



m | C | A

Automated Property-Based Testing for OCaml Modules

Ernest Ng

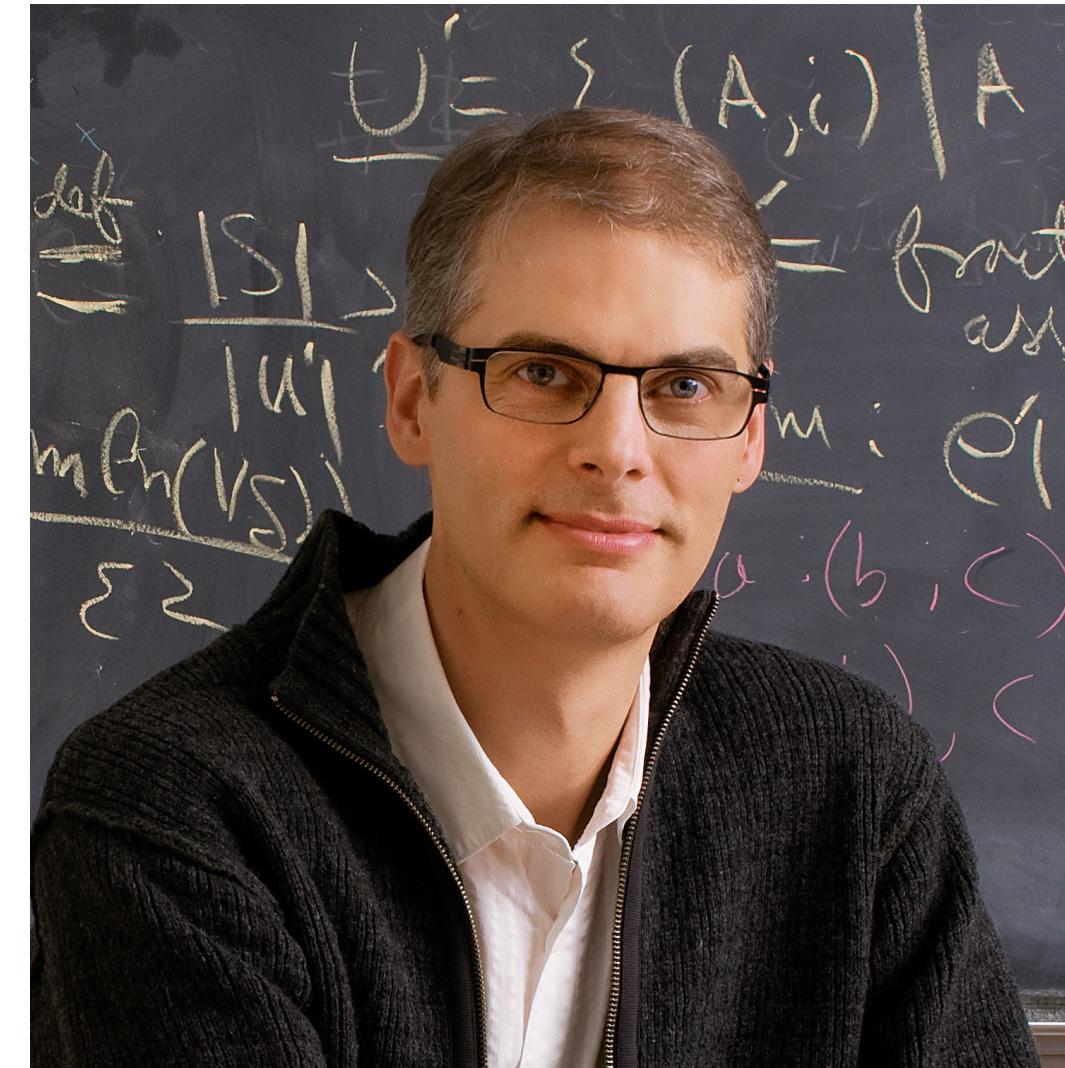
Advised by Harry Goldstein & Benjamin C. Pierce

PLClub, July 28 2023

Acknowledgements



Harry Goldstein



Benjamin C. Pierce

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

Motivation

**modules
matter
most**

- Bob Harper (2011)



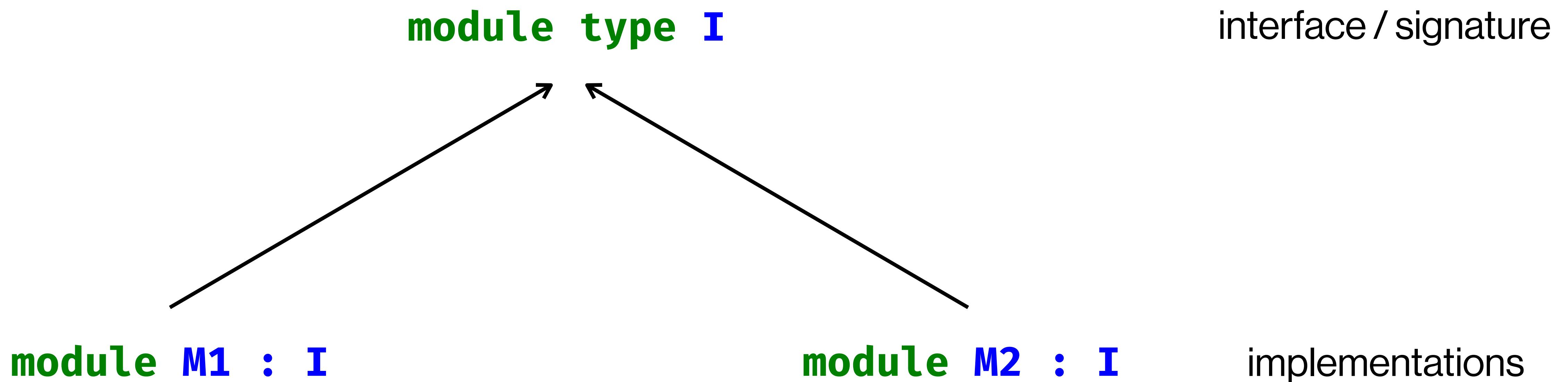
Barbara Liskov,

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

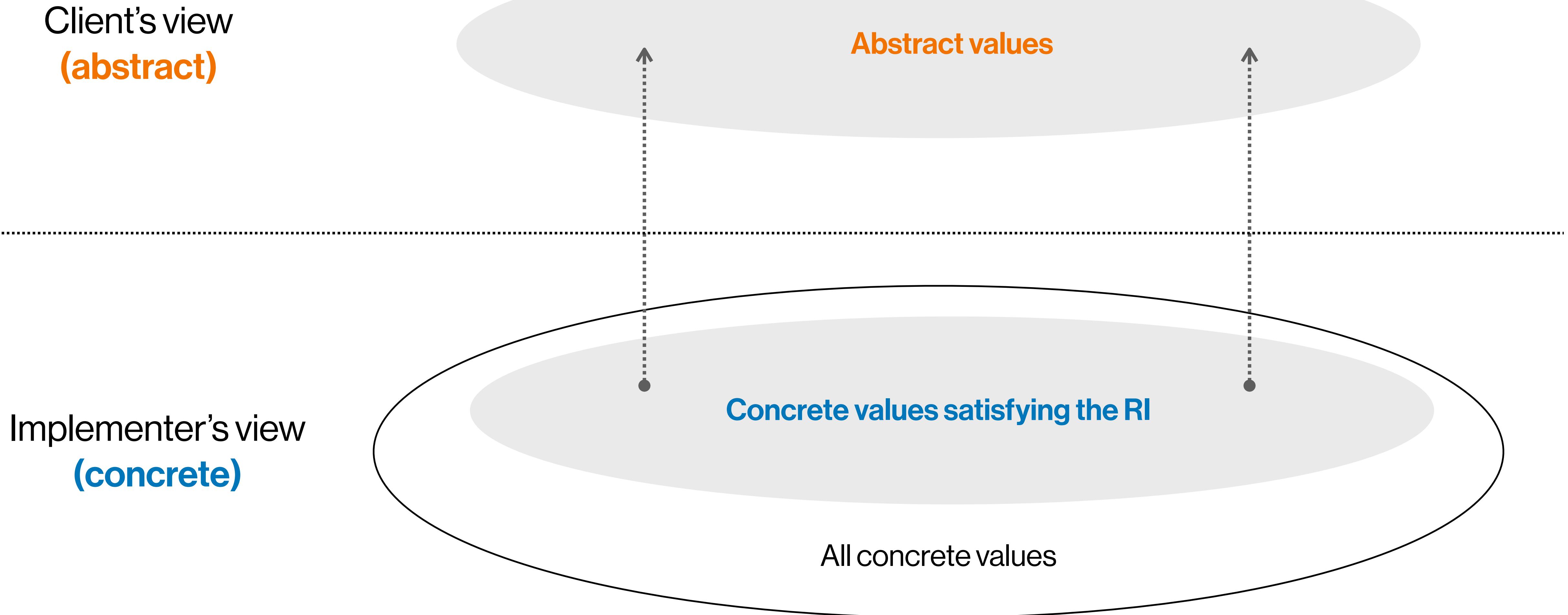


Example: Finite Sets

```
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
```

