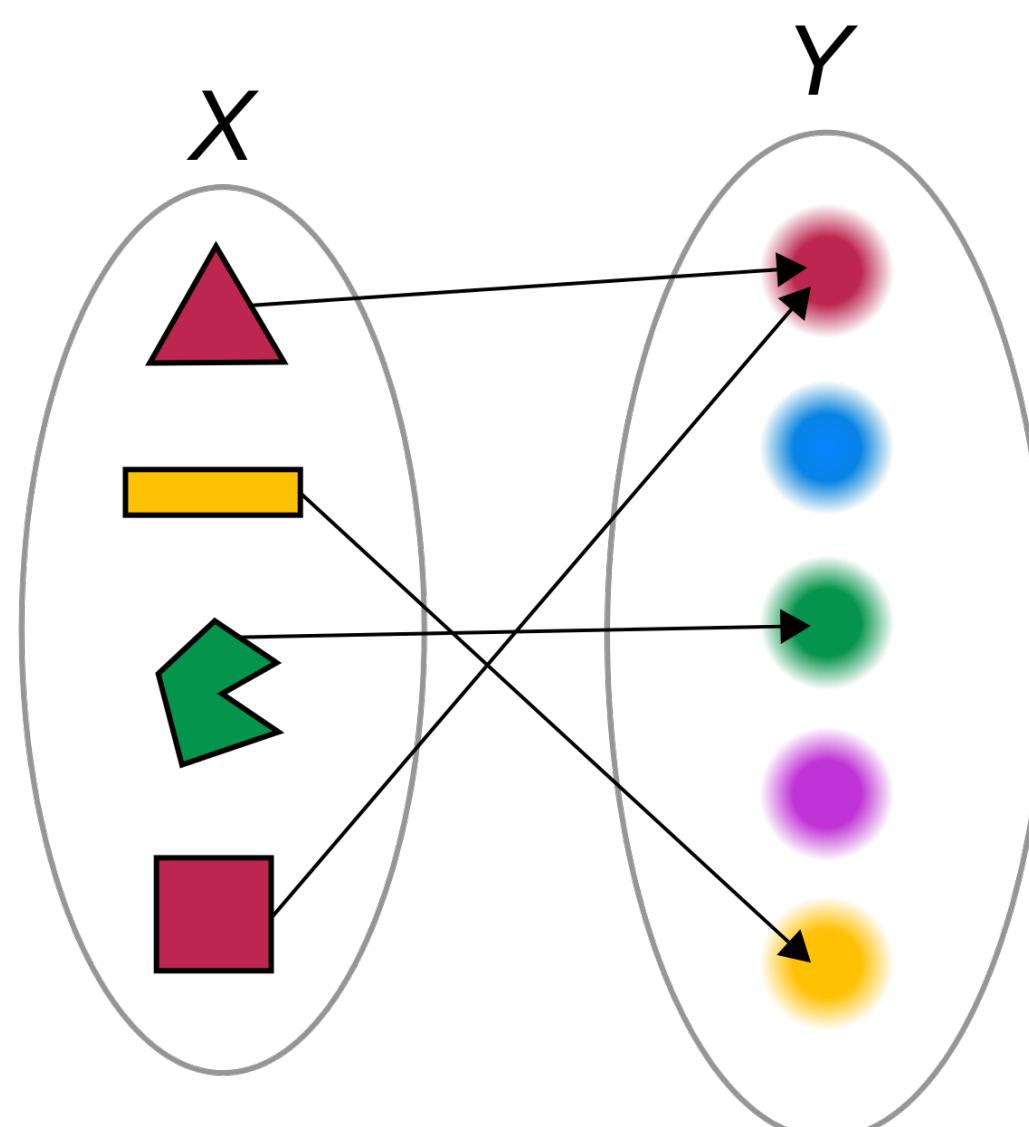
$$u^{-1}S = \{v \in \Sigma^* \mid uv \in S\}$$

