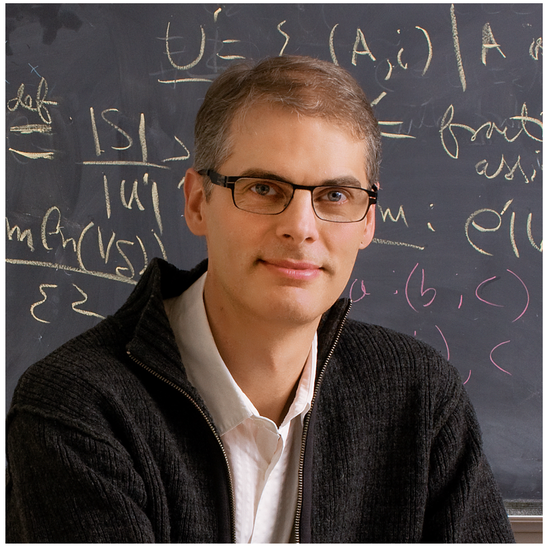
Automated Property-Based Testing for OCaml Modules

Ernest Ng
Advised by Harry Goldstein & Benjamin C. Pierce

PLClub, July 28 2023

Acknowledgements

Harry Goldstein



Benjamin C. Pierce

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

Motivation

modules matter most

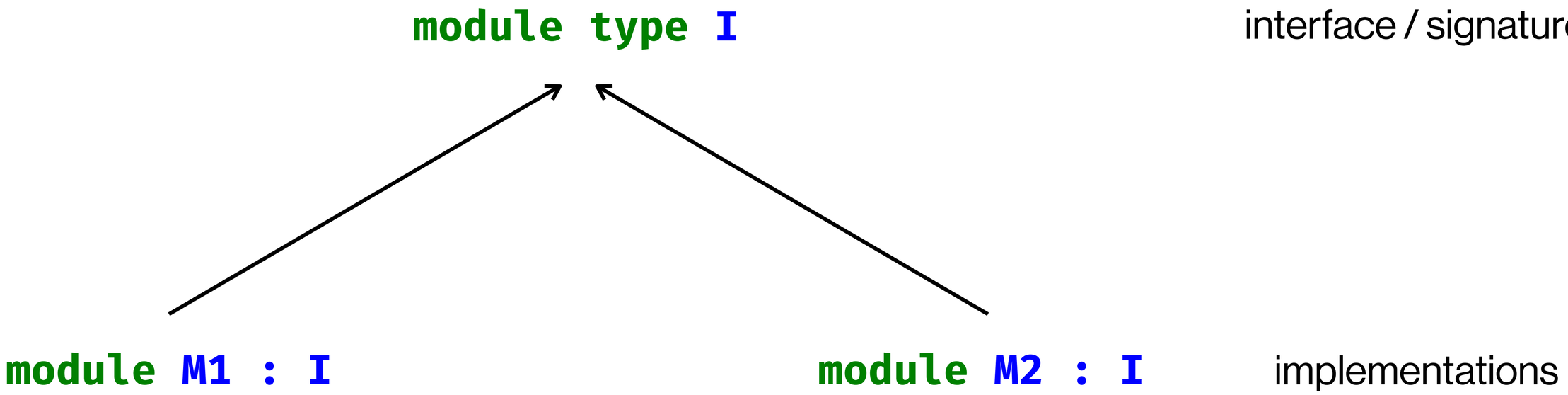
- Bob Harper (2011)

https://existentialtype.wordpress.com/2011/04/16/modules-matter-most/

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**

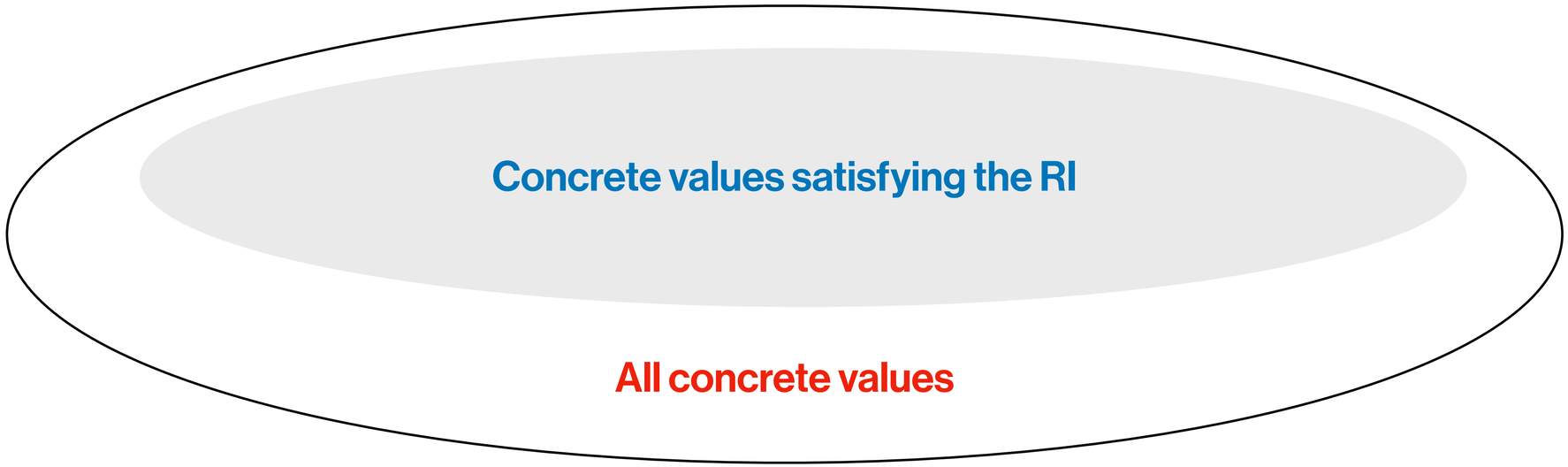
```
module type SetIntf = sig
  type 'a t
  val empty     : 'a t
  val add       : 'a → 'a t → 'a t
  val intersect : 'a t → 'a t → 'a t
  …
  val invariant : 'a t → bool
end
```