invariants that can't be captured by
the type system alone

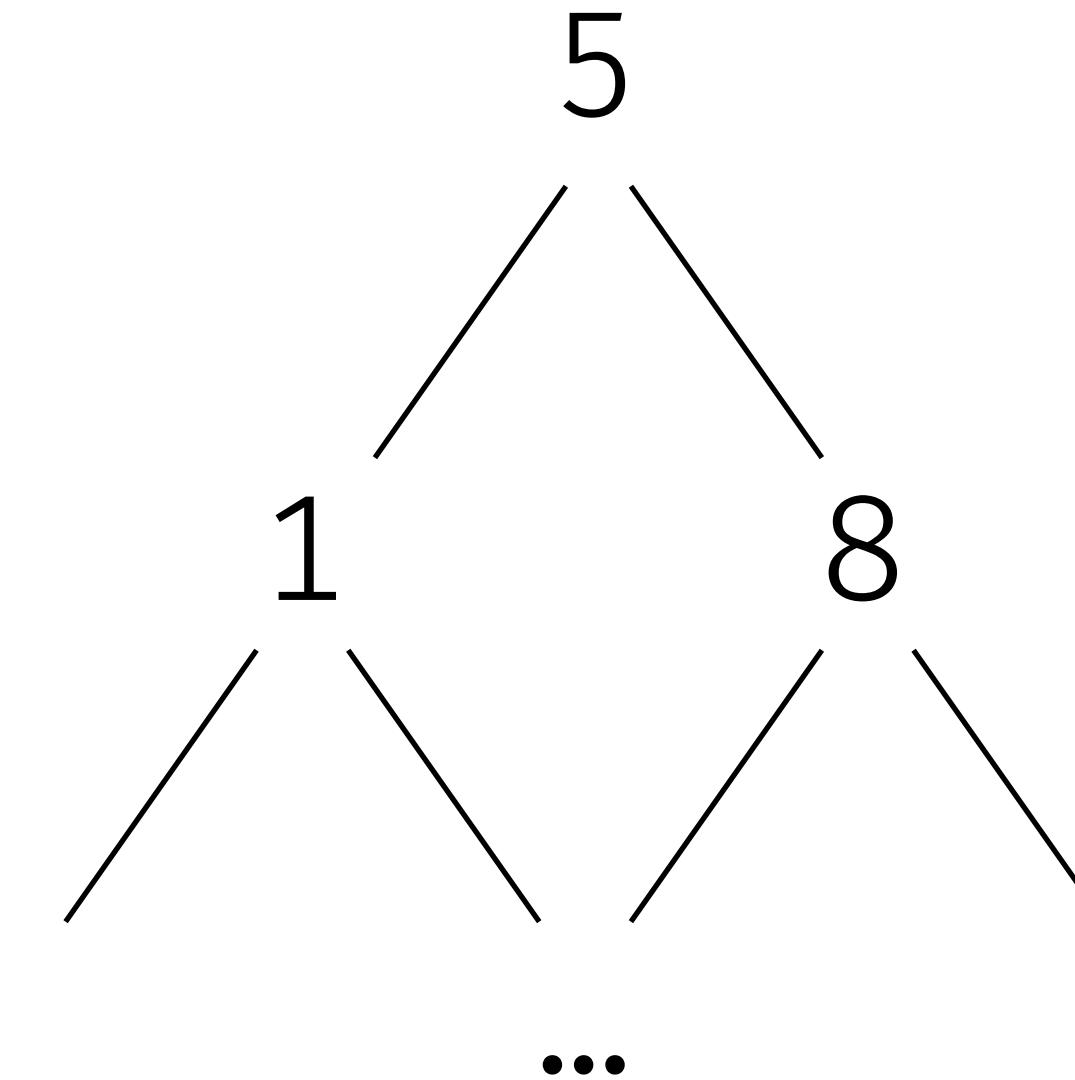
Rep. invariant determines which **concrete** values
are valid representations of **abstract** values



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

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

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



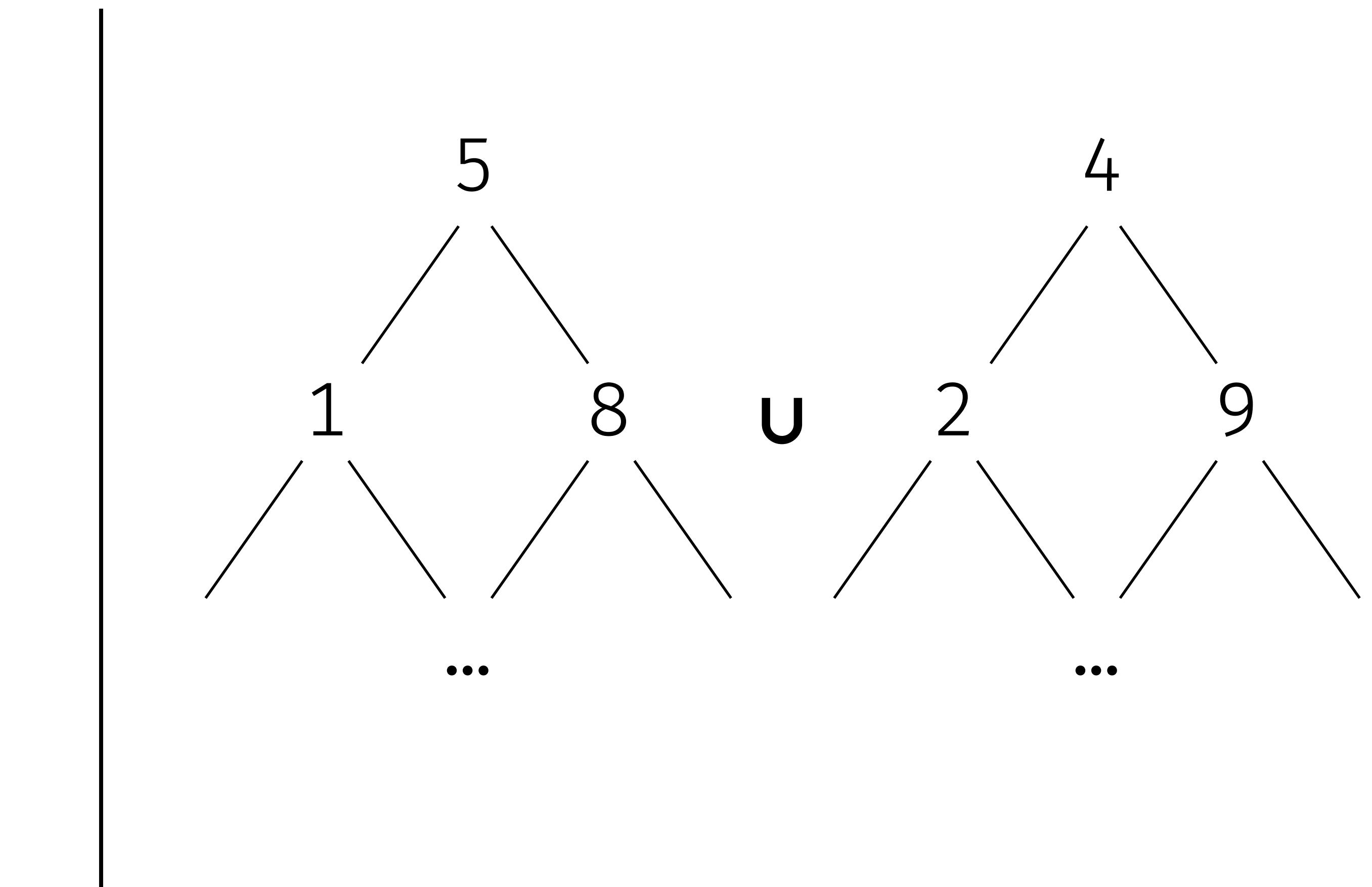
```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

```
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

1. Write a *property*



executable specification
describing desired
program behavior

2. Generate *random inputs*



~~~~~  $x_1 \ x_2 \ \dots \ x_n$

3. Test if random inputs satisfy property

# Property-Based Testing

Popularised by **QuickCheck**

[Claessen & Hughes 2000]



# A property for observational equivalence

Suppose we have a notion of equivalence  $\sim=$  between **Maps** & **AssocLists**:

$(\sim=) : \text{Map.t} \rightarrow \text{AssocList.t} \rightarrow \text{Bool}$  (pseudo-OCaml)

Given two equivalent (**Map**, **AssocList**) pairs  $s \sim= s' \wedge t \sim= t'$ ,

**Map.union**  $s$   $t \sim= \text{AssocList.union } s' t'$

**union of two maps**  $\sim=$  **union of two AssocLists**

## Problem:

How do we generate ***equivalent inputs*** and check if ***outputs*** are ***equivalent***?

# Naïvely checking observational equivalence

(pseudo-code)

Suppose modules  $M_1, M_2$  implement the same signature:

```
x1 ← M1.genInput  
x2 ← M2.genInput  
guard (equivalentInputs x1 x2)  
  
let y1 = M1.f x1  
y2 = M2.f x2  
  
assert (equivalentOutputs y1 y2)
```

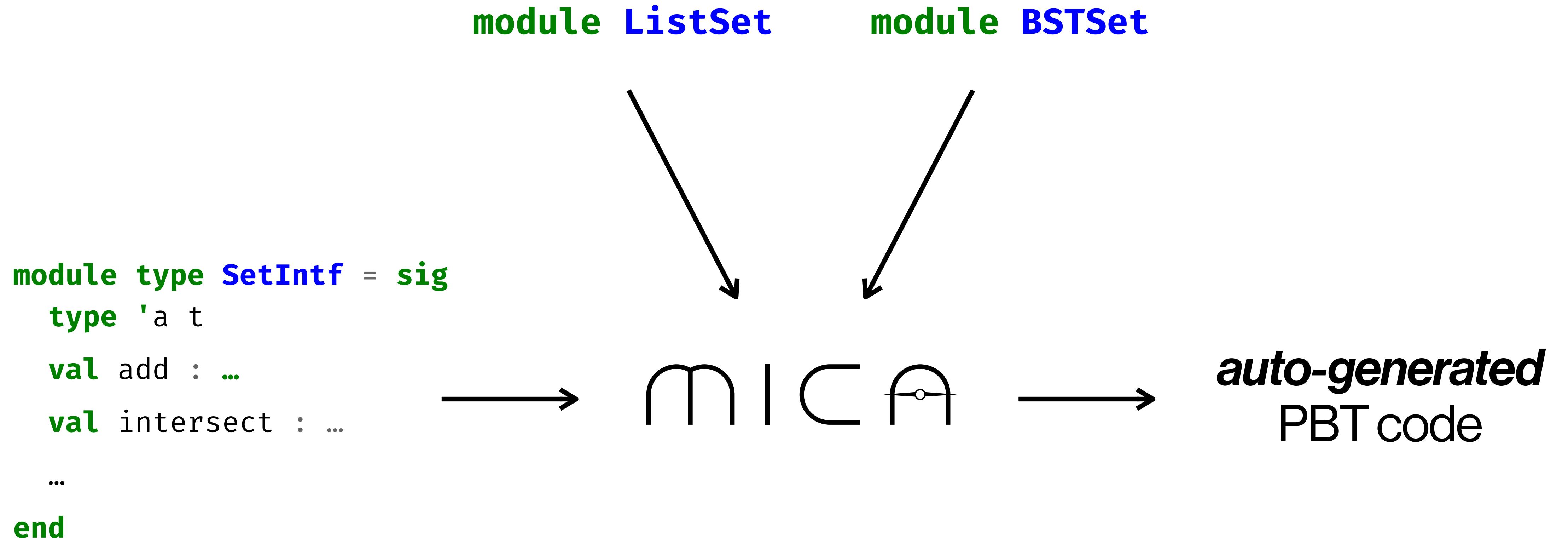
**In general,  $x_1$  and  $x_2$  may not be equivalent as they are independently generated!**

Idea:

What if we generate  
***symbolic expressions*** instead?

m | C A

# m | C A overview



# Examining the *automatically generated* PBT code



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 expressions

# The **expr** type

**expressions** = inductively-defined symbolic expressions

```
module type SetIntf = sig
```

```
  type 'a t
```

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

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

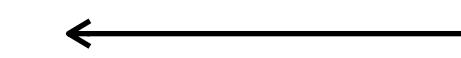
```
  val union: 'a t → 'a t → 'a t
```

```
  val size: 'a t → int
```

```
  val is_empty : 'a t → bool
```

```
  ...
```

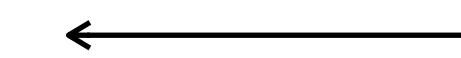
```
end
```



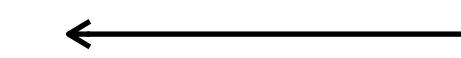
```
type expr =  
| Empty
```



```
| 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
```



MICΑ automatically produces:

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

**val** gen\_expr : ty → expr **Generator.t**

`gen_expr T`

well-typed symbolic expressions  
that return type **T**

`gen_expr T`

well-typed symbolic expressions  
that return type **T**

**Intersect (Add 2 Empty) Empty**



`gen_expr T`

well-typed symbolic expressions  
that return type **T**

**Intersect (Add 2 Empty) Empty**



**Is\_empty (Size Empty)**



# 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; ... ]
  ...