DFAs

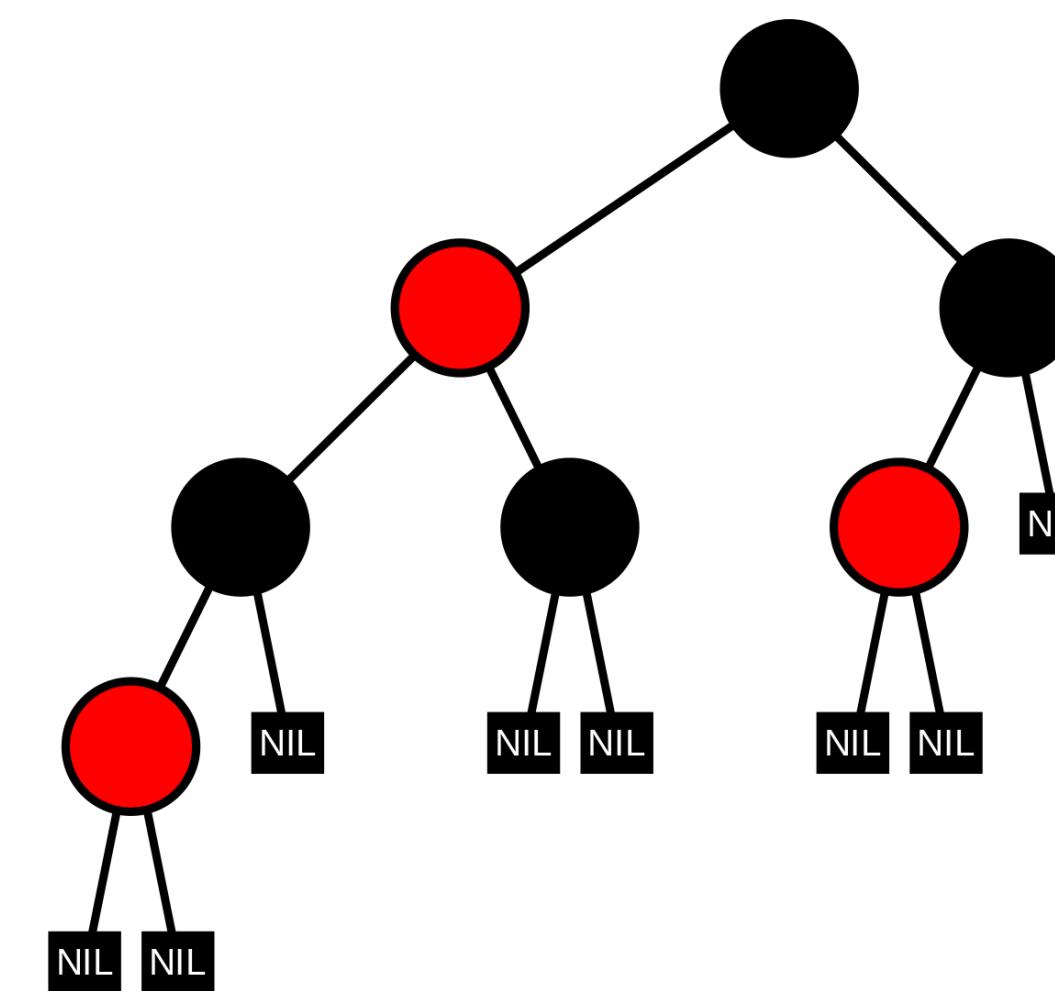


Functional maps

Association lists



Red-Black Trees



Polynomials

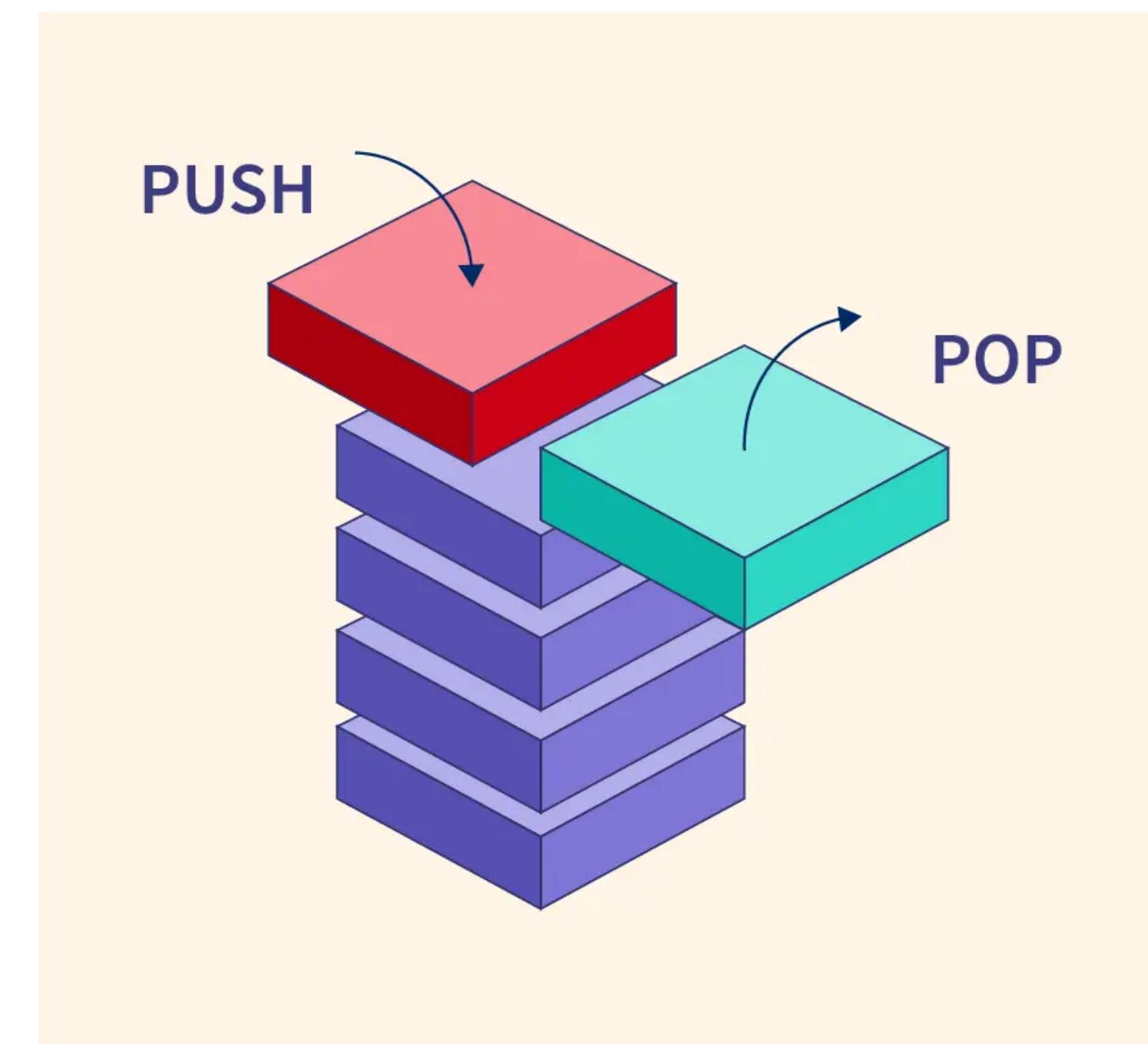
