Mica

Automated Differential Testing for OCaml Modules

Ernest Ng
Harry Goldstein
Benjamin Pierce

A module signature for sets

```
module type S = sig
  type 'a t
  val empty  : 'a t
  val insert : 'a → 'a t → 'a t
  …
end
```

```
module type S
```

{1, 5, 8, …}

```
module type S
```

{1, 5, 8, …}

```
module ListSet : S
```

[1; 5; 8; …]

```
type 'a t = 'a list
```

```
module type S
```

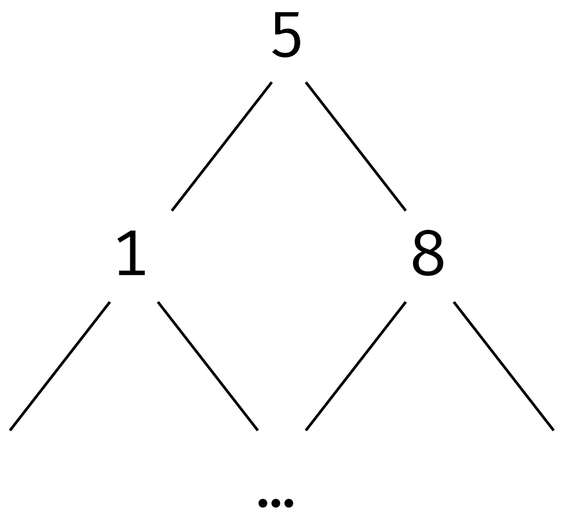
{1, 5, 8, …}

```
module ListSet : S
```

[1; 5; 8; …]

```
type 'a t = 'a list
```

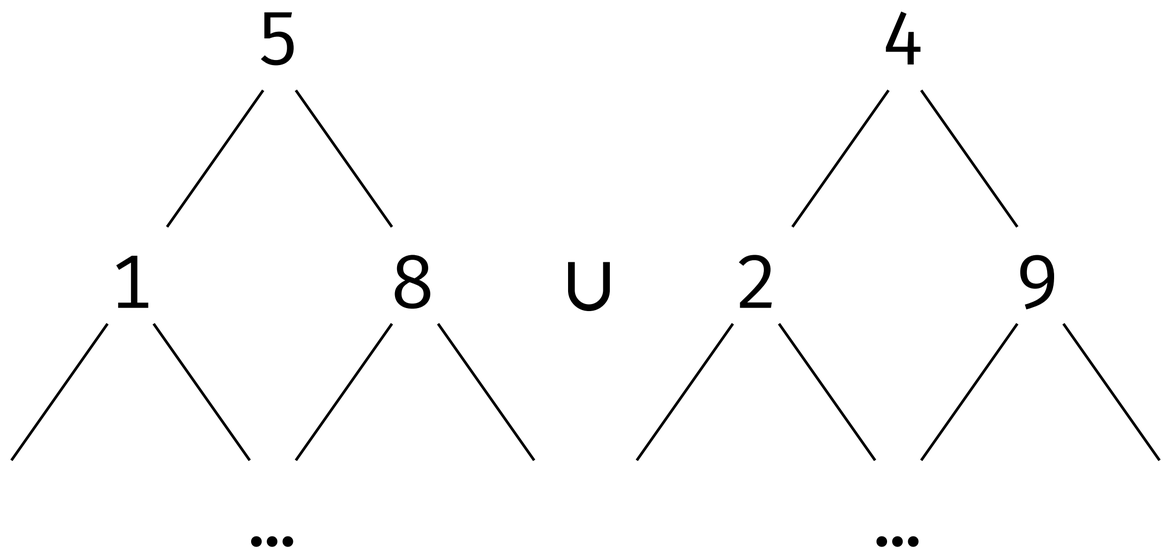
```
module BSTSet : S
```



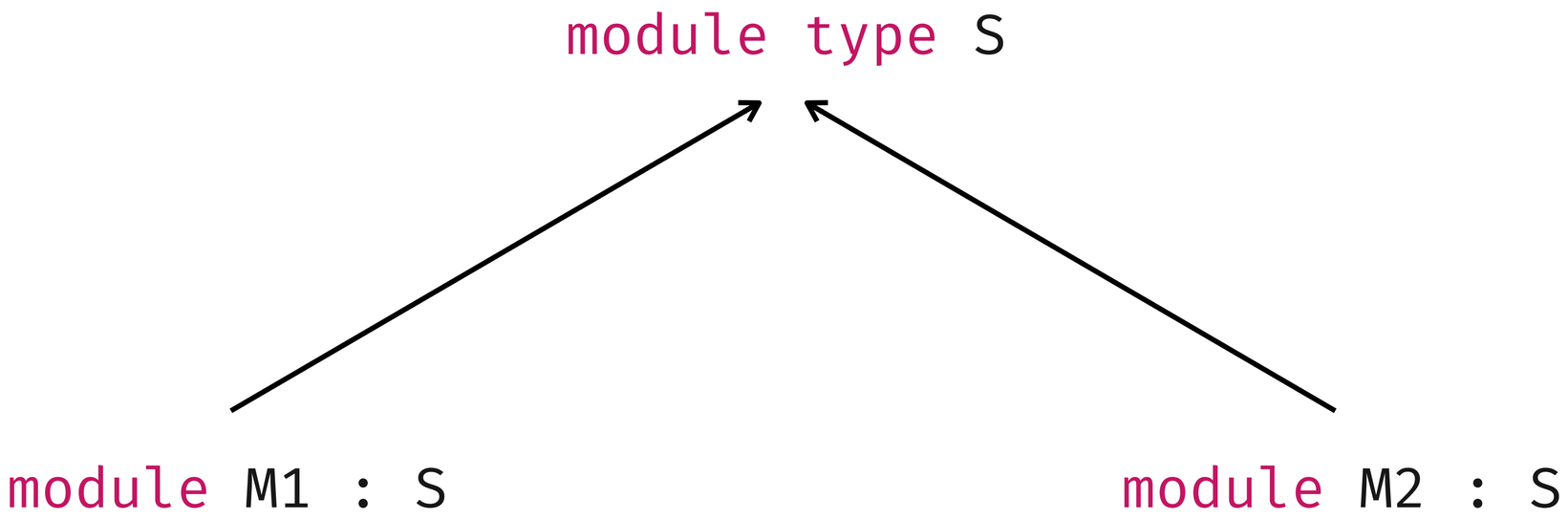

```
type 'a t = 'a tree
```

Are these ***equivalent***?