```



MICAH automatically produces:

***Interpreter*** for symbolic expressions

**val** interp : expr → value

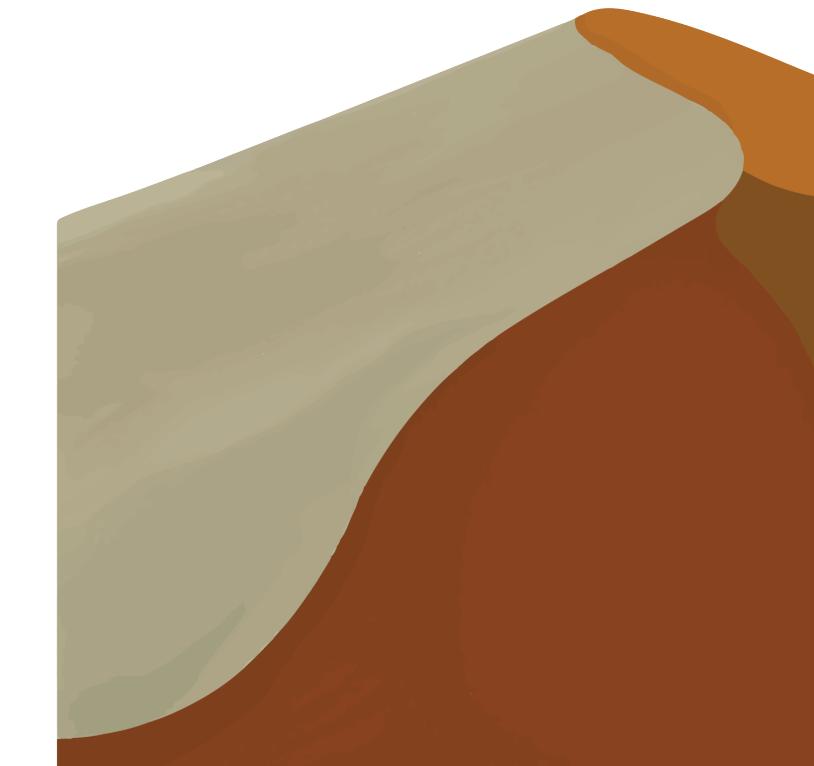
# Interpreter for expr's

```
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
```

**MICA** automatically produces:

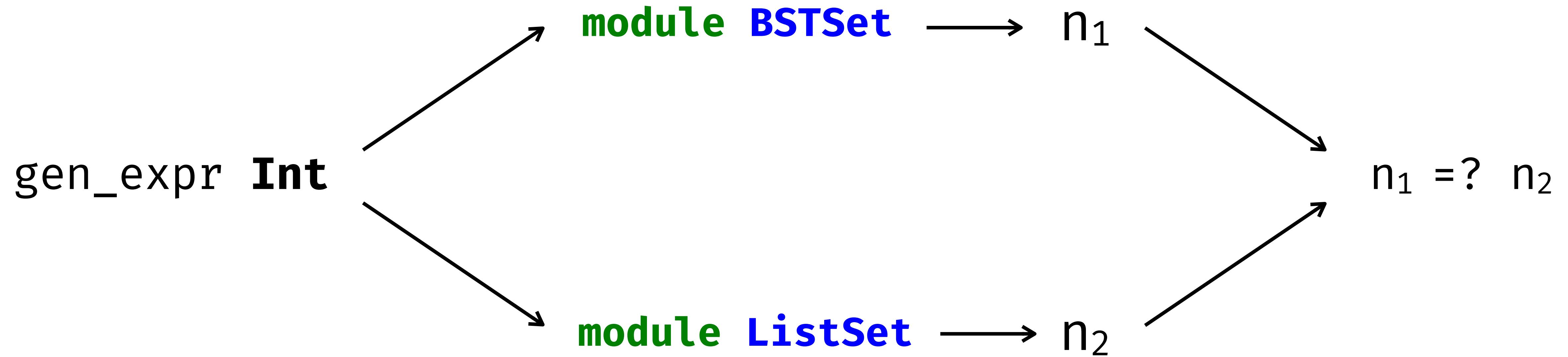
**Executable** for testing observational equivalence