Horner's algorithm

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

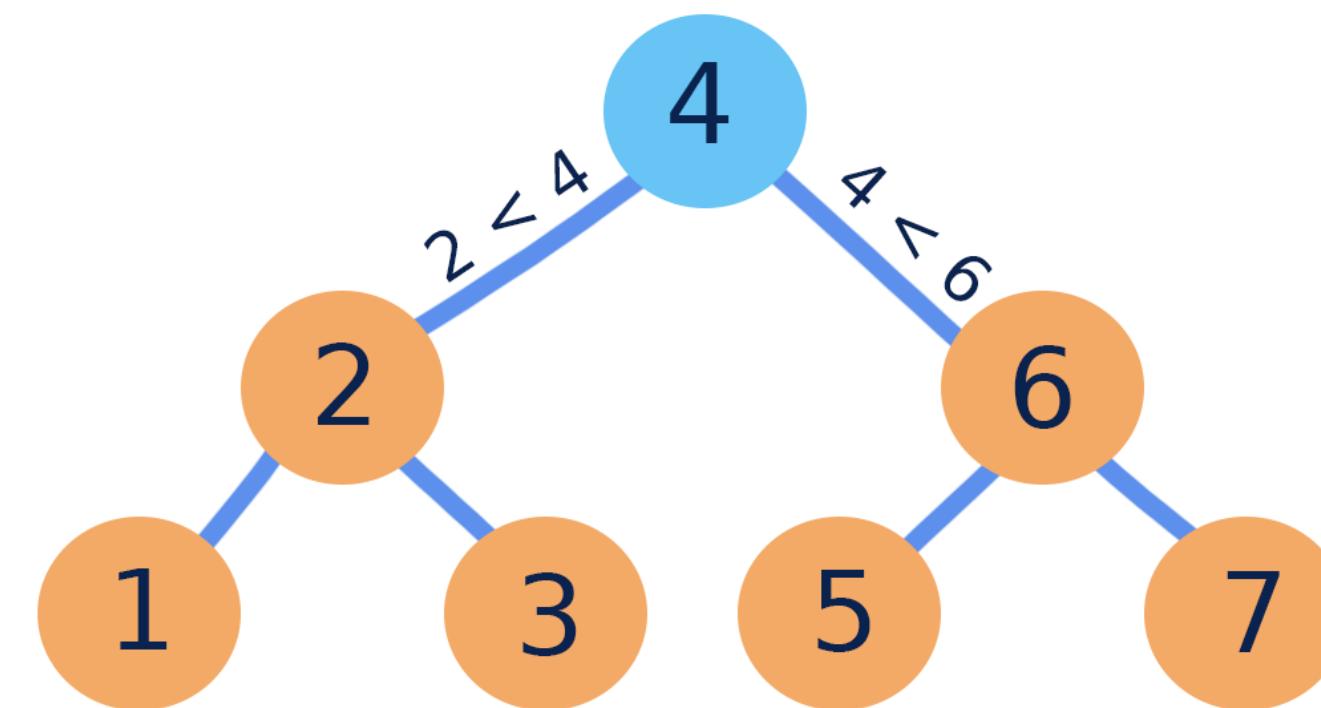
Alternate evaluation algo.