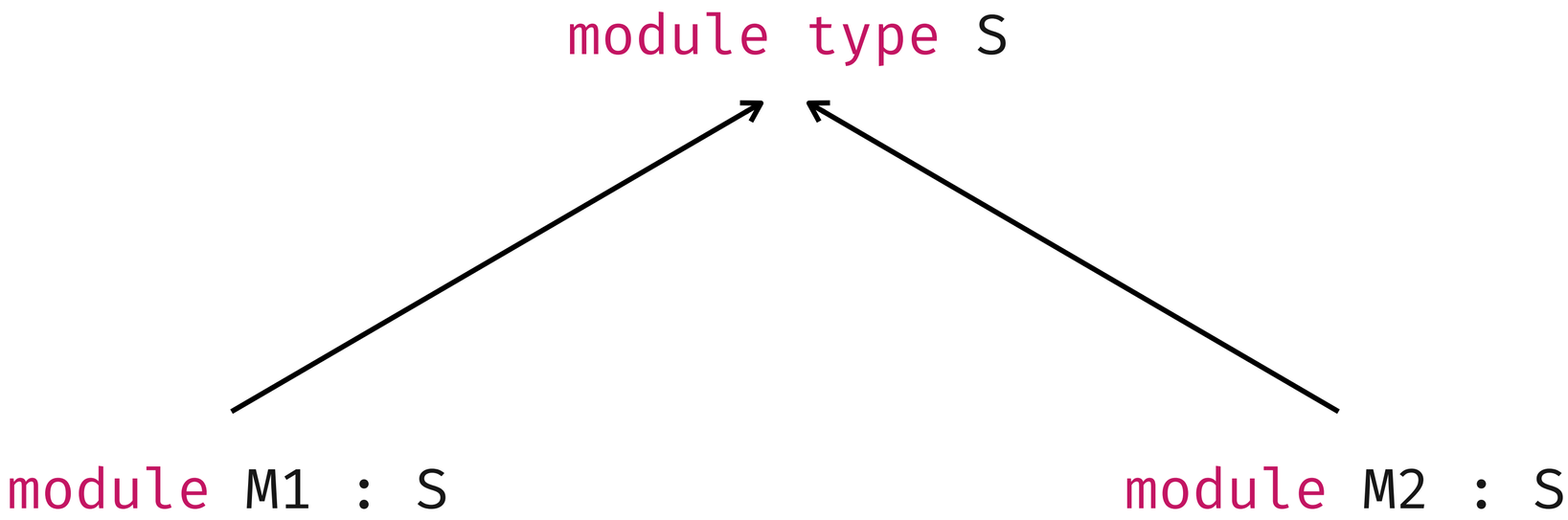
```
[1; 5; 8] ++ [2; 4; 9]
```



Two modules can implement the same interface completely differently …

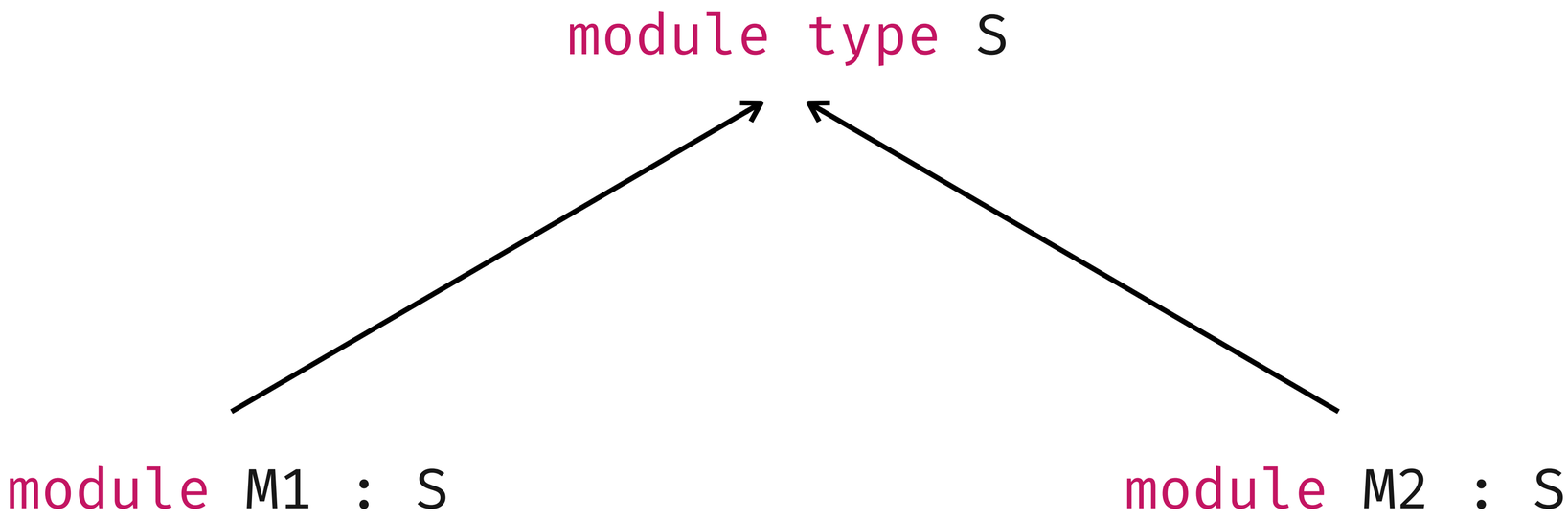


Two modules can implement the same interface completely differently …



Clients can use S without knowing whether they're getting M1 or M2!

Two modules can implement the same interface completely differently …

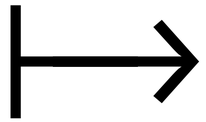


Clients can use S without knowing whether they're getting M1 or M2!

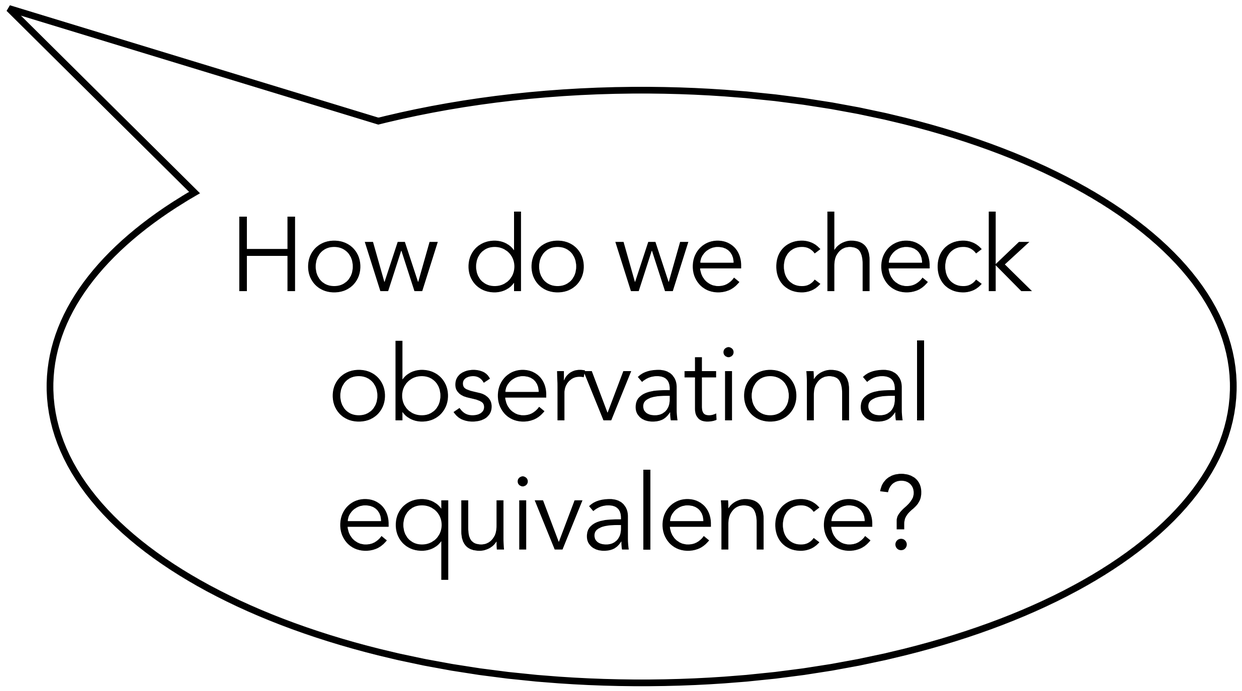
Both modules ought to behave *equivalently* w.r.t. the same interface!