**DUNE**

# **Executable** for testing observational equivalence

Compares the result of symbolic expressions  
at **observable concrete types**



# Generator

generate **random**  
symbolic expressions

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

# Generator

generate **random**  
symbolic expressions

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

# Interpreter

interpret expressions  
over modules

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

# Generator

generate **random**  
symbolic expressions

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

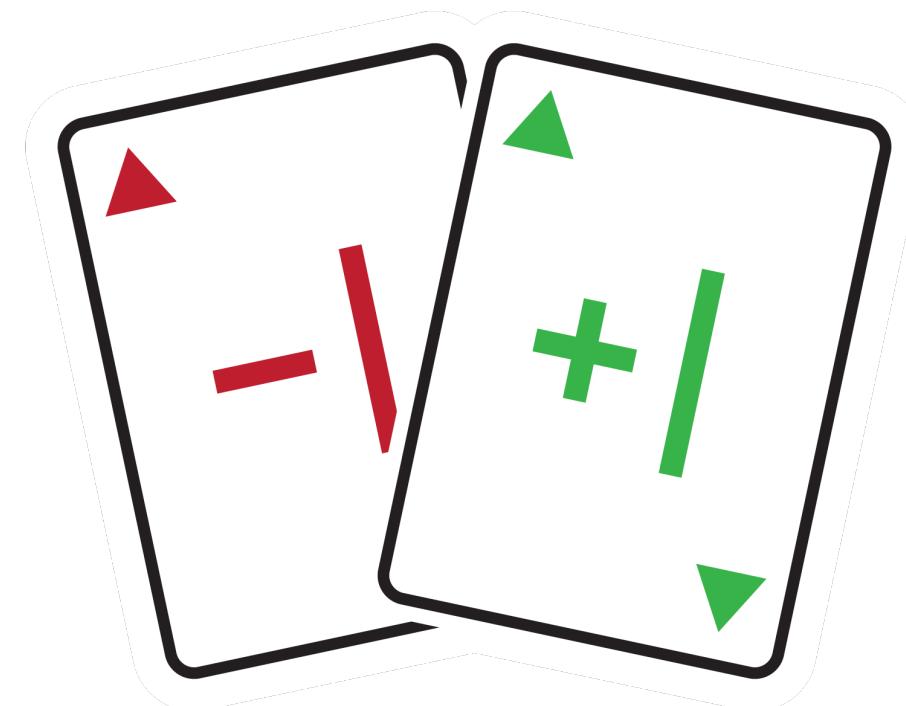
# Interpreter

interpret expressions  
over modules

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

# Executable

test for  
observational  
equivalence





Jane Street



**CORE\_QUICKCHECK**



**CORE**

# **Parser**

## **inhabitedtype/ angstrom**



Parser combinators built for speed and memory efficiency

# **Code generator**

## **fpottier/pprint**

A pretty-printing combinator library for OCaml



# Demo

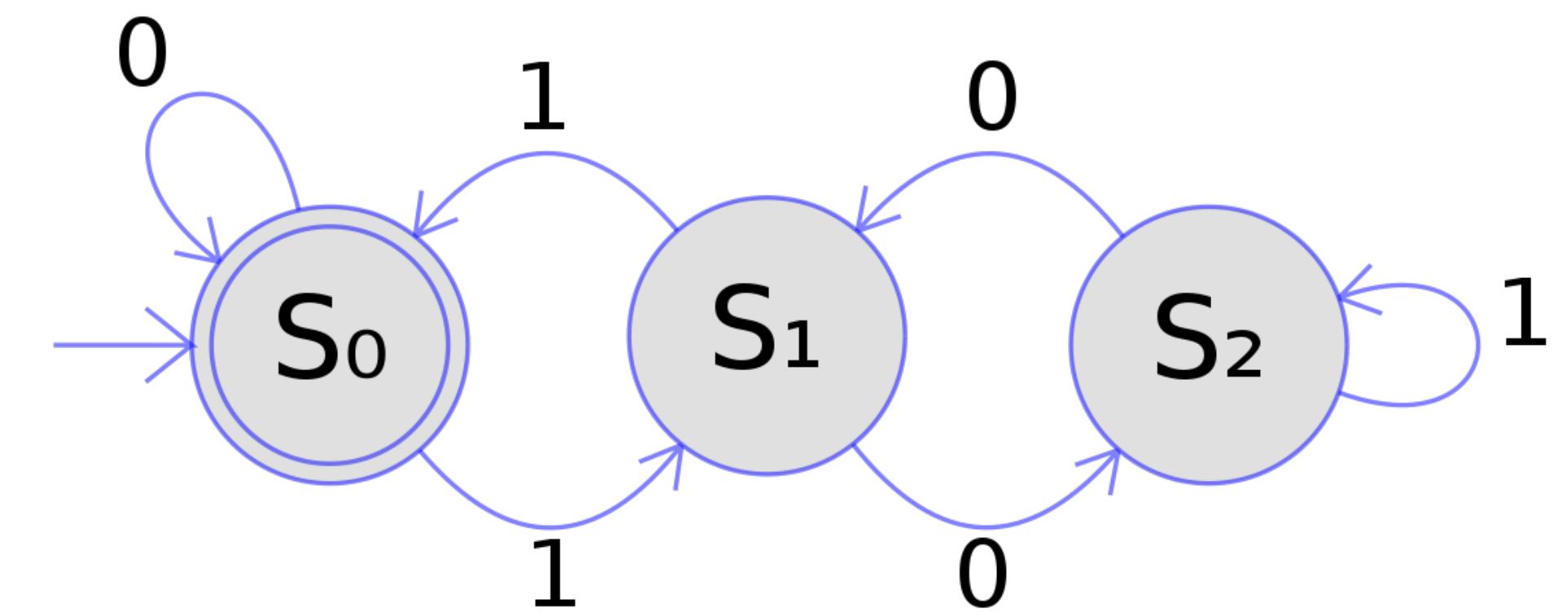
# **Case studies**

# Regex matching

**Brzozowski derivatives**

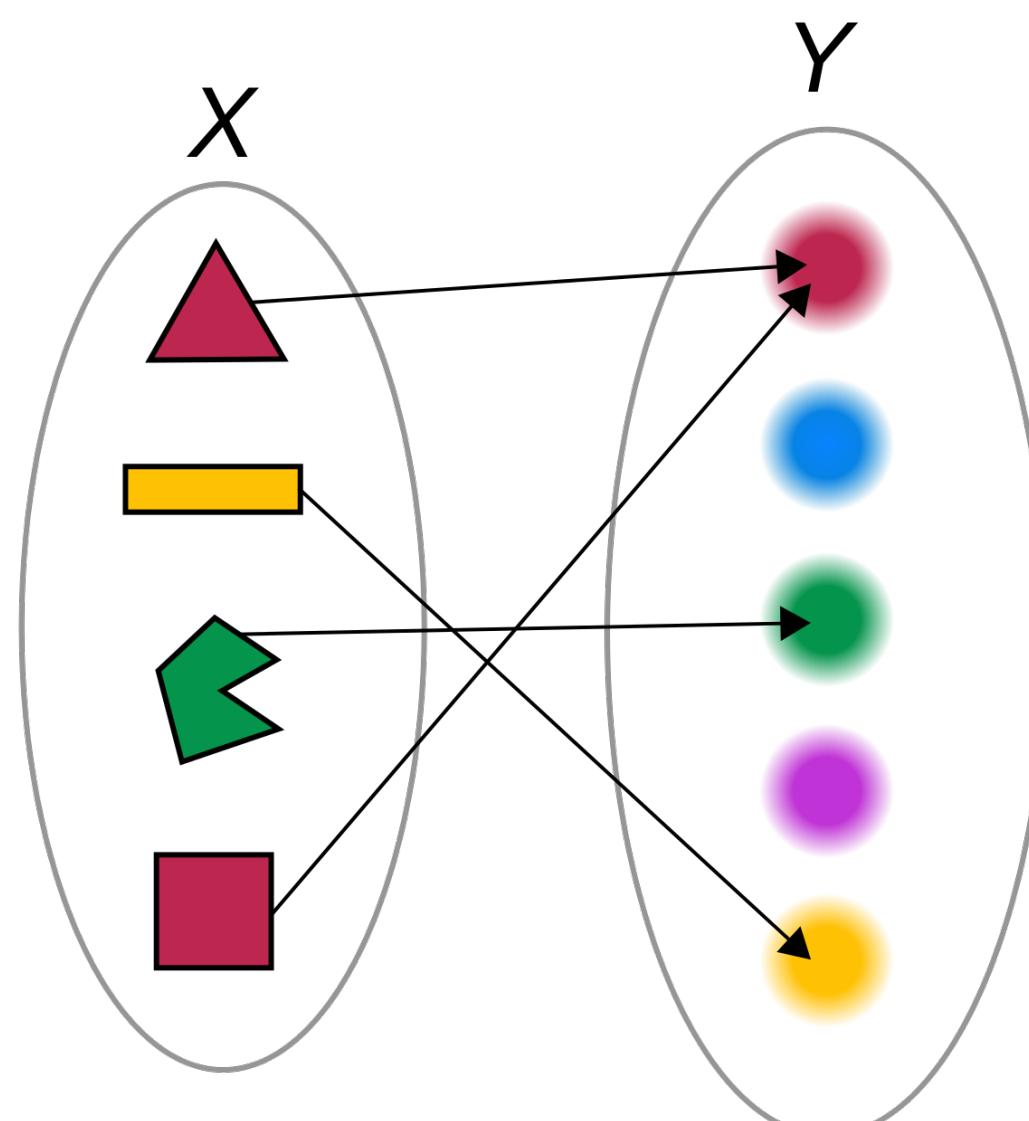
$$u^{-1}S = \{v \in \Sigma^* \mid uv \in S\}$$