Adapted from CIS 1200

Aside: Representation invariants

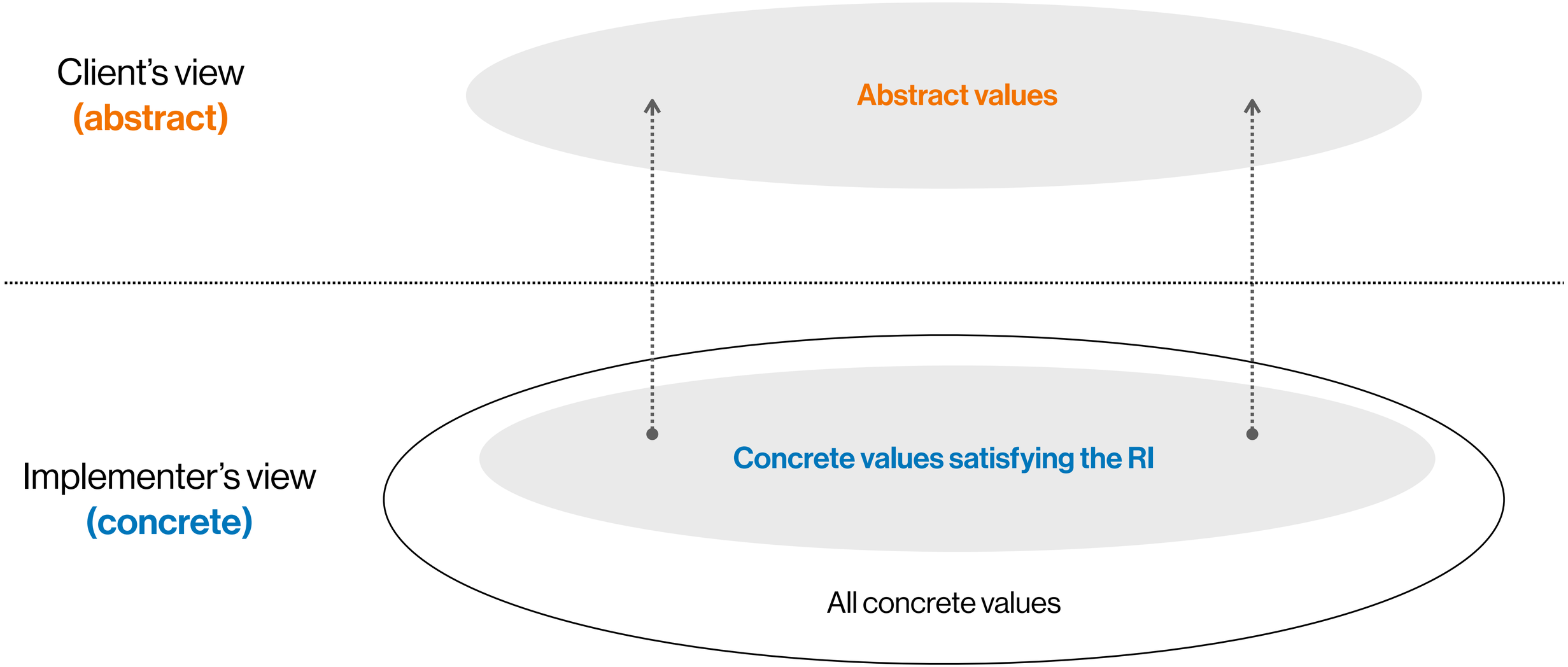
Distinguish valid ***concrete values***
from invalid ***concrete values***

Implementer's view
(concrete)



Adapted from Cornell CS 3110

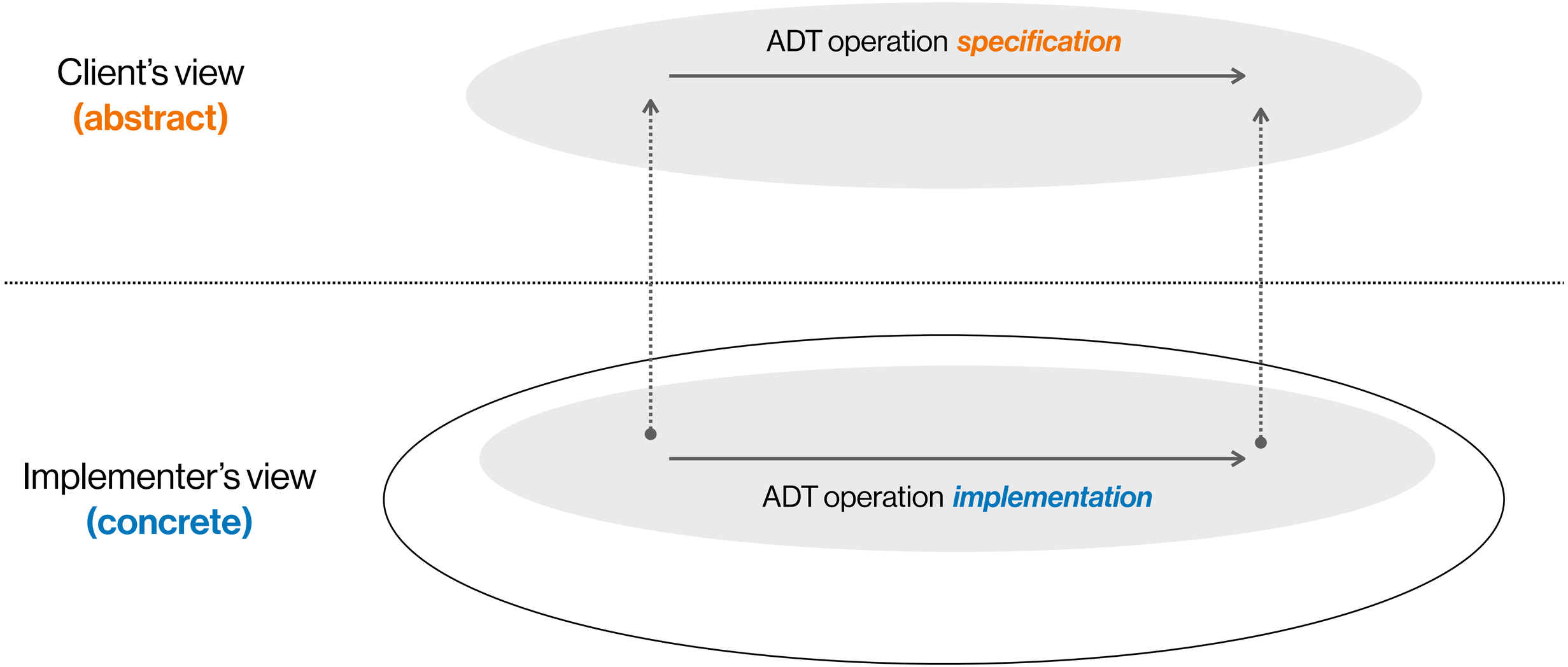
Rep. invariant determines which ***concrete*** values are valid representations of ***abstract*** values



Adapted from Cornell CS 3110

All concrete operations should preserve the RI

Client's view
(abstract)



Implementer's view
(concrete)