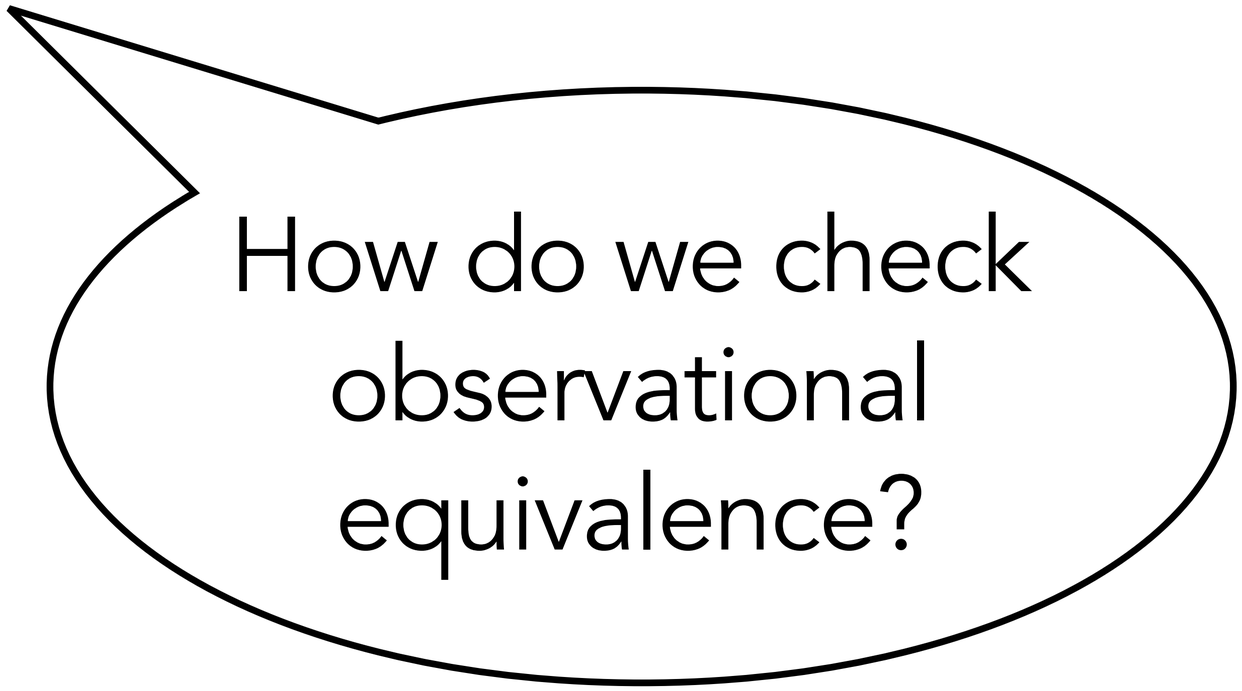
Observational Equivalence

equivalent
inputs



equivalent
outputs


How do we check observational equivalence?



We can use
property-based testing!

Property-Based Testing

Property-Based Testing

1. Write ***properties***

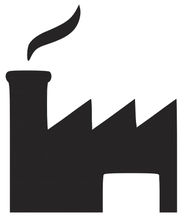
```
∀ x. P(x)
```

Property-Based Testing

1. Write ***properties***

$$\forall\ x.\ P(x)$$

2. Generate ***random inputs***



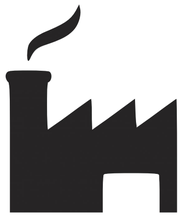
$$\rightsquigarrow \quad x_1 \quad x_2 \quad \ldots$$

Property-Based Testing

1. Write ***properties***

$\forall\ x.\ P(x)$

2. Generate ***random inputs***



$\rightsquigarrow\quad x_1\ \ x_2\ \ \ldots$

3. Check if inputs satisfy property

Property-Based Testing

Popularized by:

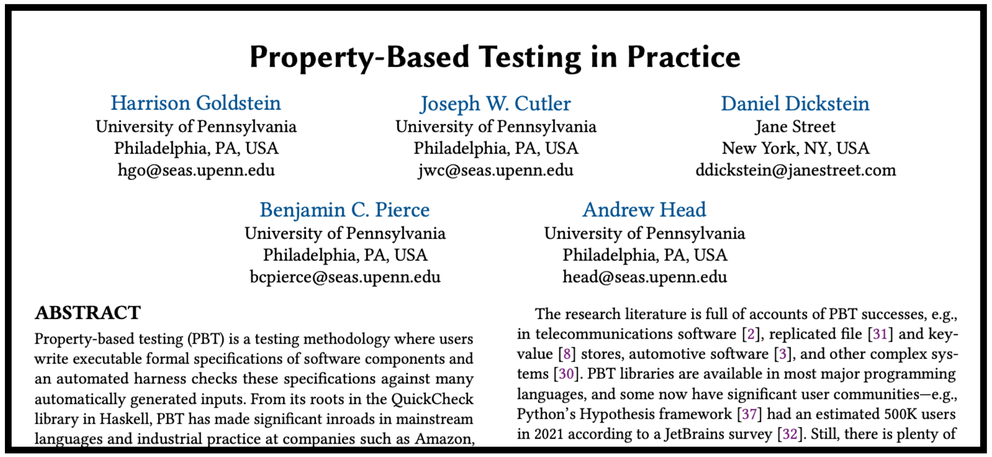
QuickCheck

Claessen & Hughes (ICFP 2000)

Why should we care?

1. Testing observational equivalence requires significant programmer effort

Goldstein et al. (ICSE '24)



Property-Based Testing in Practice

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Joseph W. Cutler
University of Pennsylvania
Philadelphia, PA, USA
jwc@seas.upenn.edu

Daniel Dickstein
Jane Street
New York, NY, USA
ddickstein@janestreet.com

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

ABSTRACT

Property-based testing (PBT) is a testing methodology where users write executable formal specifications of software components and an automated harness checks these specifications against many automatically generated inputs. From its roots in the QuickCheck library in Haskell, PBT has made significant inroads in mainstream languages and industrial practice at companies such as Amazon,

The research literature is full of accounts of PBT successes, e.g., in telecommunications software [2], replicated file [31] and key-value [8] stores, automotive software [3], and other complex systems [30]. PBT libraries are available in most major programming languages, and some now have significant user communities—e.g., Python's Hypothesis framework [37] had an estimated 500K users in 2021 according to a JetBrains survey [32]. Still, there is plenty of

30 OCaml developers
interviewed on their use of PBT

Jane Street

Why should we care?

1. Testing observational equivalence requires significant programmer effort

- Developers described this process as ***"tedious"*** & ***"overwhelming"***
- High ***"overhead"*** associated with writing PBT boilerplate

Goldstein et al. (ICSE '24)

> in languages like OCaml with rich module structures, researchers should aim to increase automation around differential testing and produce a test harness for comparing modules without requiring any manual setup

Why should we care?

2. Large OCaml software systems are built using multiple modules that implement the same signature

MirageOS

Module Signatures

Implementations

Module type	Implementations
Mirage_kv.RO	Crunch, Kv_Mem, Kv_unix, Mirage_tar, XenStore, Irmin, Filesystems
Mirage_kv.RW	Wodan
Mirage_fs.S	Fat, Git, Fs_Mem, Fs_unix
Mirage_net.S	tuntap, vmnet, rawlink
ARP, IP, UDP, TCP	IPV4, IPV6, Qubesdb_IP, Udp, Updv4_socket, Tcp, Tcpv4_socket, …
STACK	Direct, Socket, Qubes, Static_IP, With_DHCP
RANDOM	Stdlib, Nocrypto, Test
HTTP	Cohttp, Httpaf
FLOW	Conduit.With_tcp, Conduit.With_tls
DNS, DHCP, SYSLOG	Dns, Unix, Charrua_unix, Charrua, Syslog.Tcp, Syslog.Udp, Syslog.Tls
	Jitsu, Irmin, …

Radanne et al. (2019)

What if I told you …

What if I told you …

You can take two modules that implement the same signature …

```
module type S
```

```
module M1 : S
```

```
module M2 : S
```

What if I told you …

You can take two modules that implement the same signature …

```
module type S
```

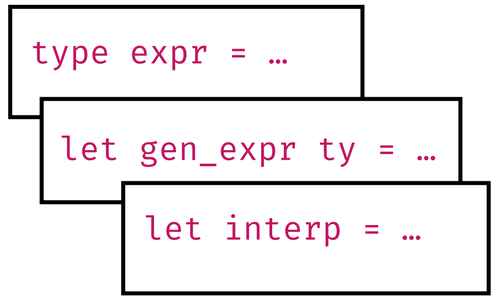
```
module M1 : S
```

```
module M2 : S
```

… and ***automatically*** get PBT code that compares them?