**DFA**s

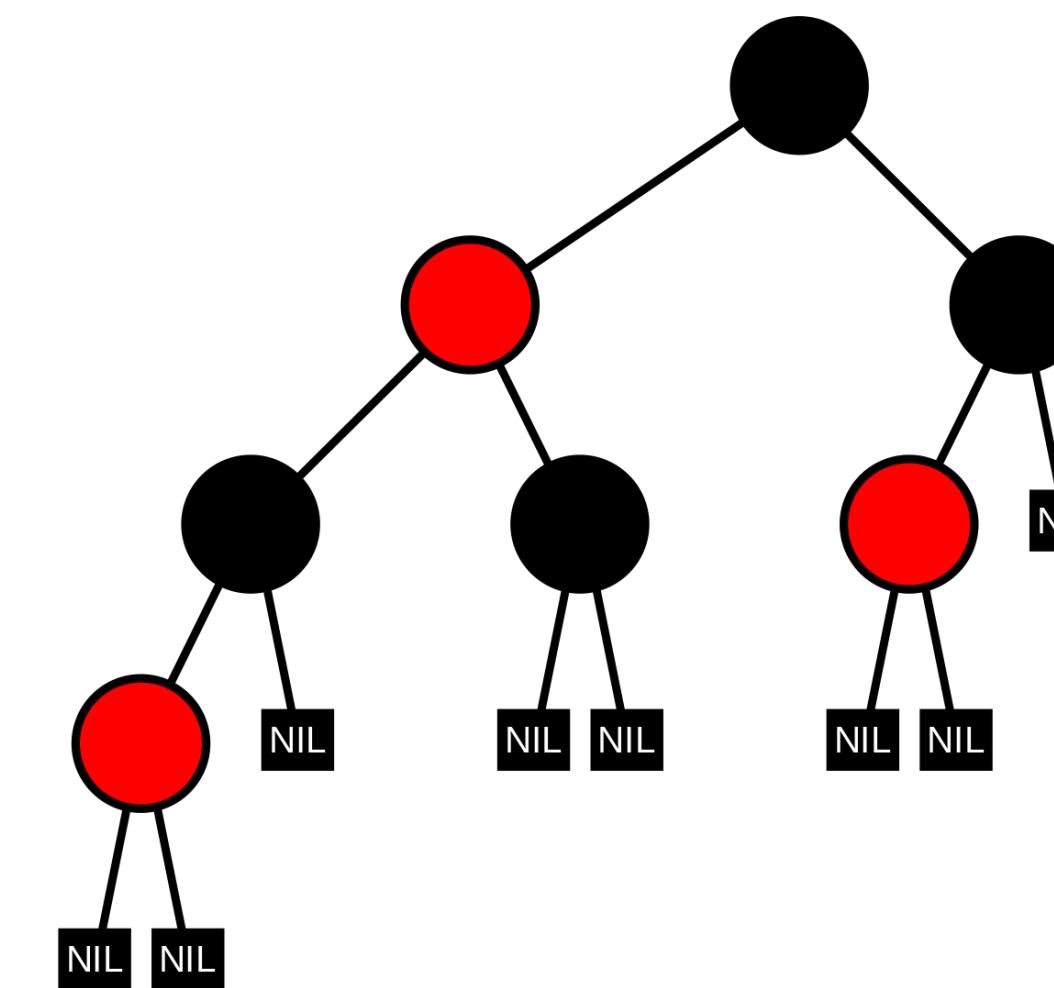


# Functional maps

## Association lists



## Red-Black Trees

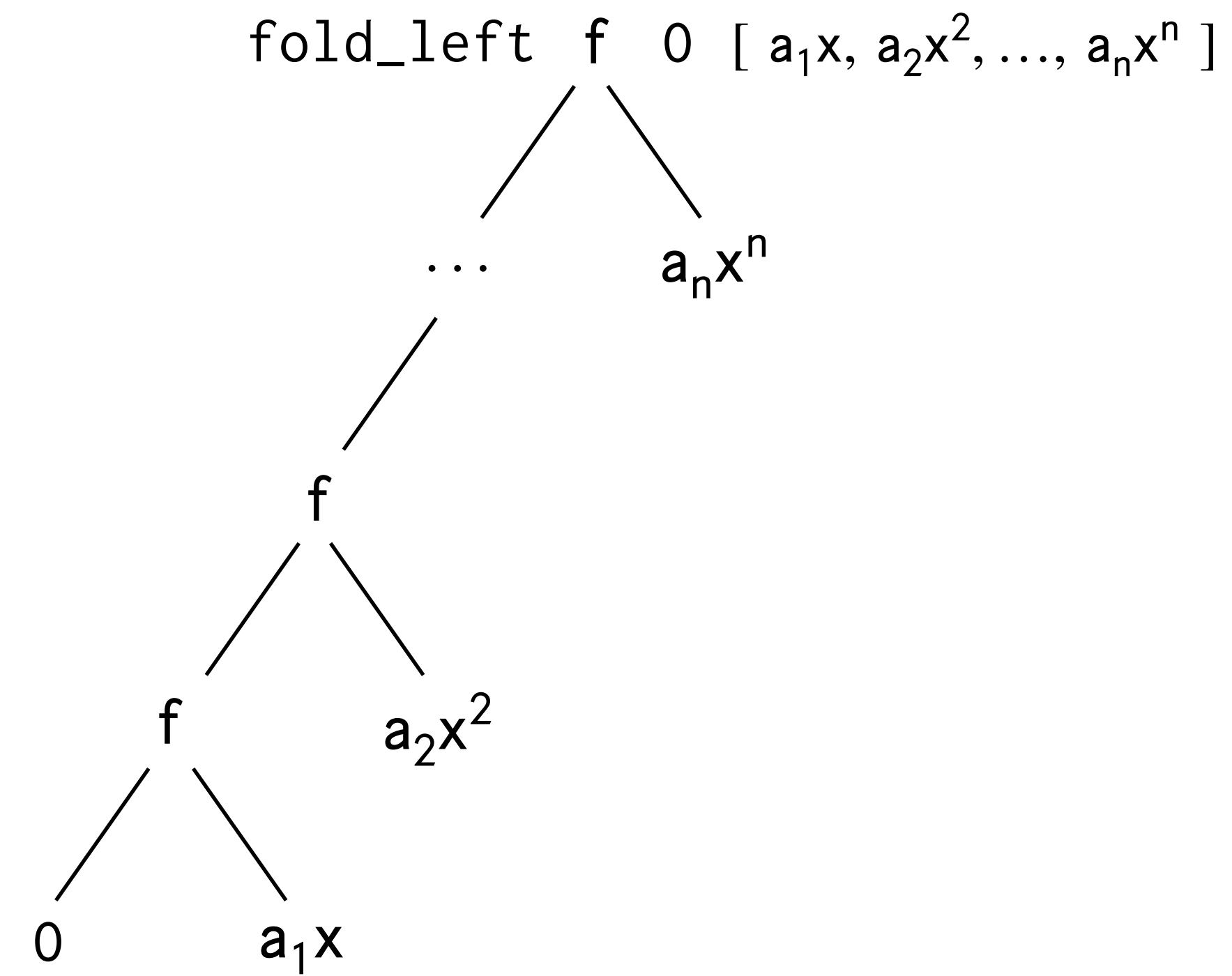


# Polynomials

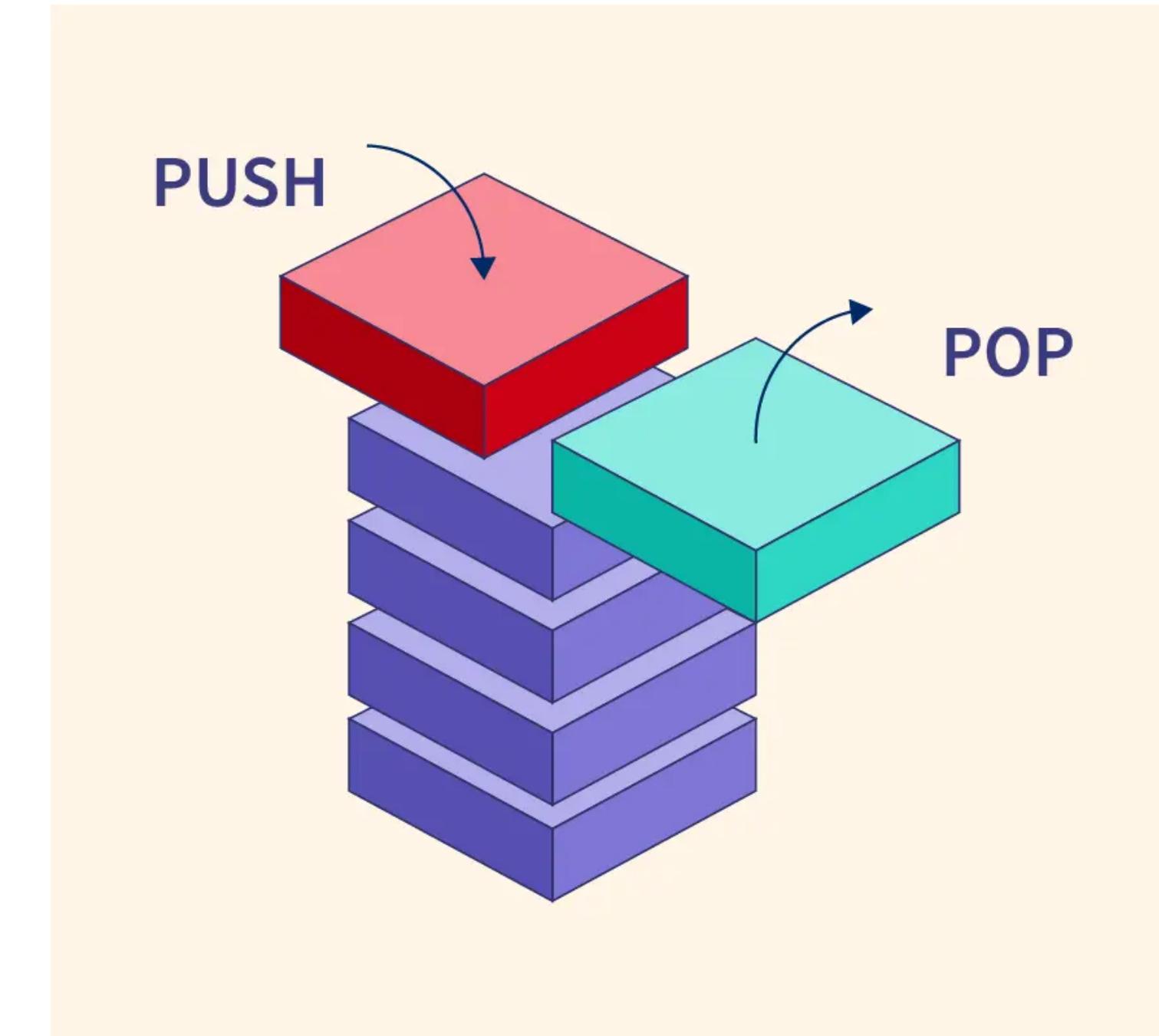
## Horner's algorithm

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

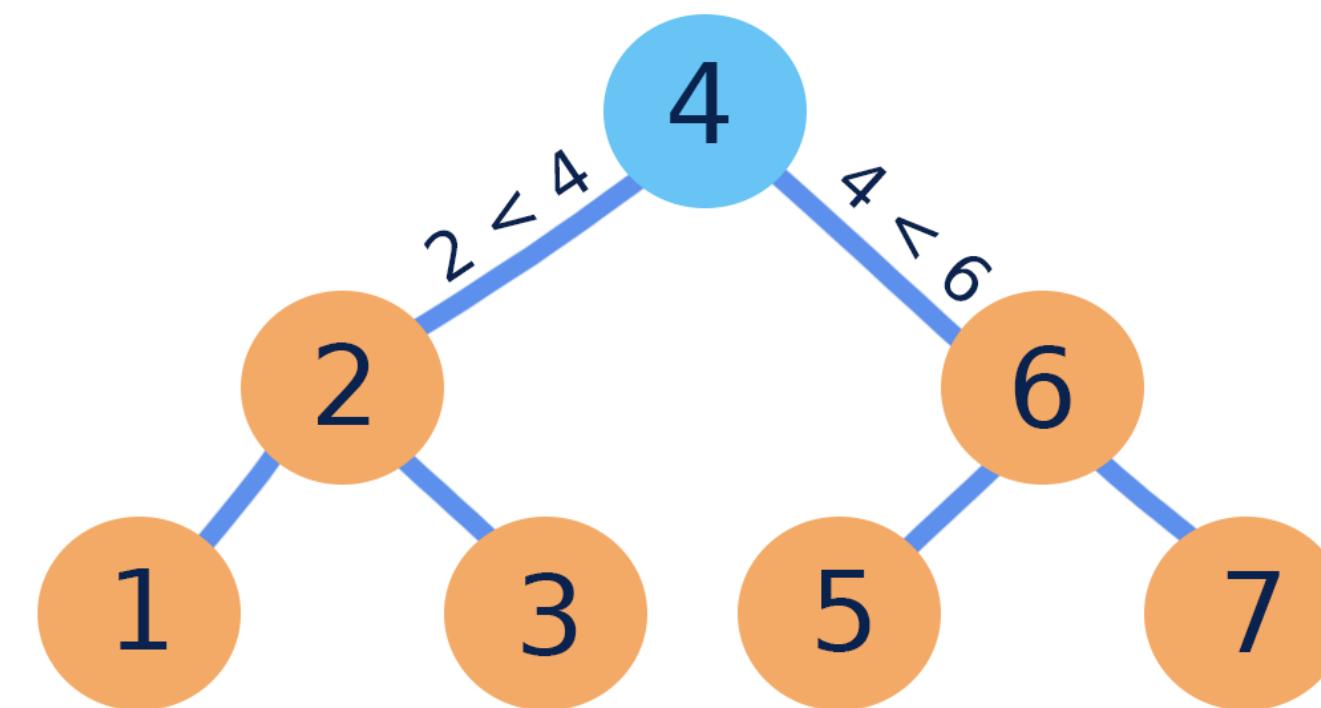
## Fold over list of monomials



# Stacks

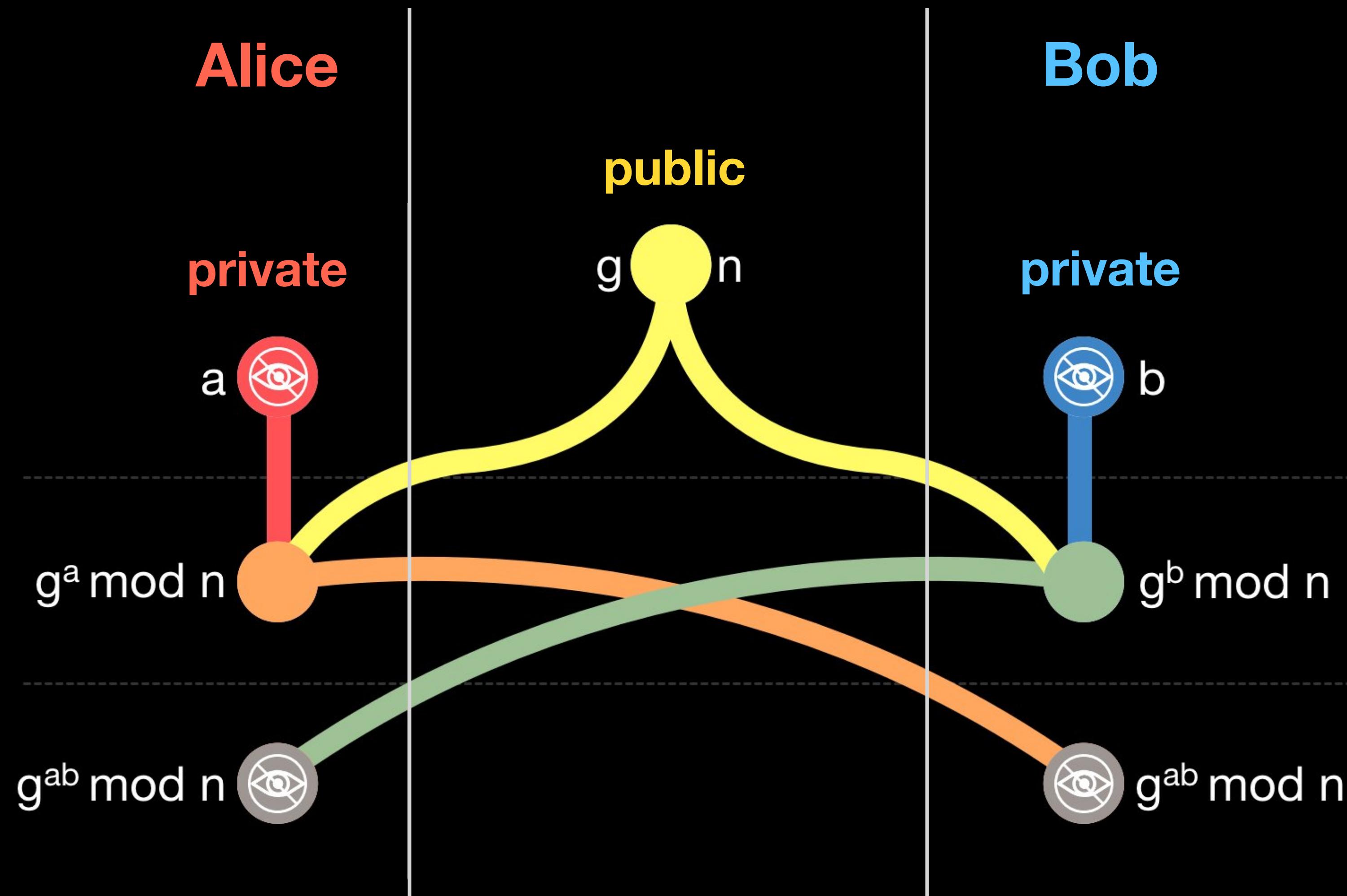


# Sets (BSTs, lists)



# Diffie-Hellman key exchange

(work in progress)



# Challenges with testing Diffie-Hellman

# Challenges with testing Diffie-Hellman

```
module type Set = sig
  type 'a t
  ...
end
```

Other examples

*one abstract type*

DH interface

*two abstract types*

```
module type DH = sig
  type public_key
  type private_key
  ...
end
```

# Challenges with testing Diffie-Hellman

```
module type Set = sig
  type 'a t
  ...
end
```

```