$$\begin{array}{llll} t_3 = a_7x + a_6 & t_2 = a_5x + a_4 & t_1 = a_3x + a_2 & t_0 = a_1x + a_0 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ t_5 = t_3x^2 + t_2 & & & \\ \downarrow & & & \\ t_6 = t_5x^4 + t_4 & & & t_4 = t_1x^2 + t_0 \end{array}$$

Stacks



Sets (BSTs, lists)



FUTURE WORK

Future work

Encode dependencies in generated command sequences

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

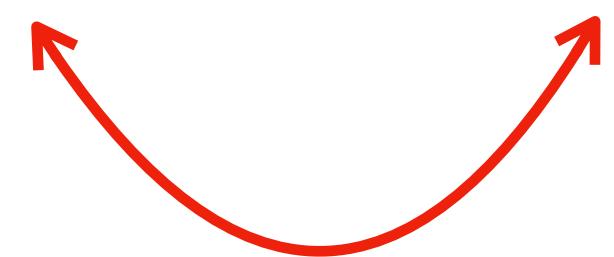


Future work

Encode dependencies in generated command sequences

Handle modules with multiple abstract types

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



```
module type M = struct  
  type 'a t  
  type k  
end
```

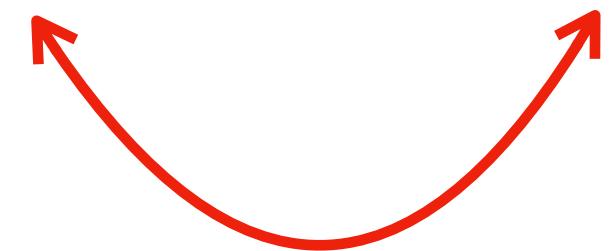
Future work

Encode dependencies in generated command sequences

Handle modules with multiple abstract types

Empirical evaluation

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



```
module type M = struct  
  type 'a t  
  type k  
end
```



Docs: ngernest.github.io/module_pbt

CONTENTS

[Generic utility functions](#)

[Functions for generating PBT code](#)

[Functions for generating the executable for comparing 2 modules](#)

Functions for generating PBT code

`val imports : Base.string -> Base.string -> Base.string -> PPrint.document`

`imports` `filepath` prints out a `PPrint` document that imports the requisite modules for the PBT code. The `sigName`, `modName1`, `modName2` arguments are the names of the module signatures & the two module implementations, which must be the same as their corresponding `.ml` files.

`val sexpAnnotation : PPrint.document`

Document for printing the PPX annotation for S-Expr serialization (indented), followed by a newline

`val isArrowType : ParserTypes.valDecl -> Base.bool`

`isArrowType v` returns true if the value declaration `v` has an arrow type

`val tyIsArrow : ParserTypes.ty -> Base.bool`

`tyIsArrow ty` returns true if `ty` corresponds to an arrow type

`val extractArgTypes : ParserTypes.valDecl -> PPrint.document`

Extracts the argument types of functions defined in the module signature, and generates constructors for the `expr` ADT that take these types as type parameters

**Thanks!
(See you at ICFP!)**

Twitter: @ngernest2

GitHub: github.com/ngernest/module_pbt