Mica

```
module type S = …
[@@deriving mica]
```



Jane Street

QUICKCHECK

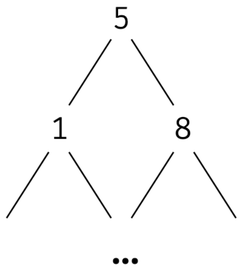
```
$ Mica: OK, passed 10000 tests.
```

Mica *automatically* generates random S-operations & tests that `M1`, `M2` are observationally equivalent w.r.t. `S`

```
module M1 : S
```

```
[1; 5; 8; …]
```

```
module M2 : S
```

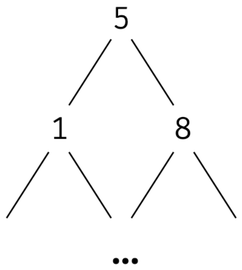


Mica *automatically* generates random S-operations & tests that `M1`, `M2` are observationally equivalent w.r.t. `S`

```
module M1 : S
```

```
[1; 5; 8; …]
```

```
module M2 : S
```




```
M1.size (M1.add 2 M1.empty)   ≡≡≡   M2.size (M2.add 2 M2.empty)
```

⋮ randomly generated ⋮

```
M1.is_empty (…)   ≡≡≡   M2.is_empty (…)
```

Mica derives the following automatically:

Mica derives the following automatically:

Types

(to be explained later)

```
expr
 ty
value
```

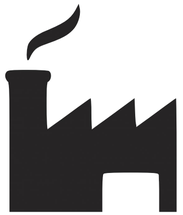
Mica derives the following automatically:

Types

(to be explained later)

```
expr
ty
value
```

QuickCheck Generator



$\rightsquigarrow$ e_1 e_2 …

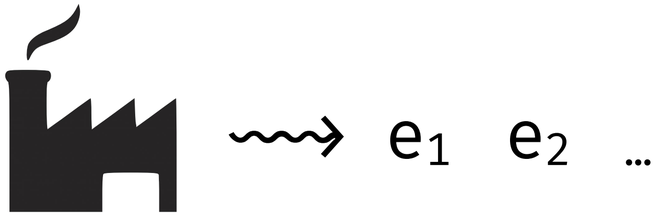
Mica derives the following automatically:

Types

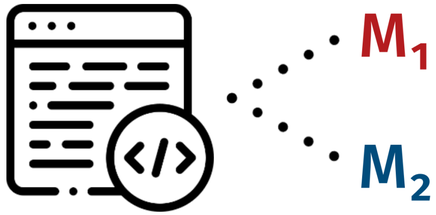
(to be explained later)

```
expr
ty
value
```

QuickCheck Generator



Interpreter



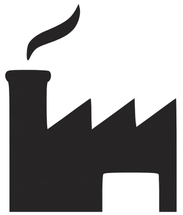
Mica derives the following automatically:

Types

(to be explained later)

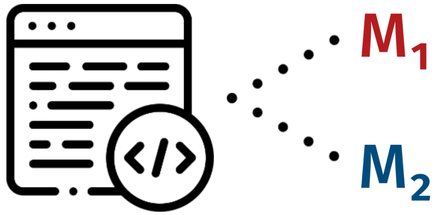
```
expr
ty
value
```

QuickCheck Generator

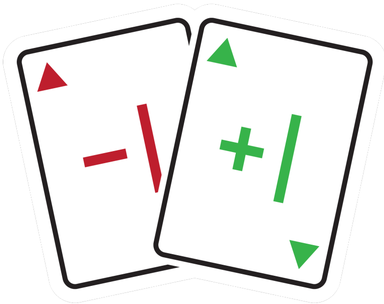


$\rightsquigarrow$ e_1 e_2 …

Interpreter



Test Harness



Symbolic Expressions

Model operations in the module signature using an inductively-defined algebraic data type

module signature

```
val empty    : 'a t
val is_empty : 'a t → bool
val insert   : 'a → 'a t → 'a t
…
```


the **expr** type

```
type expr =
  | Empty
  | Is_empty of expr
  | Insert of int * expr
  …
```

Symbolic Expressions

Types

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

Symbolic Expressions

Values

```
type value =
  | ValBool of bool
  | ValIntT of int M.t
  | …
```

Interpretation Functor

```
module Interpret (M : S) = …
```

```
expr    ⟶    value

Insert (2, Empty)   ⟼   M.insert 2 M.empty
```

QuickCheck Generator

randomly generate
symbolic representations of
well-typed expressions

```
gen_expr : ty → expr Generator.t
```

QuickCheck Generator

randomly generate
symbolic representations of
well-typed expressions

```
gen_expr : ty → expr Generator.t
```

```
Union (Insert (2, Empty), Empty)
```

✓

QuickCheck Generator

randomly generate
symbolic representations of
well-typed expressions

```
gen_expr : ty → expr Generator.t
```

`Union (Insert (2, Empty), Empty)` ✓

`Is_empty (Size Empty)` X

Test Harness Functor

Checks observational equivalence at ***concrete*** types

```
module TestHarness (M1 : S) (M2 : S) = …
```

```
int
```

✓

```
'a t
```

X

Observational equivalence is only well-defined for ***concrete*** types

Observational equivalence is only well-defined for ***concrete*** types

Concrete types

(Defined *only* in terms of primitive types, polymorphic equality works!)

```
int

string list

char * bool
```

Can be compared directly

✓

Observational equivalence is only well-defined for ***concrete*** types

Concrete types

(Defined *only* in terms of primitive types, polymorphic equality works!)

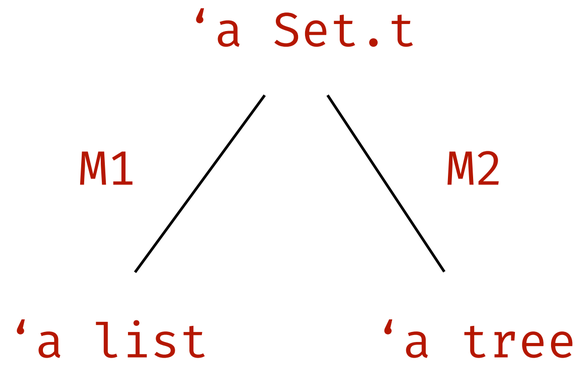
```
int

string list

char * bool
```

Can be compared directly

✓

Abstract types

(Definition is hidden, abstract notion of equality)



Can't be compared directly

X

Type-Directed Observational Equivalence

Mica checks for equivalence at all unique ***concrete*** types

which appear as the return types of functions in S

```
module type S = sig
  type 'a t
  val member : 'a → 'a t → bool
  val size   : 'a t → int
  …
end
```

Type-Directed Observational Equivalence

Mica checks for equivalence at all unique ***concrete*** types

which appear as the return types of functions in S

```
module type S = sig
  type 'a t
  val member : 'a → 'a t → bool
  val size   : 'a t → int
  …
end
```

} the results of ***observation functions***
(clients of S can only *observe* discrepancies
between M1 & M2's behavior by calling these functions)

QuickCheck Generator

Generates ***random*** symbolic expressions

```
(Size
  (Union (Add 2 Empty) …)
```

QuickCheck Generator

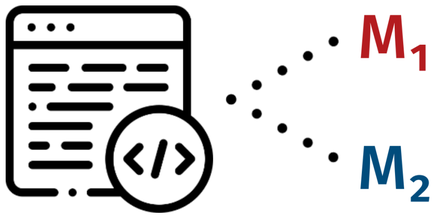
Generates ***random*** symbolic expressions

```
(Size
  (Union (Add 2 Empty) …)
```

Interpreter

Interprets expressions over modules

```
M.size
  (M.union (M.add 2 M.empty) …)
```