Adapted from Cornell CS 3110

Set interface (redux)

```
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
```

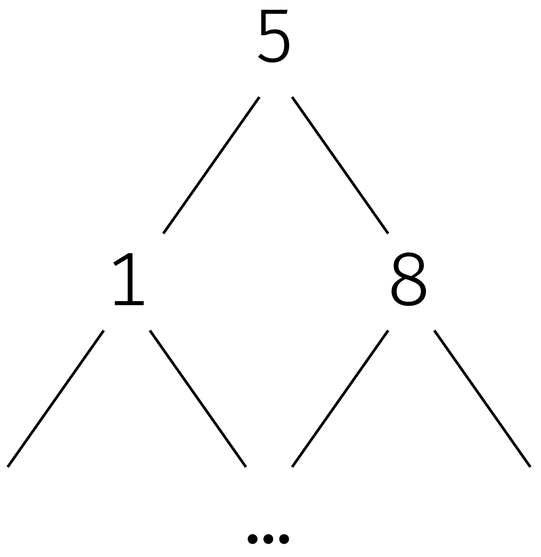

Adapted from CIS 1200

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

```
module ListSet : SetIntf = struct
  type 'a t = 'a list

  (* No duplicates in list *)
  let invariant s =  ...

   ...
end
```


Adapted from CIS 1200

{1, 5, 8, …} ⇝




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

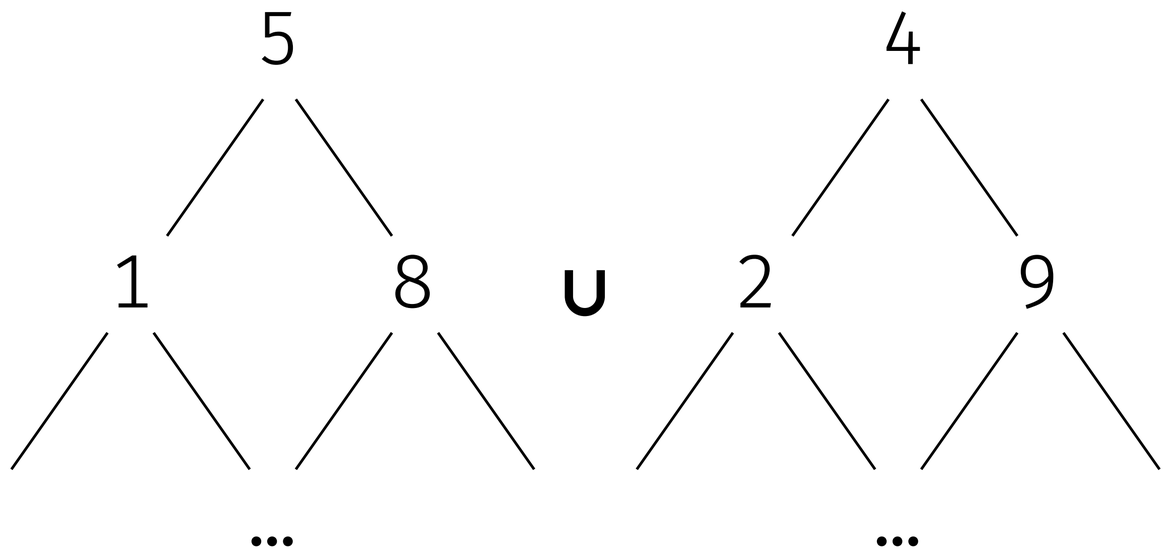
```
module BSTSet : SetIntf = struct
  type 'a t = 'a tree

  (* BST invariant *)
  let invariant s = ...

  …
end
```


Adapted from CIS 1200

Are these equivalent?

{1, 5, 8} ∪ {2, 4, 9}

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



Observational equivalence

equivalent ***inputs*** $\longmapsto$ equivalent ***outputs***

How do we test for

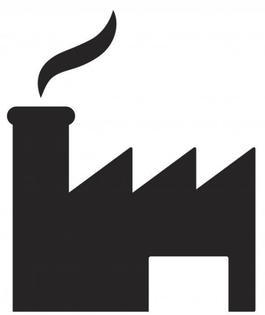
observational equivalence ?

Property-Based Testing

1. Write a ***property***

executable specification
describing desired
program behavior

2. Generate ***random inputs***



$\rightsquigarrow$ x_1 x_2 … x_n

3. Test if random inputs satisfy property

A property for observational equivalence

Suppose we have a notion of equivalence `~=` between **Maps** & **AssocLists**:

```
(~=) : Map.t → AssocList.t → Bool
```
(pseudo-OCaml)

Given two equivalent (**Map**, **AssocList**) pairs `s ~= s' ∧ t ~= t'`,

```
Map.union s t ~= AssocList.union s' t'
```

union of two maps `~=` **union of two AssocLists**

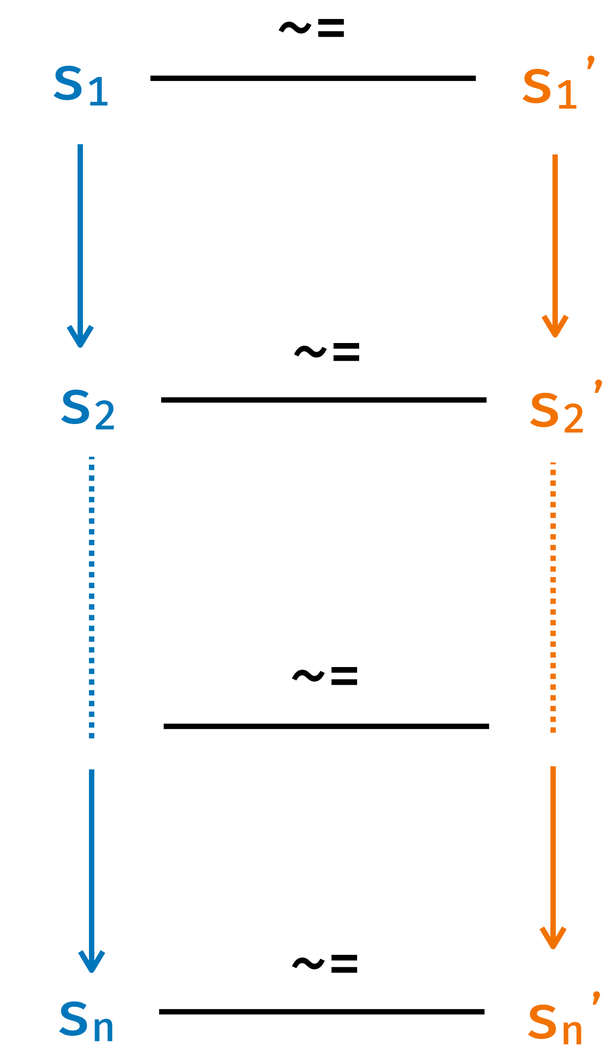


Diagram adapted from Xavier Leroy's slides (2019)