emptySet    : 'a t
getAbsType : ? → 'a t
```

Other examples

**one abstract type**

interface tells us how to  
to obtain an '**a t**'

DH interface

**two abstract types**

```
module type DH = sig
  type public_key
  type private_key
  ...
end
```

no way to get a **private\_key**  
using only the interface

# Challenges with testing Diffie-Hellman

```
module type Set = sig
  type 'a t
  ...
end
```

```
emptySet    : 'a t
getAbsType : ? → 'a t
```

Add  
Union  
Size  
...  
~~~~~ Set ~~~~ Add  
Union
Size
...

Other examples

one abstract type

interface tells us how to
to obtain an '**'a t**'

no prescribed order
for calling functions

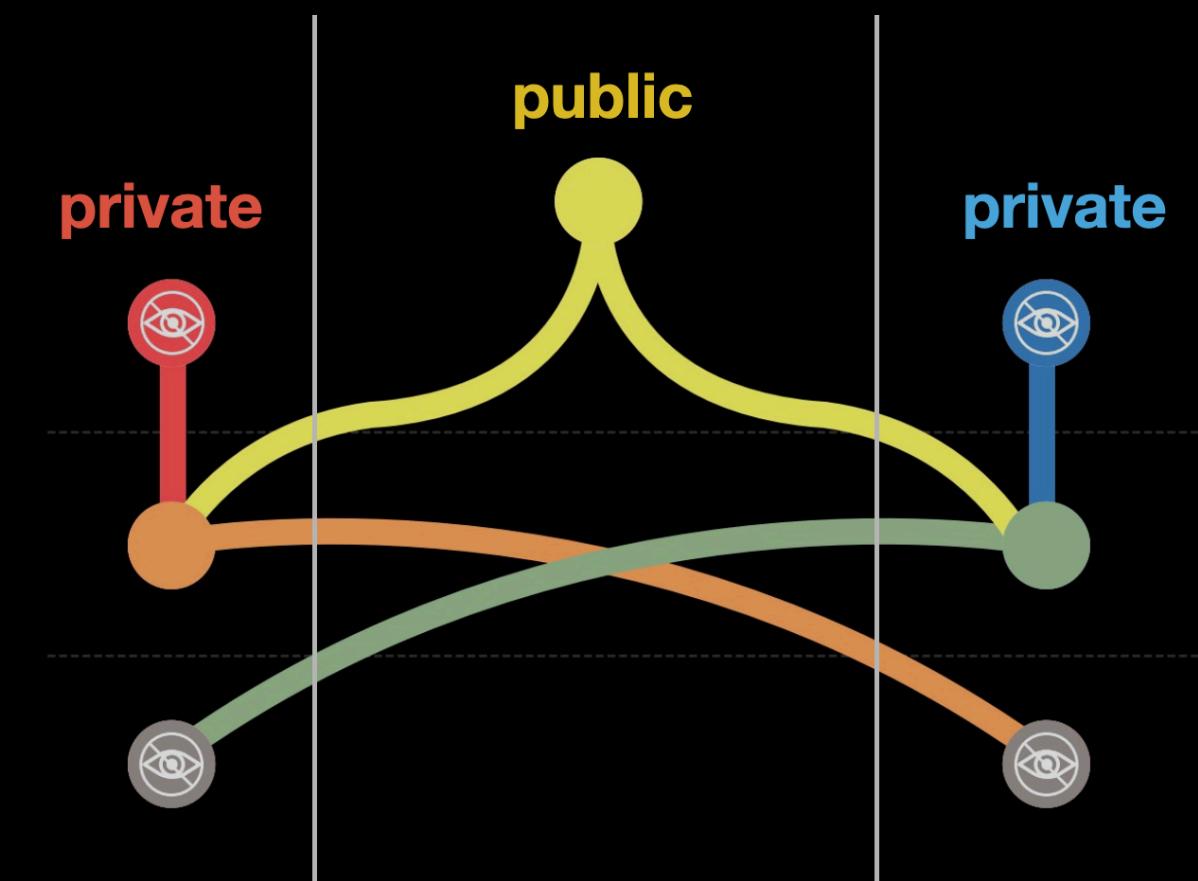
DH interface

two abstract types

no way to get a **private_key**
using only the interface

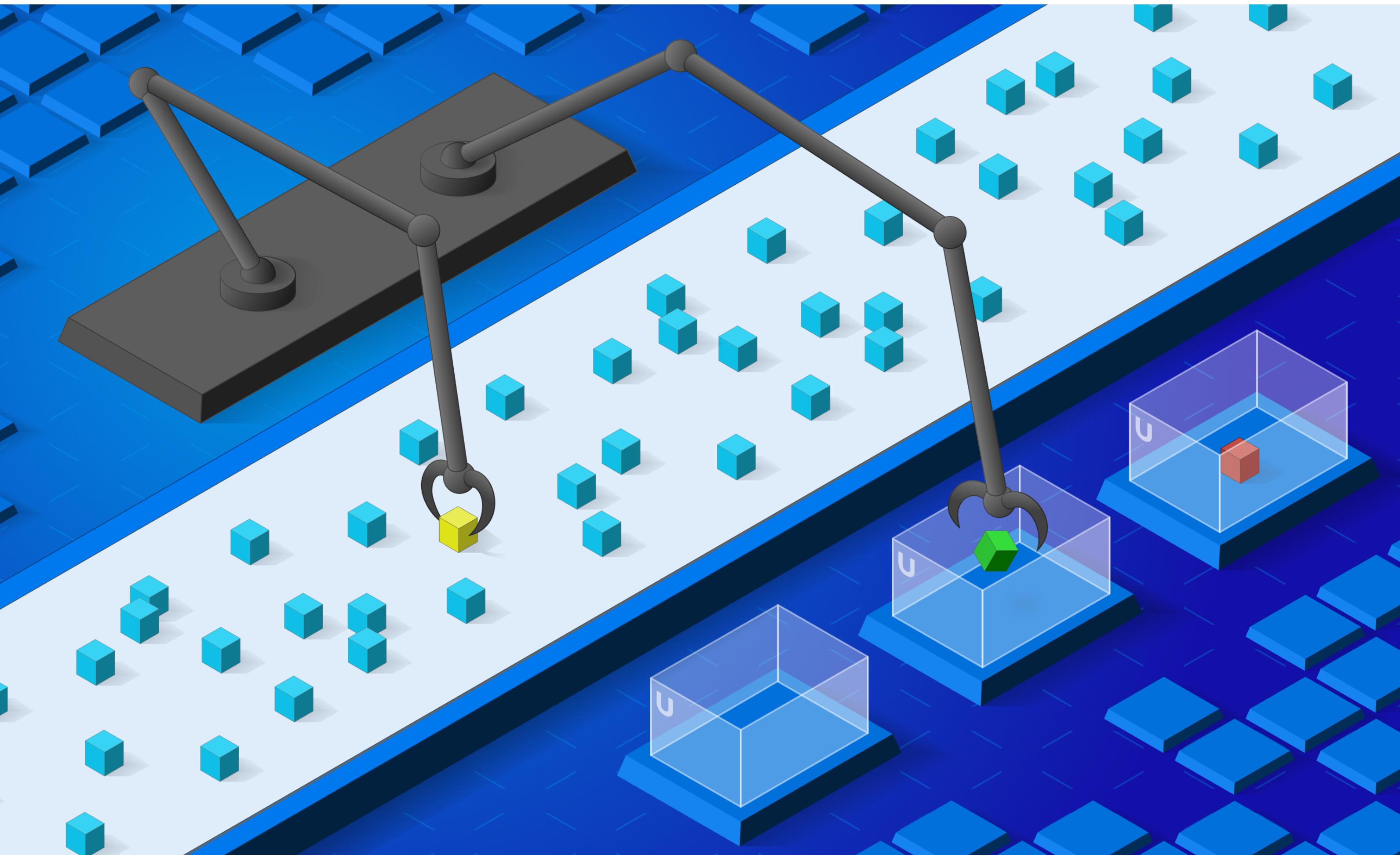
prescribed order
for calling functions
(specified by DH)