QuickCheck Generator

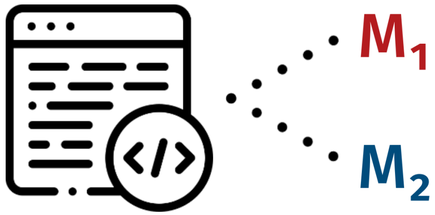
Generates ***random*** symbolic expressions

```
(Size
  (Union (Add 2 Empty) …)
```

Interpreter

Interprets expressions over modules

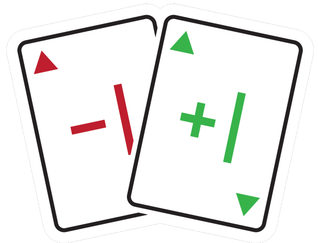
```
M.size
  (M.union (M.add 2 M.empty) …)
```



Test Harness

Checks observational equivalence

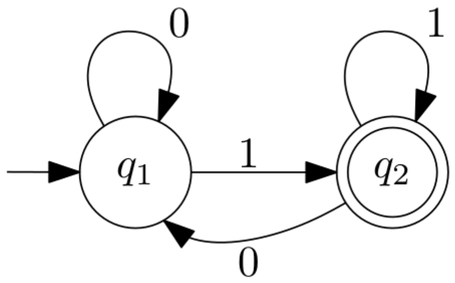
```
v1 =? v2
```



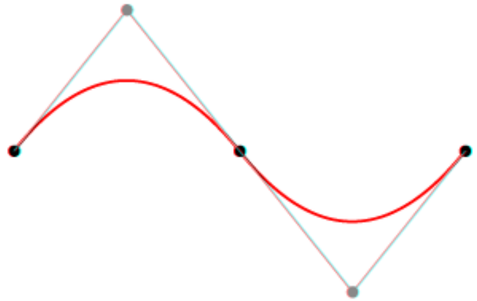
Case Studies

Case Studies

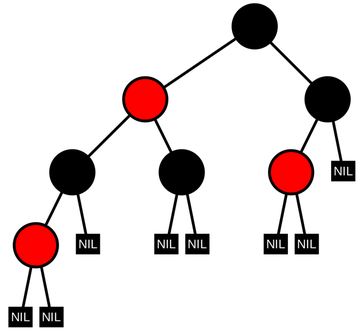
Regex Matchers



Polynomials



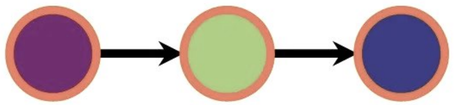
Persistent Maps



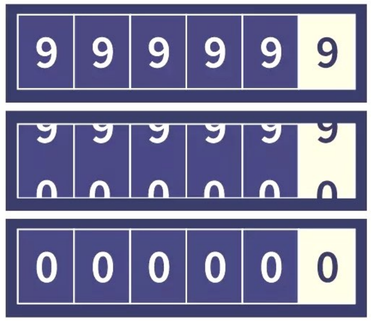
Character Sets

Á Â Ã Ä Å Æ Ç È É
Ñ Ò Ó Ô Õ Ö × Ø Ù
á â ã ä å æ ç è é

Ephemeral Queues

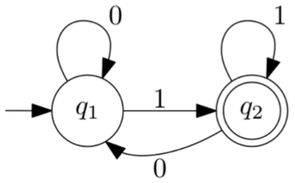


Unsigned Integers

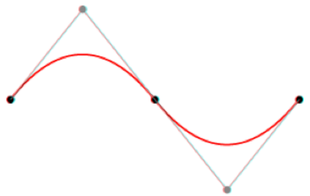


35 manually-inserted bugs caught

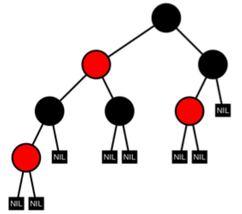
Regex Matchers



Polynomials



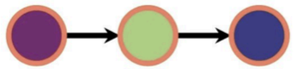
Persistent Maps



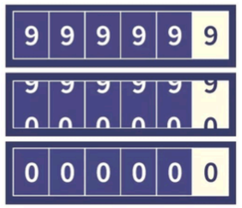
Character Sets

Á Â Ã Ä Å Æ Ç È É
Ñ Ò Ó Ô Õ Ö × Ø Ù
á â ã ä å æ ç è é

Ephemeral Queues



Unsigned Integers



6 real-world OCaml libraries

Case study: ***How to Specify It***

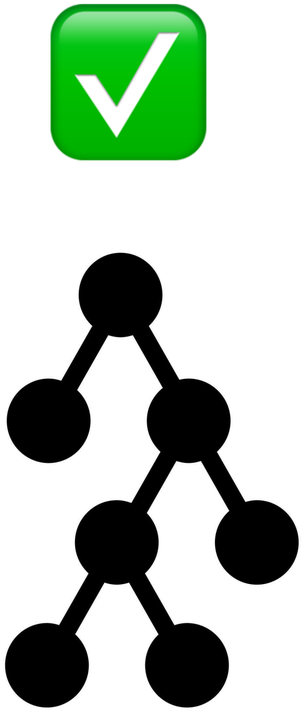
John Hughes

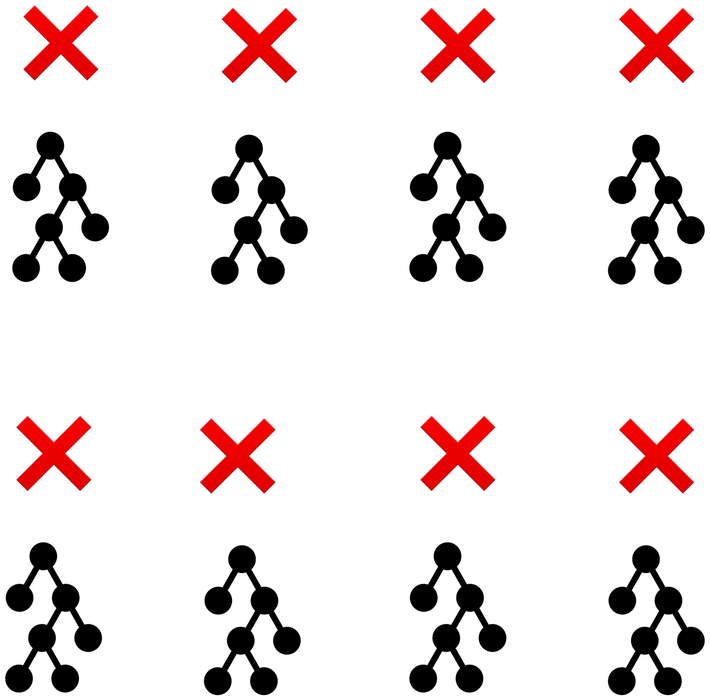
How to Specify it!

A Guide to Writing Properties of Pure Functions.

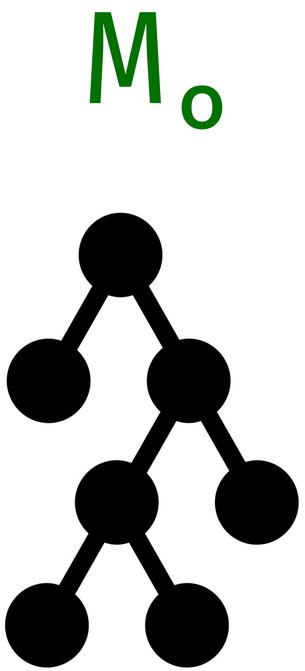
(TFP '19)