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)
```

Naïvely checking observational equivalence

Need to implement :

- `M1.genInput`
- `M2.genInput`
- `equivalentInputs`
- `equivalentOutputs`

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

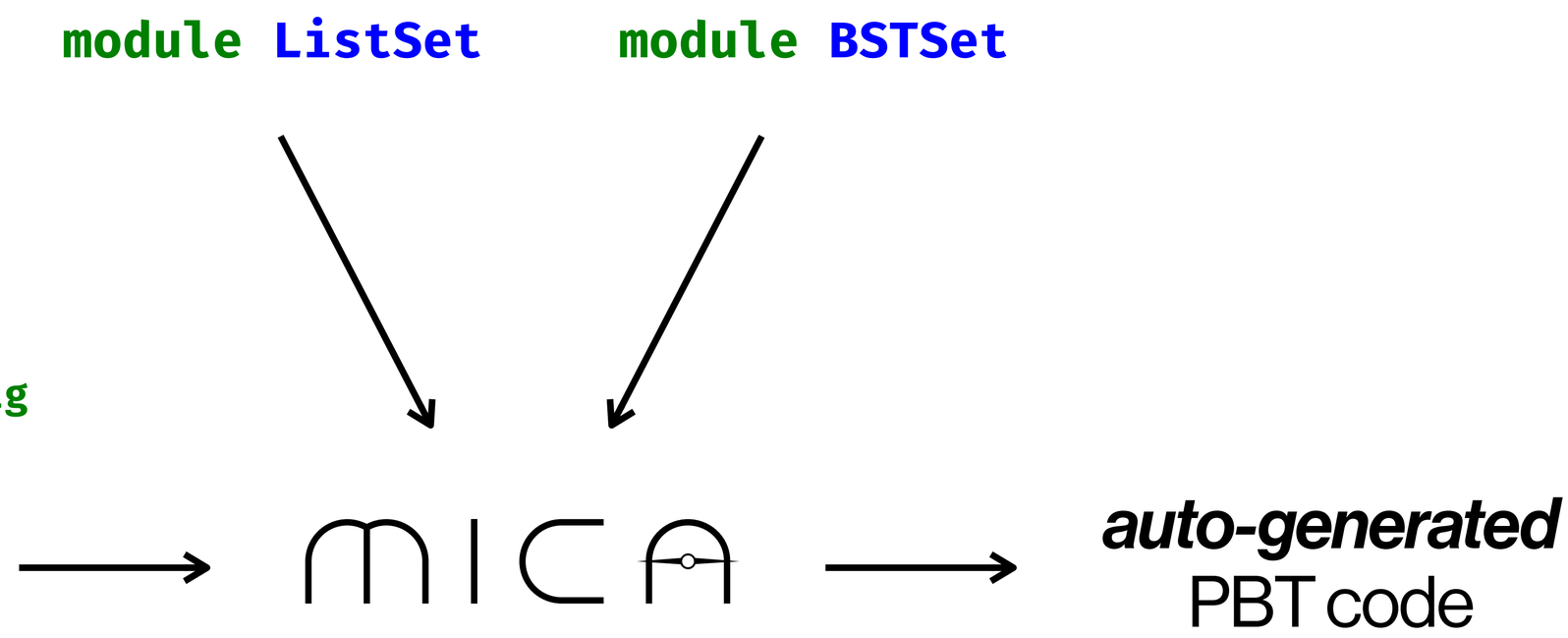
Significant programmer effort required !

Idea :

What if we generate
symbolic expressions instead ?

overview

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



Demo

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                                          type expr =
  val empty : 'a t                       ⟷            | Empty

  val add : 'a → 'a t → 'a t             ⟷            | Add of int * expr

  val union: 'a t → 'a t → 'a t          ⟷            | Union of expr * expr

  val size: ‘a t → int                   ⟷            | Size of expr
  val is_empty : 'a t → bool             ⟷            | Is_empty of expr
  …                                                   …
end
```

Possible **types** & **values** that can be returned by **`expr`**s

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

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

 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)
```

X

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; … ]
  …
```

automatically produces:

Interpreter for symbolic expressions

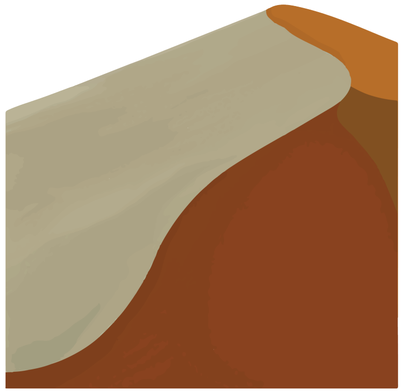
```
val interp : expr → value
```

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

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))
```

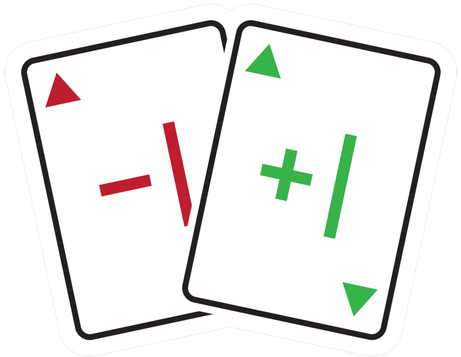
Interpreter

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

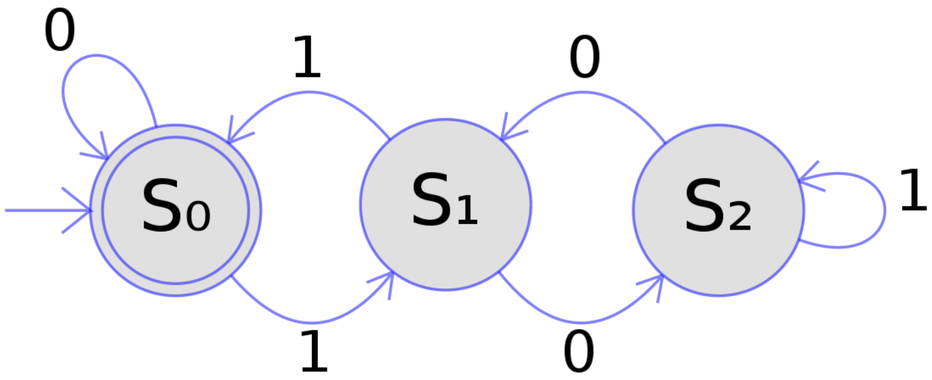
Case studies

Regex matching

Brzozowski derivatives

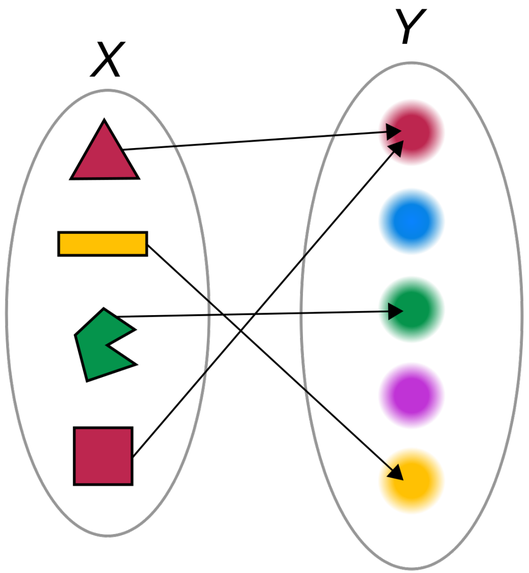
$$u^{-1}S = \{v \in \Sigma^* \mid uv \in S\}$$

DFAs

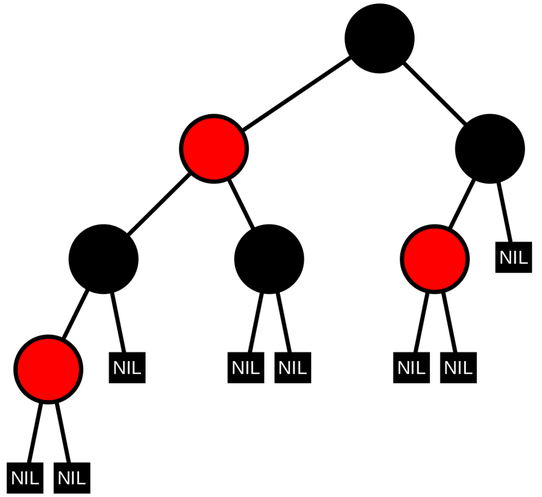


Functional maps

Association lists



Red-Black Trees

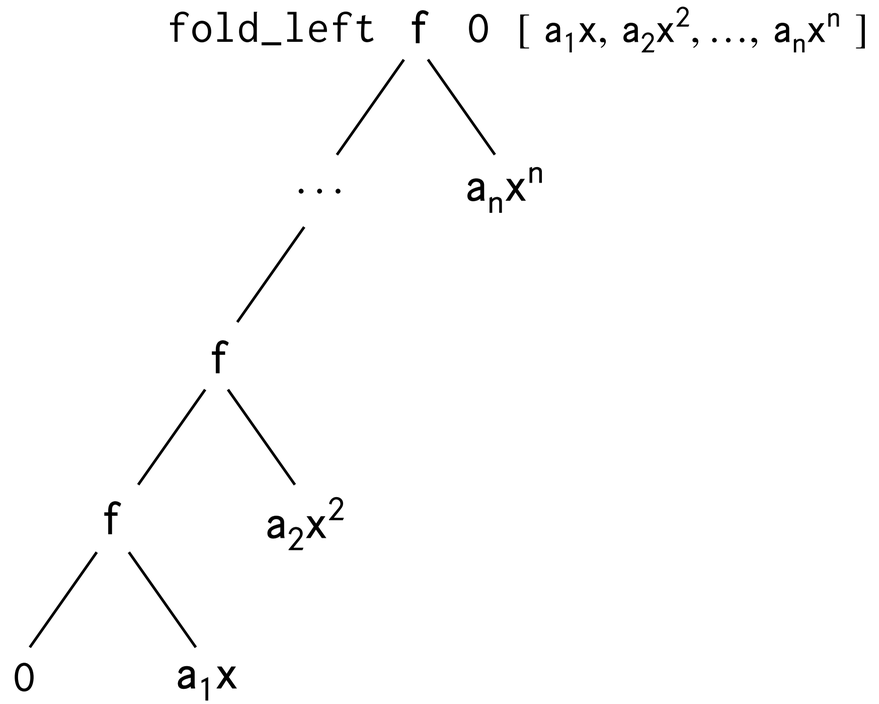


Polynomials

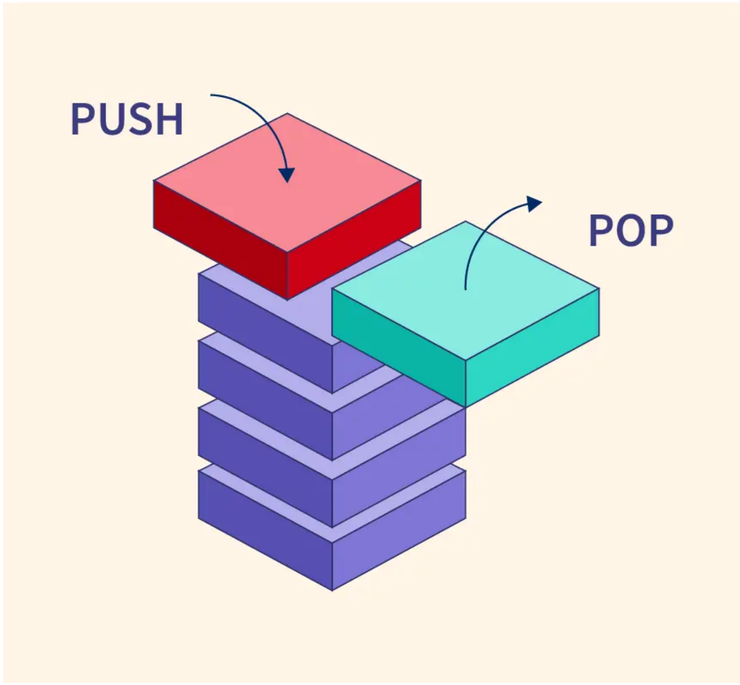
Horner's algorithm

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

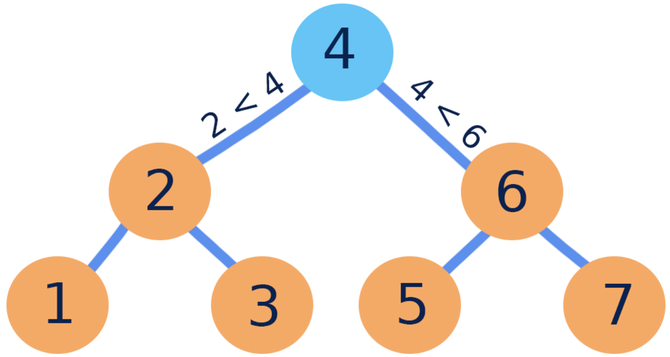
Fold over list of monomials



Stacks

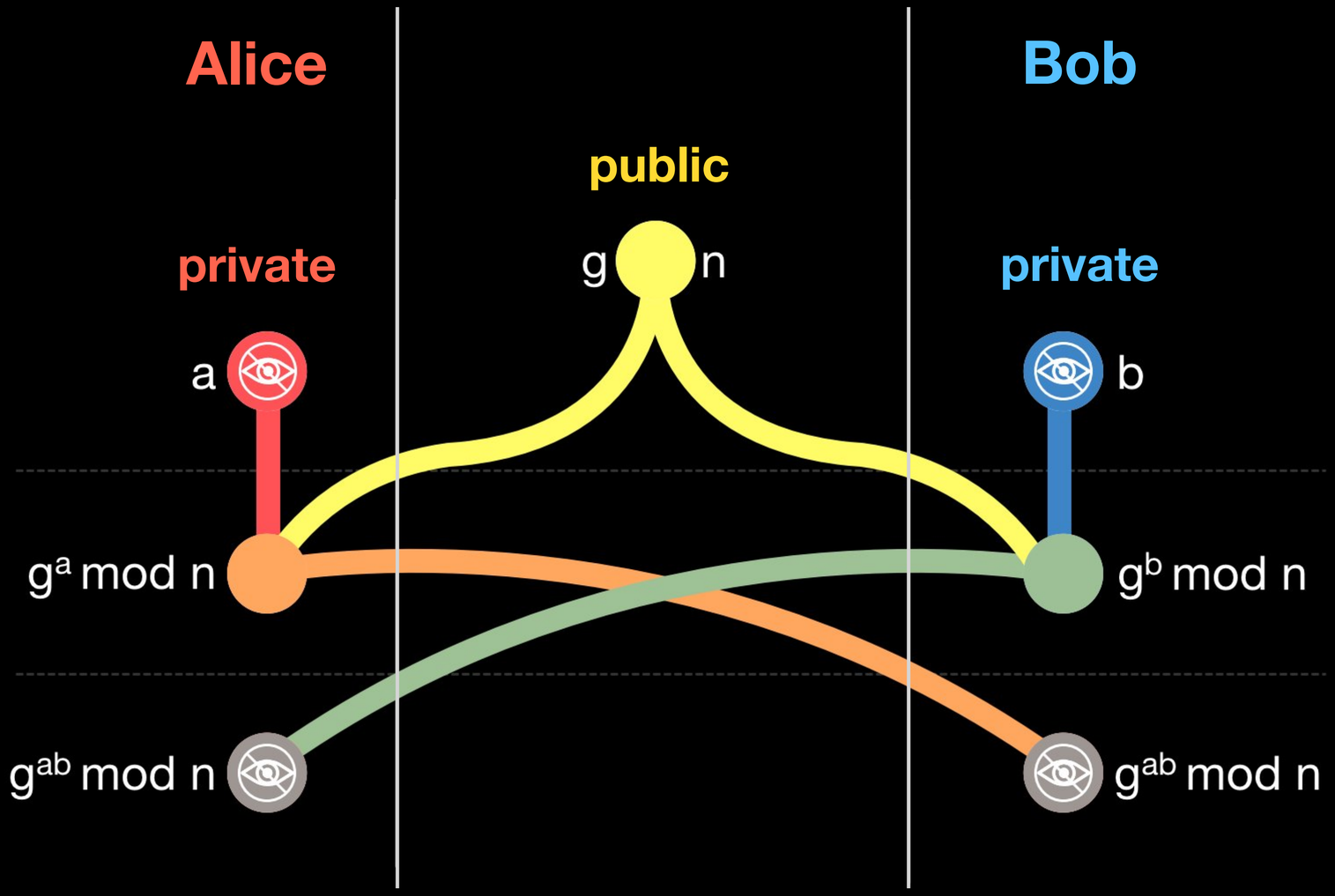


Sets (BSTs, lists)



Diffie-Hellman key exchange

(work in progress)



Challenges with testing Diffie-Hellman

Challenges with testing Diffie-Hellman

Other examples

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

one abstract type

DH interface

two abstract types

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

Challenges with testing Diffie-Hellman

Other examples

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

one abstract type

Add
Union
Size
…
⇝ Set ⇝
Add
Union
Size
…