```
module type DH = sig
  type public_key
  type private_key
  ...
end
```



Catching bugs with Mica

(work in progress)



Catching bugs

(work in progress)

- Total of 15 artificial bugs inserted in { Set, Stack, Polynomial } modules
- 14 bugs caught

Experimental limitations:

- Bugs were introduced one at a time
- In each trial, the bug was only introduced in one module
(other module remained correct)

Bug: Not enforcing Set invariant for lists

Artificial bug

```
let union s1 s2 =  
  s1 @ s2 (* ▷ dedup *)
```

Bug: Not enforcing Set invariant for lists

| Artificial bug | Failing test case |
|--|-------------------------|
| <pre>let union s1 s2 =
 s1 @ s2 (* ▷ dedup *)</pre> | Invariant [-3; 5; 5; 9] |

Bug: Not enforcing Set invariant for lists

| Artificial bug | Failing test case | Correct result |
|--|-------------------------|-----------------------------------|
| <pre>let union s1 s2 =
 s1 @ s2 (* ▷ dedup *)</pre> | Invariant [-3; 5; 5; 9] | Invariant
[-3; 5; 9]
= TRUE |

Bug: Not enforcing Set invariant for lists

| Artificial bug | Failing test case | Correct result | Erroneous result |
|--|-------------------------|-----------------------------------|---------------------------------------|
| <pre>let union s1 s2 =
 s1 @ s2 (* ▷ dedup *)</pre> | Invariant [-3; 5; 5; 9] | Invariant
[-3; 5; 9]
= TRUE | Invariant
[-3; 5; 5; 9]
= FALSE |

Bug in Stack implementation

Artificial bug