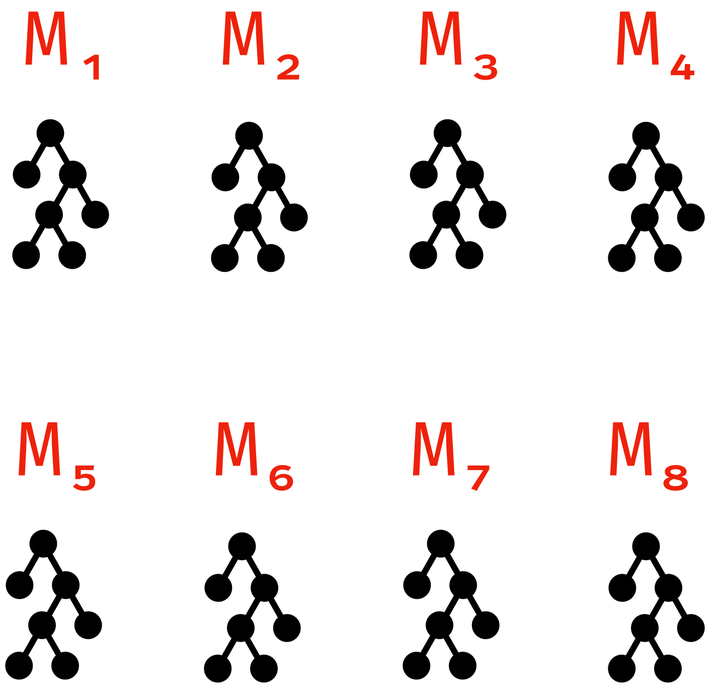
Case study: **BSTs done 9 ways**



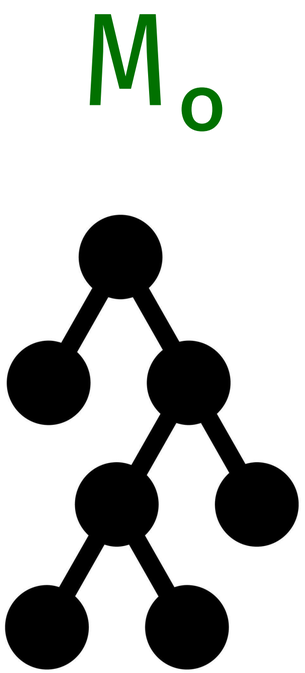


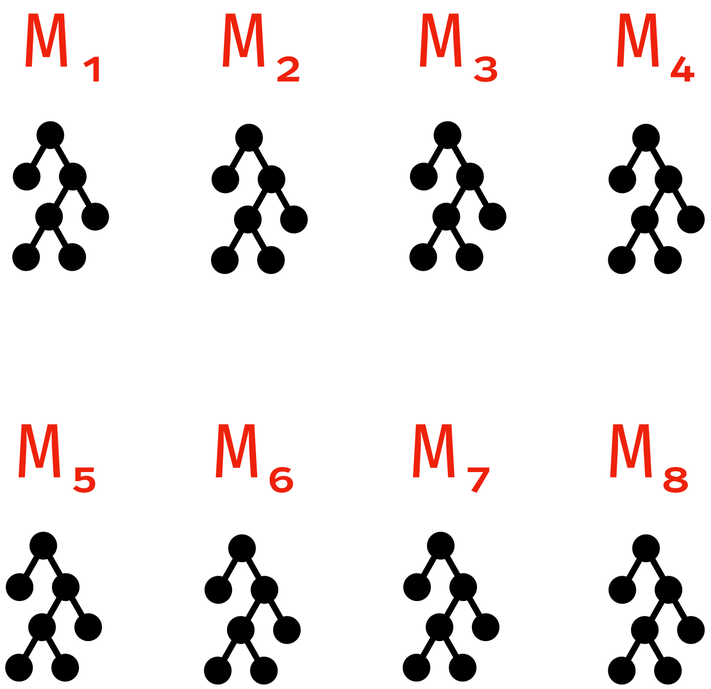
Case study: **BSTs done 9 ways**





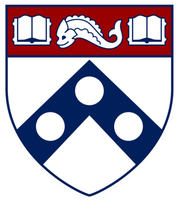
Case study: **BSTs done 9 ways**

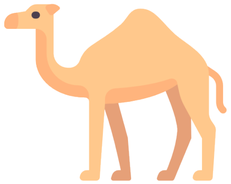




Each bug caught within ~170 randomly generated symbolic expressions

Case study: **Finding bugs in student assignments**

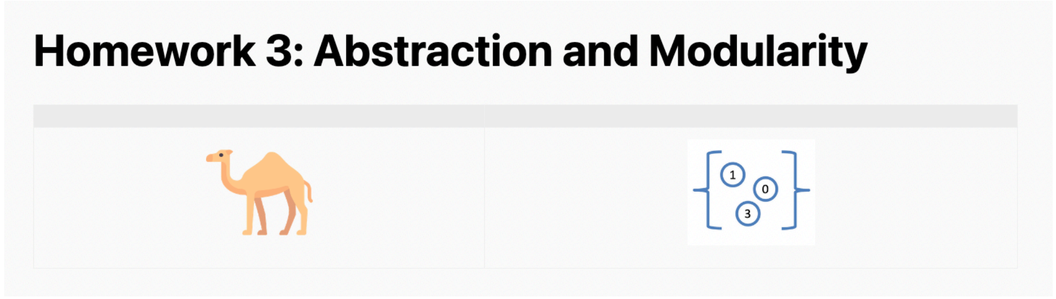




- Penn's undergrad intro OCaml class
  - 400 students every semester

- Natural source of bugs!

Case study: **Sets done 400 ways**



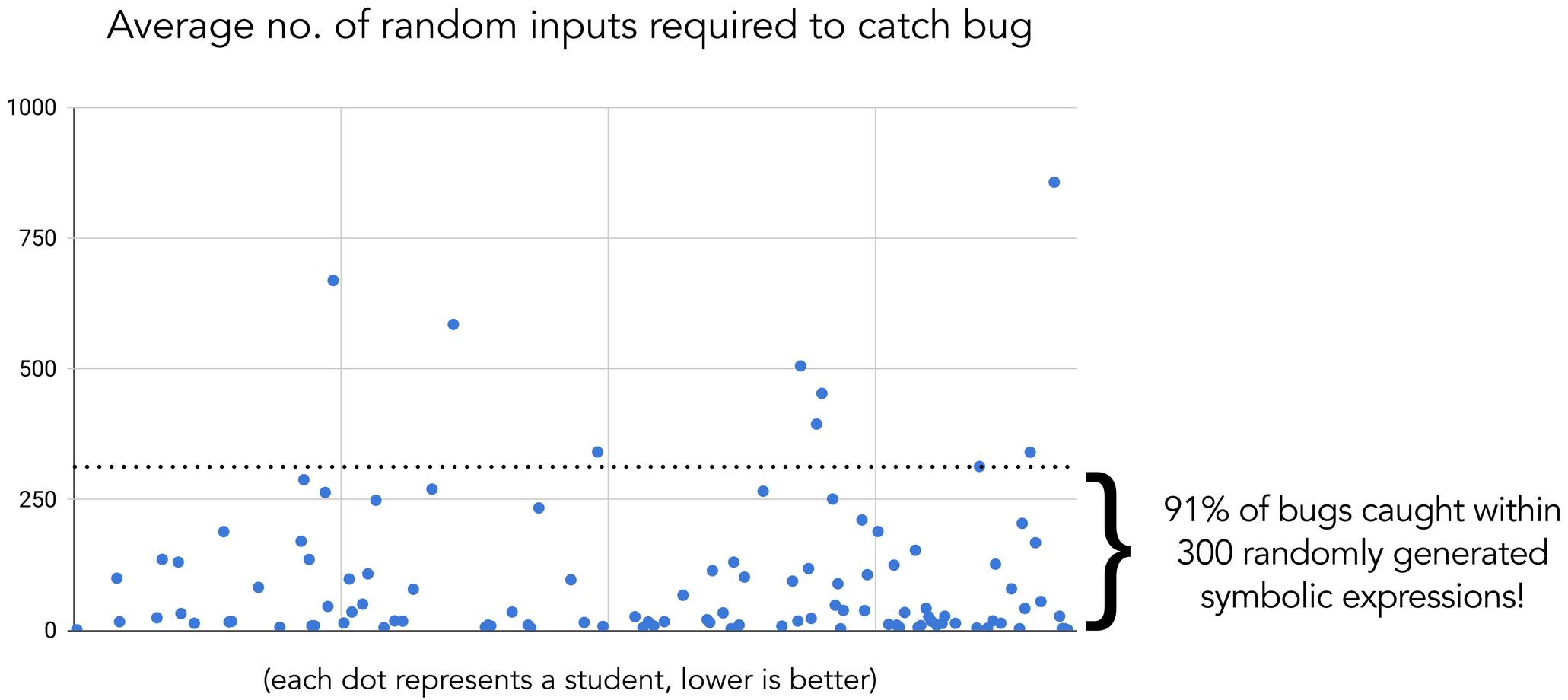
Penn CIS 1200, Fall 2023

As homework, students were asked to implement sets using lists & BSTs

(we looked at historical data from Fall ’23)

Are students’ implementations observationally equivalent?