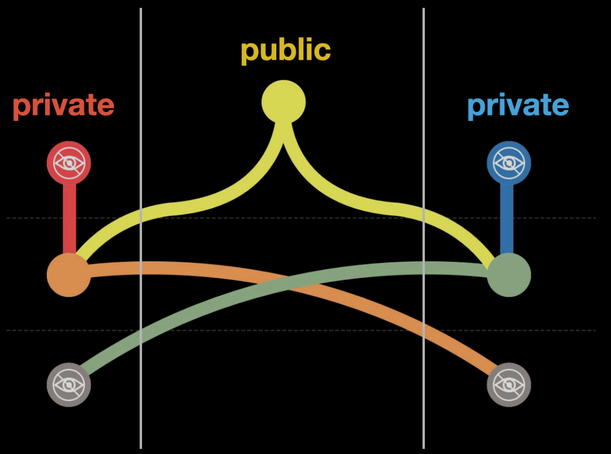
no prescribed order
for calling functions

DH interface

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

two abstract types

prescribed order
for calling functions
(specified by DH)



Challenges with testing Diffie-Hellman

Other examples

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

one abstract type

Add
Union
Size
...
⇝ Set ⇝
Add
Union
Size
...

no prescribed order
for calling functions

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

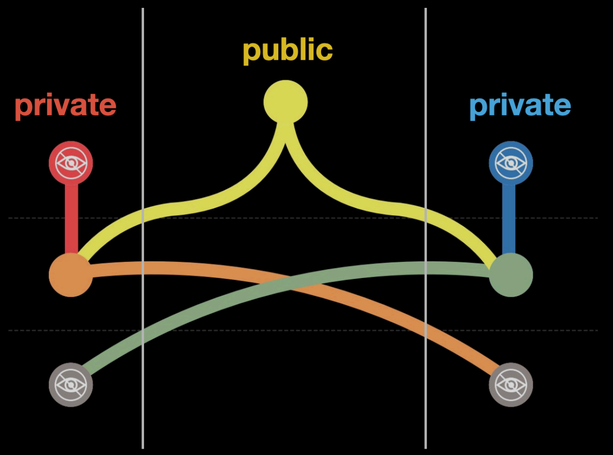
interface tells us how to
to obtain an **`'a t`**

DH interface

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

two abstract types

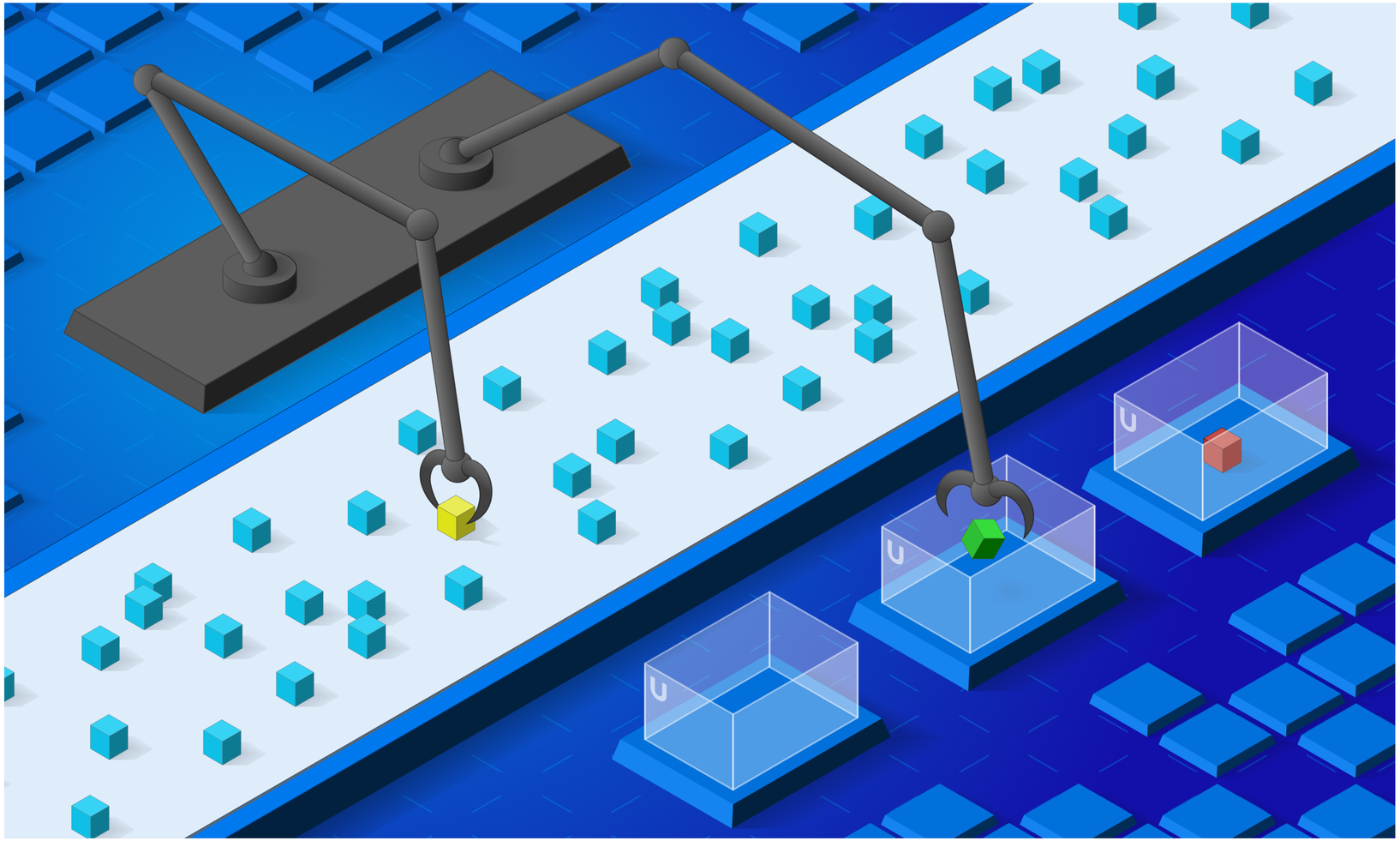
prescribed order
for calling functions
(specified by DH)



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

Catching bugs with Mica

(work in progress)



Graphic from Jane Street

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
`let union s1 s2 =` `  s1 @ s2 (* ▷ dedup *)`	`Invariant [-3; 5; 5; 9]`

Bug: Not enforcing Set invariant for lists

Artificial bug	Failing test case	Correct result

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

`let union s1 s2 =`
`  s1 @ s2 (* ▷ dedup *)`
	`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	Failing test case
`push x stack = stack @ [x]`	`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

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