```
push x stack = stack @ [x]
```

Bug in Stack implementation

Artificial bug

```
push x stack = stack @ [x]
```

Failing test case

```
Peek (Push -8  
      (Push 3  
      Empty))
```

Bug in Stack implementation

| Artificial bug | Failing test case | Correct result |
|----------------------------|-------------------------------------|----------------|
| push x stack = stack @ [x] | Peek (Push -8
(Push 3
Empty)) | -8 |

Bug in Stack implementation

| Artificial bug | Failing test case | Correct result | Erroneous result |
|----------------------------|-------------------------------------|----------------|------------------|
| push x stack = stack @ [x] | Peek (Push -8
(Push 3
Empty)) | -8 | 3 |

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

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

not caught

Technically, [rem x lt] doesn't do anything here
(it's not a bug)

Bug caught

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 expressions
- 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

Differential Testing

(very big field)



CSmith
(unguided, tests C compilers)

[Yang et al. 2011]



NEZHA
(guided, uses coverage-guided fuzzing)

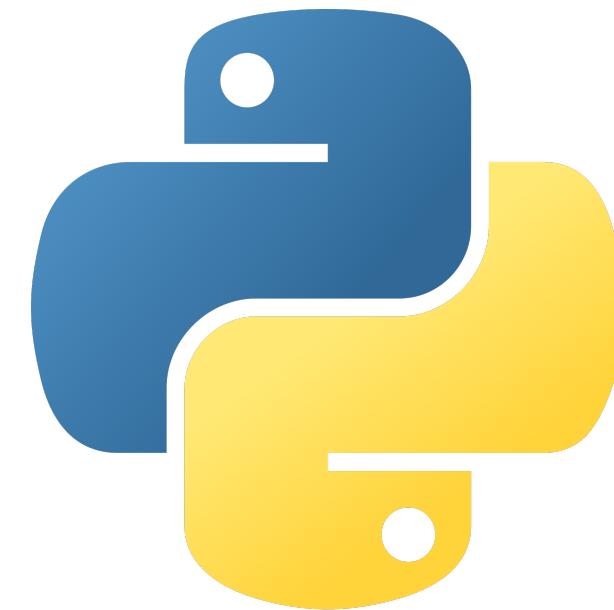
[Petsios et al. 2017]



Efftester
(PBT-based differential testing
of OCaml compiler backends)

[Midtgaard et al. 2017]

Automatic generation of PBT code

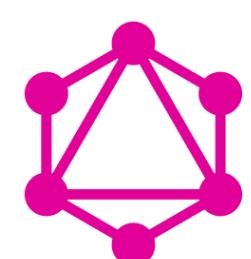


Hypothesis
Ghostwriter

[Hatfield-Dodds et al. 2020]



Clojure



GraphQL

QuickREST

[Karlsson et al. 2020, 2019]

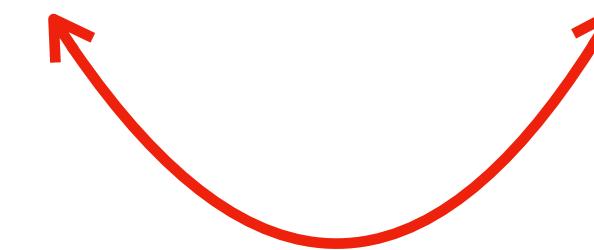
Future work

Future work

Encode dependencies
in generated symbolic expressions

Set example

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

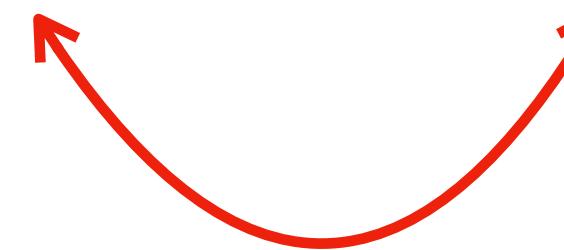


Future work

Encode dependencies
in generated symbolic expressions

Set example

Rem 2 (Add 2 Empty)



Regex example

do

re \leftarrow genRegex

s \leftarrow genRegexString re

Future work: Handle imperative code



Questions:

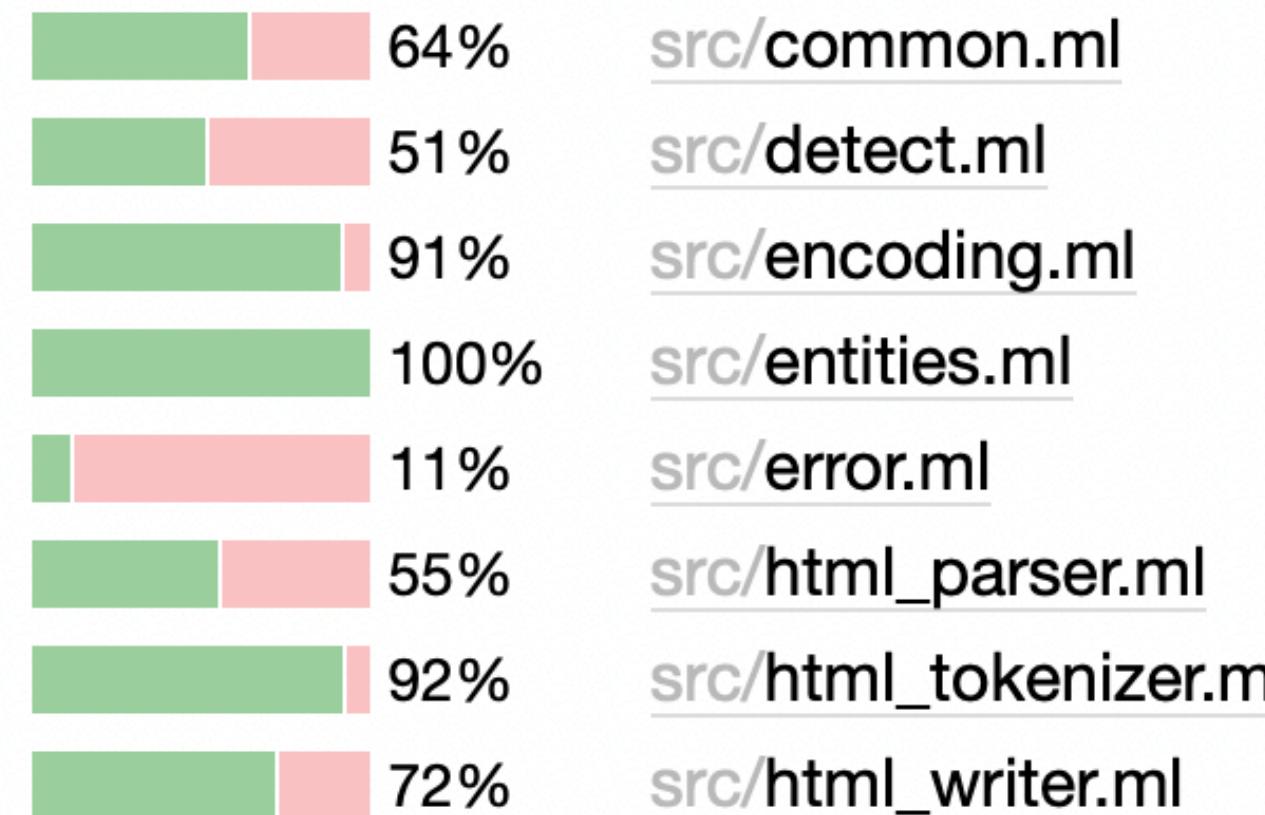
- How should we keep track of state in the generator for symbolic expressions?
- How do we deal with exceptions / effects?

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

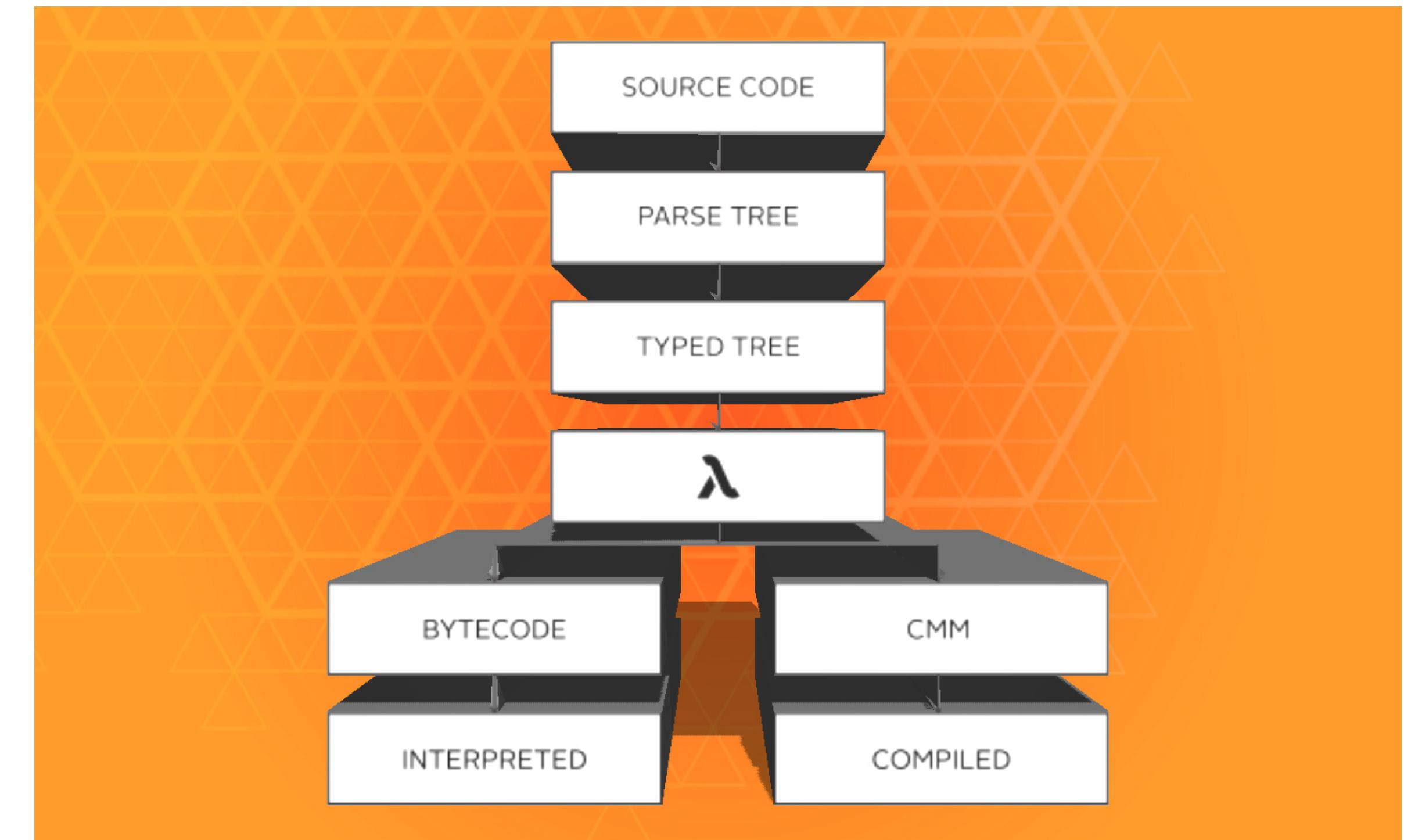
Shrinking

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



Add **5** Empty

Integrate with OCaml compiler



Graphic from [Jane Street](#)

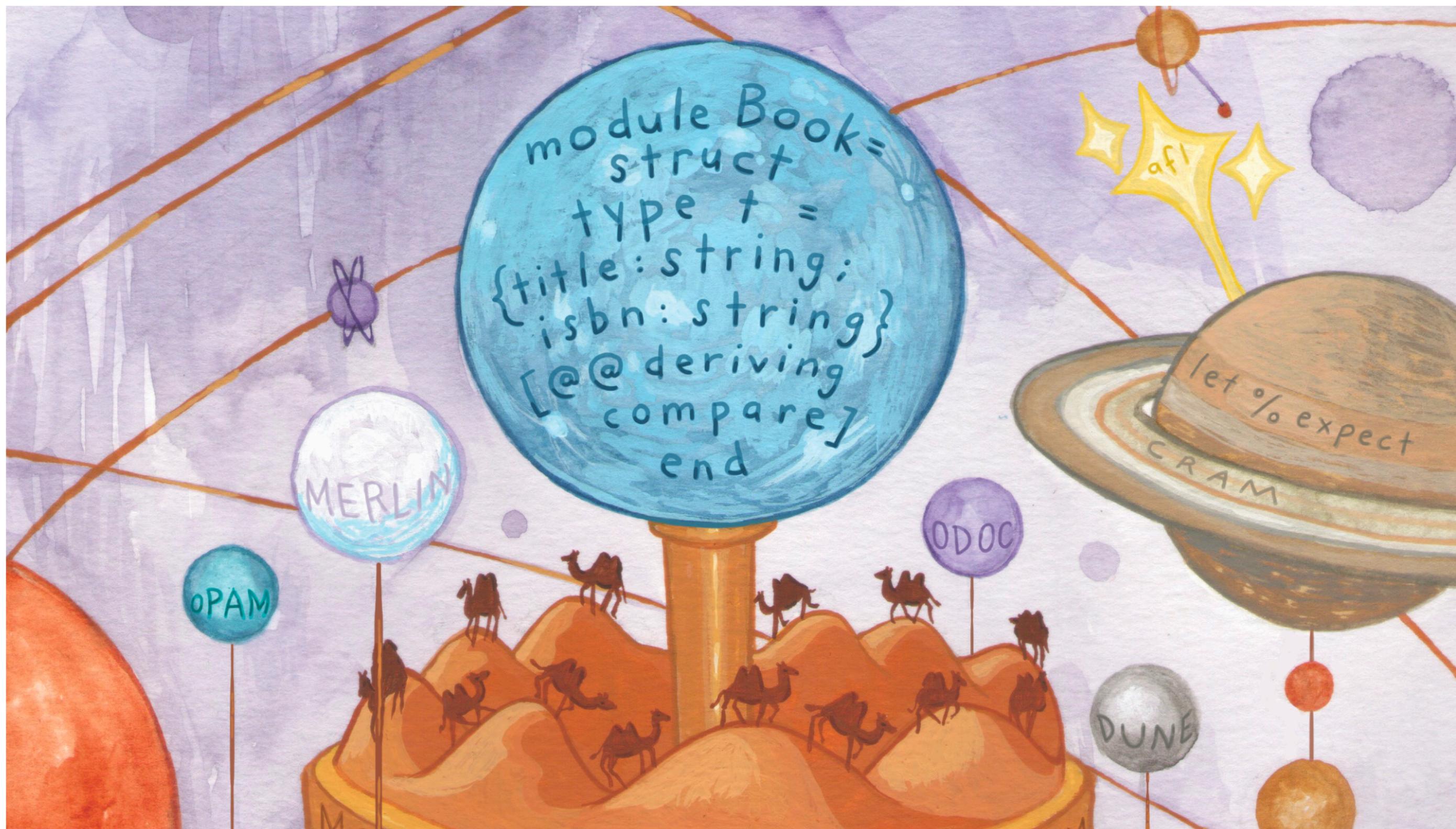
Takeaways



Graphic from *Real World OCaml*

Takeaways

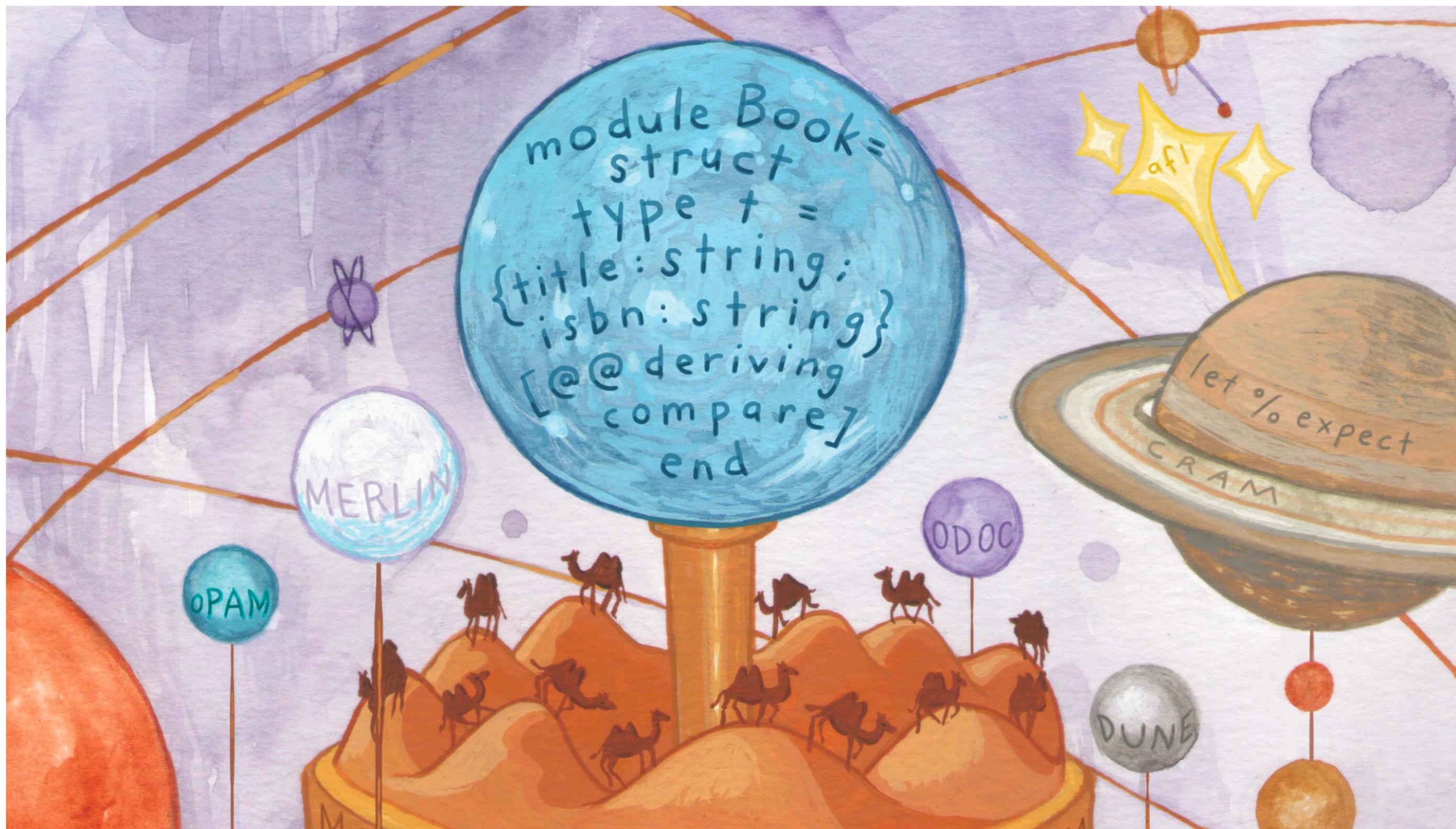
1. Checking observational equivalence requires significant programmer effort



Graphic from *Real World OCaml*

Takeaways

1. Checking observational equivalence requires significant programmer effort
2. $M \vdash A$ can automate this process via PBT!



Graphic from *Real World OCaml*

Thank you!

(Questions?)

ngernest@seas

github.com/ngernest/module_pbt