Mica caught bugs in 107 students' submissions! (29% of the class)



VS Code Integration with **Tyche**

Goldstein et al. (to appear at UIST '24)

TYCHE: Making Sense of Property-Based Testing Effectiveness

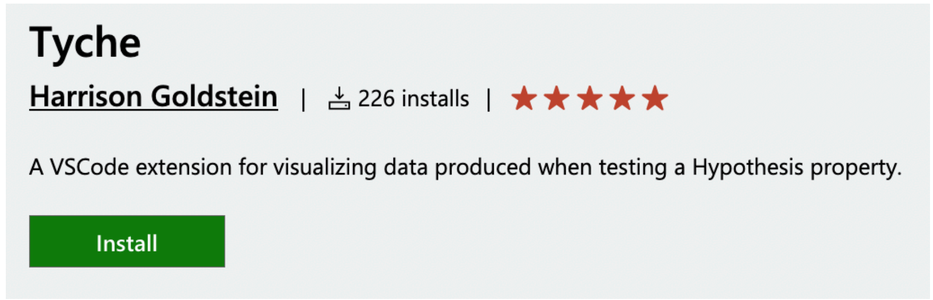
Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Jeffrey Tao
University of Pennsylvania
Philadelphia, PA, USA
jefftao@seas.upenn.edu

Zac Hatfield-Dodds*
Anthropic
San Francisco, CA, USA
zac.hatfield.dodds@gmail.com

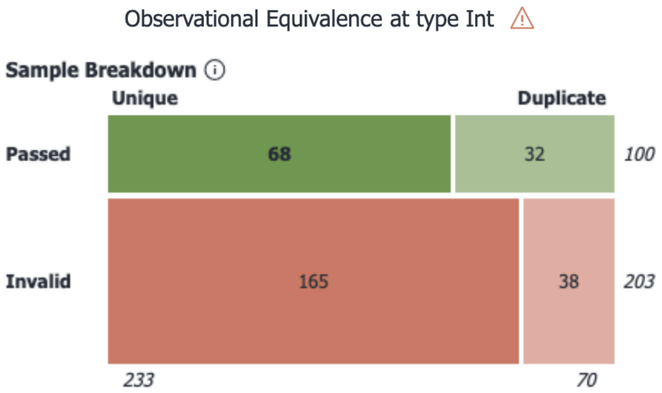
Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

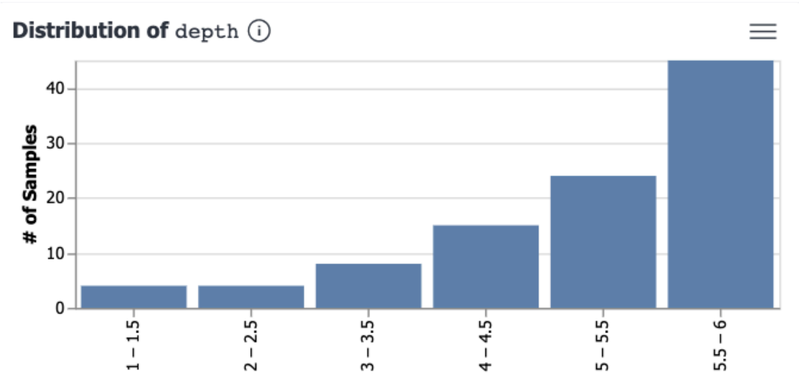


Using **Tyche** to display **Mica's** test results

Observational equivalence test results

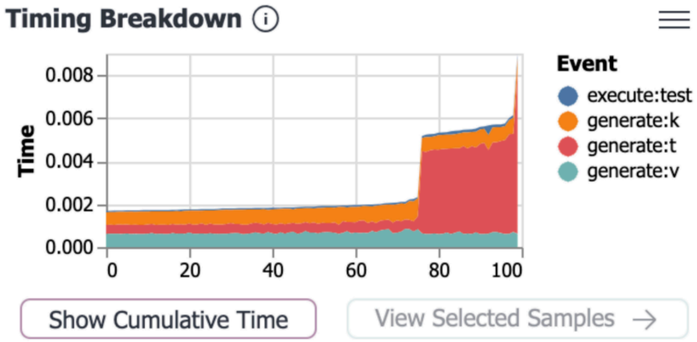


Distribution of symbolic expressions




```
(Size (Add 4 Empty))                                                    4×

(Size (Add 6 Empty))                                                    3×

(Size
 (Union (Intersect (Add 6 (Add 6 Empty)) (Rem 7 (Add 7 Empty)))
  (Add 7 Empty)))
```

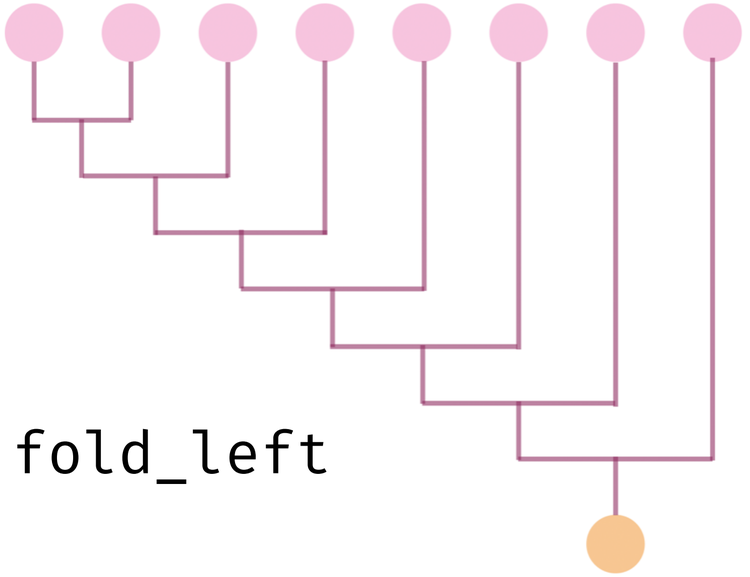
Timing information



Future Work

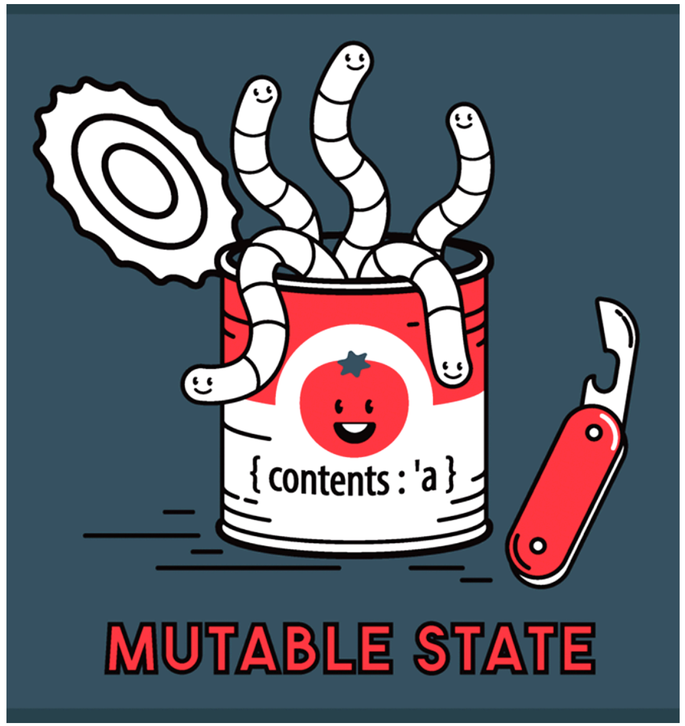
Future Work

Support more **higher-order functions**



Future Work

Support **imperative code**



Graphic from Ahrefs

```
type expr = …
  | Seq of expr * expr
```

```
Seq (e1, e2) ⟹ e1; e2
```