```
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

Bug *not* caught

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

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

Missing recursive call

Bug caught

Related work

History of model-based PBT

Monadic QuickCheck [Claessen & Hughes 2002]

QuviQ QuickCheck [Hughes 2016]

Model-based PBT

QCSTM [Midtgaard 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)

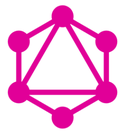
[Midtgaard et al. 2017]

Automatic generation of PBT code

Hypothesis
Ghostwriter

[Hatfield-Dodds et al. 2020]

Clojure



GraphQL

QuickREST

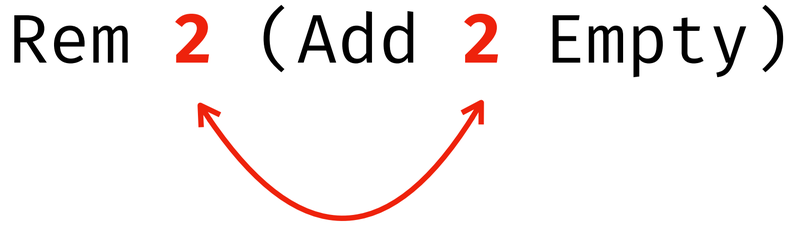
[Karlsson et al. 2020, 2019]

Future work

Future work

Encode dependencies in
generated symbolic expressions

(Set example)



Future work

Encode dependencies in
generated symbolic expressions

Set example

```
Rem 2 (Add 2 Empty)
```

Regex example

```
do
  re ← genRegex
  s  ← genRegexString re
```

Future work : Handle imperative code

Questions:

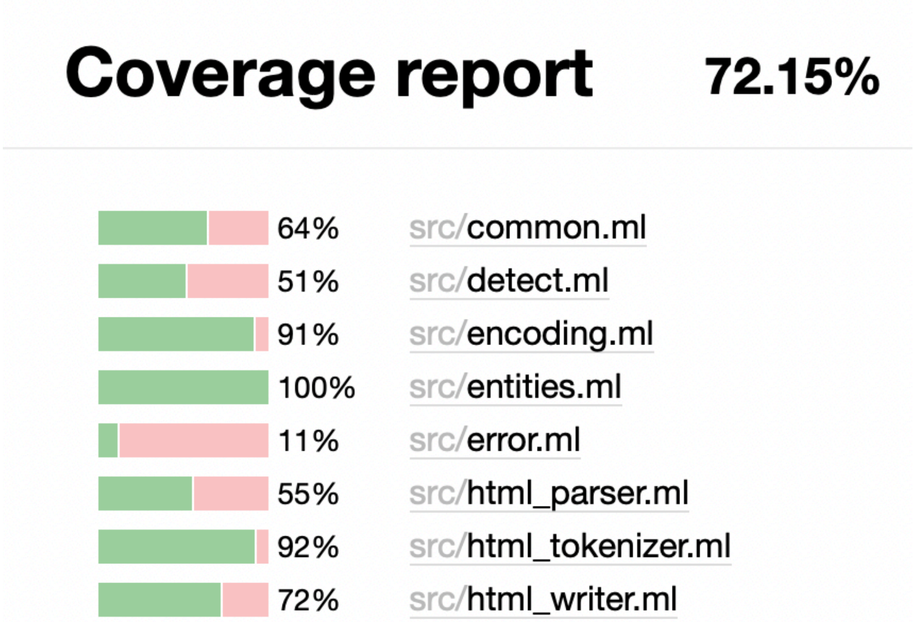
- How should we keep track of state in the generator for symbolic expressions?
- How do we deal with exceptions / effects?

Graphic from Ahrefs

Future work

Empirical evaluation

ppx_bisect



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
```

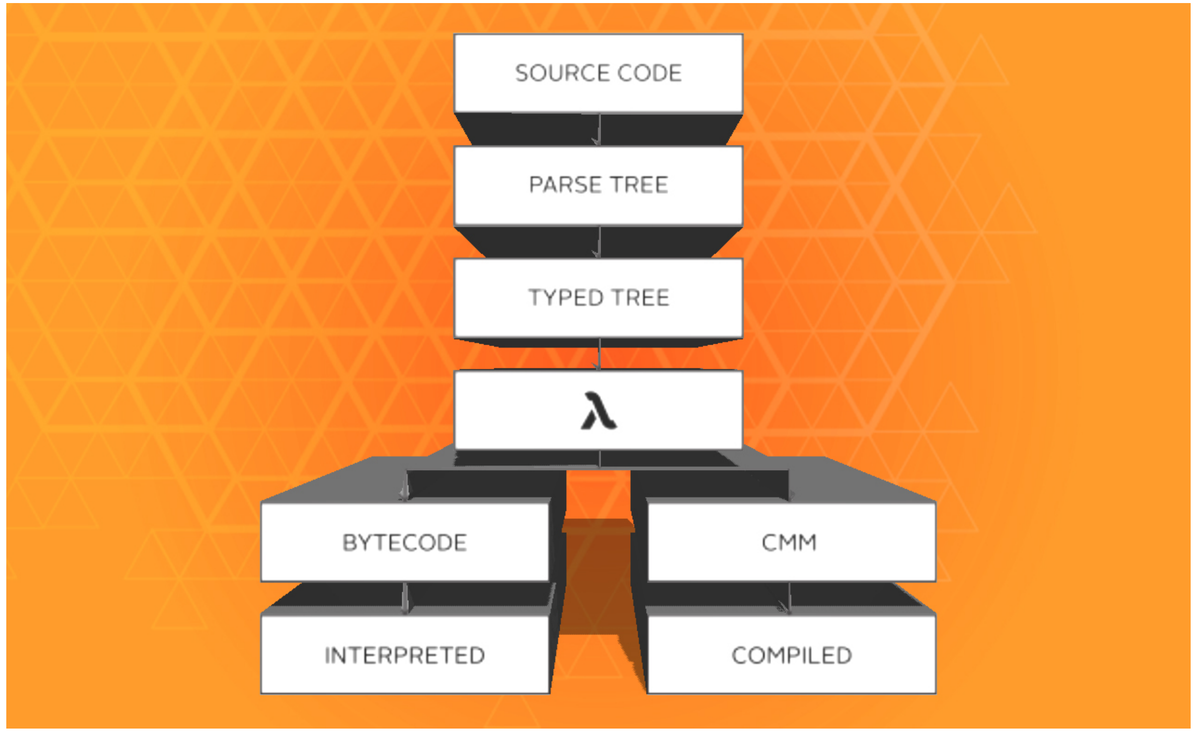
Future engineering work

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. MICA can automate this process via PBT !

Graphic from *Real World OCaml*

Thank you!

(Questions?)

ngernest@seas

github.com/ngernest/module_pbt

Appendix

Appendix 1:
More 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)
`let invariant (s : 'a list) : bool =`
`  (* not @@ *)`
`  List.contains_dup ~compare:(==) s`
	`Invariant Empty`	`TRUE`	`FALSE`

Not enforcing Set invariant (Union for ListSet)
`let union s1 s2 =`
`  s1 @ s2 (* ▷ dedup *)`
	`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`

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

Artificially introduced bug	Failing test case	Correct	Erroneous

Spurious recursive call (BST insertion)
`let rec add x t = match t with …`
`  | Node (lt, n, rt) →`
`  if x < n`
`  then Node (add x lt, n, add x rt) …`
	Fails BST invariant
	`Invariant`
	`  (Rem 2 (Add -7`
	`  (Add -3 (Intersect Empty Empty))))`
	`= …`
	`= Invariant (… (Node (-7, 3, -7))`
		TRUE	FALSE

Spurious negation (Intersection of two BSTs)
`let intersect t1 t2 =`
`  let commonElts =`
`  filter ~f:(fun x → not @@ mem x t1)`
`  (inorderTraversal t2) in`
`List.fold commonElts ~init:empty`
`  ~f:(fun acc x → add x acc)`
	`(Is_empty`
	`  (Intersect (Rem 8 (Rem -7 (Rem 7 Empty)))`
	`  (Intersect (Union (Add 2 Empty)`
	`  (Union Empty Empty))`
	`  (Add 8 (Add 4 Empty))`
	`= …`
	`= Is_empty Empty`
		TRUE	FALSE

Changing the base case of the BST invariant function to **False** instead of True
(details omitted)
	`Invariant`
	`  (Union (Add 9 (Add -4 (Rem 6 Empty)))`
	`  (Intersect (Add -10 (Intersect Empty Empty))`
	`  (Union (Add -6 Empty)`
	`  (Union Empty Empty))))`
	`= Invariant {-10, -6, -4, 9}`
		TRUE	FALSE

BST implementation of Sets: removing an element

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

```
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

Bug *not* caught

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

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

Missing recursive call

Bug caught

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