Future Work

Support differential testing of **functors**

```
module F (M1 : S1) … (Mn : Sn) = …

module G (N1 : S1) … (Nn : Sn) = …
```

Future Work

Use **coverage-guided fuzzing** to guide Mica's QuickCheck generator

Crowbar
(OCaml '17)

FuzzChick
(OOPSLA '19)

ParaFuzz
(OCaml '21)

Mica is:

a ***PPX extension***

`[@@deriving mica]`

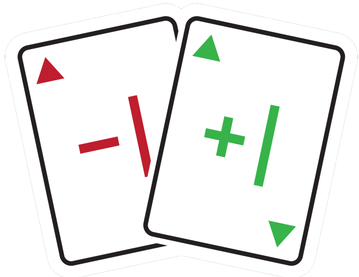
that ***automatically*** derives

PBT code

QUICKCHECK

for testing

module observational equivalence



Trying out Mica

Installation

`opam install ppx_mica`

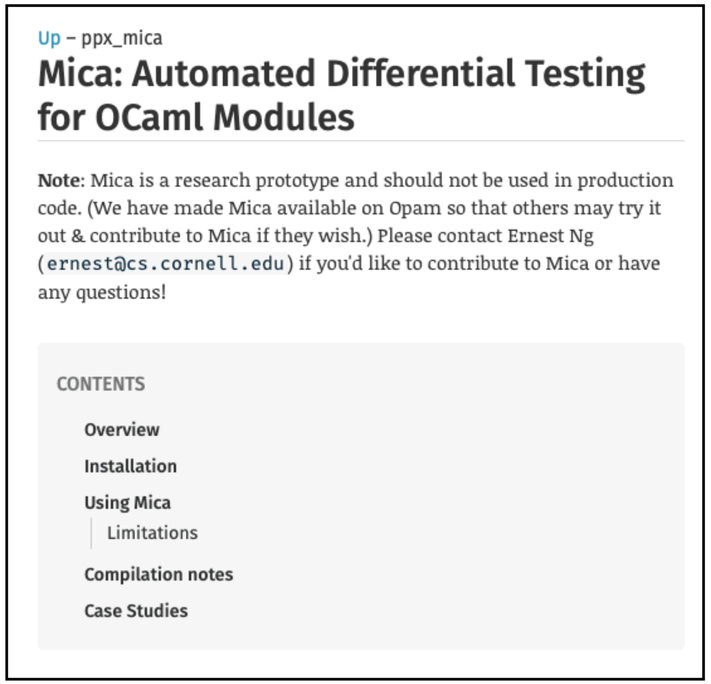
Contributions welcome!

Web Demo

(Displays code produced by Mica)

Docs

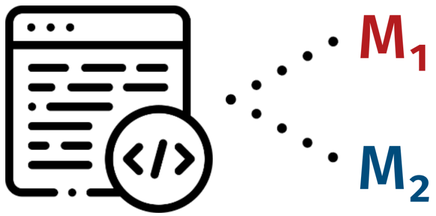
`ngernest.github.io/mica`



PPX Extension

[@@deriving **mica**]

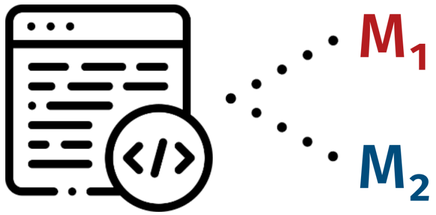
QUICKCHECK



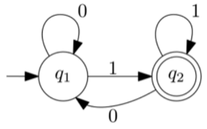
PPX Extension

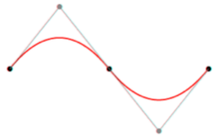
[@@deriving **mica**]

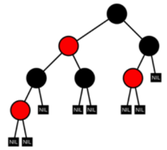
QUICKCHECK

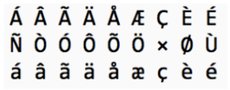


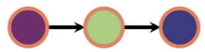
Case Studies

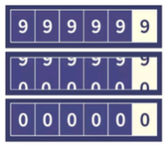


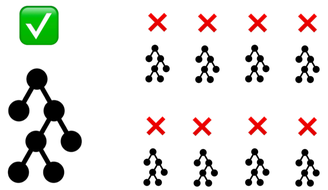


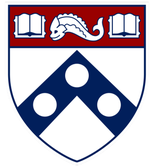


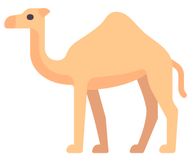








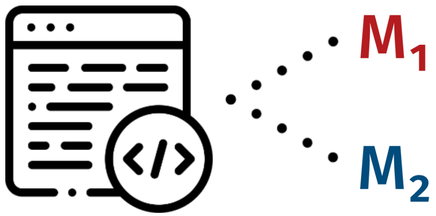




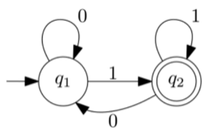
PPX Extension

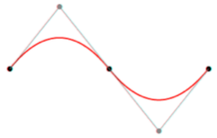
`[@@deriving mica]`

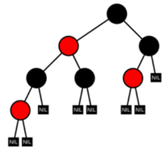
QUICKCHECK

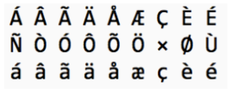


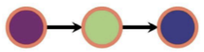
Case Studies

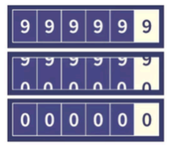


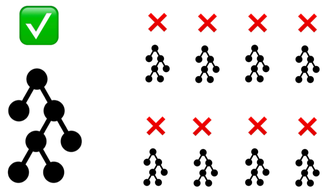


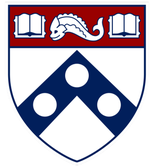












VS Code Integration

Goldstein et al. (UIST '24)

Thanks!

ernest
@cs.cornell.edu

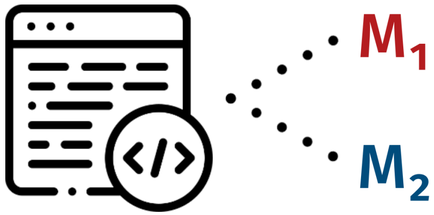
ngernest/mica

opam install ppx_mica

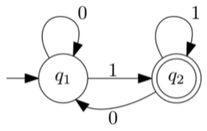
PPX Extension

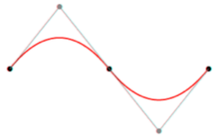
[@@deriving **mica**]

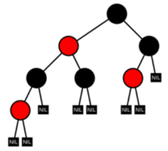
QUICKCHECK

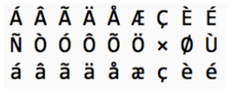


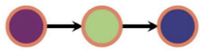
Case Studies

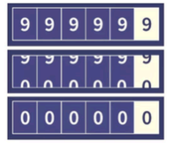


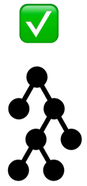


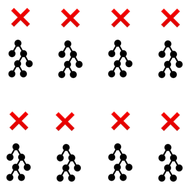


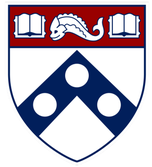


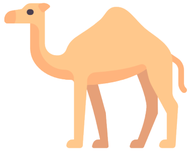












VS Code Integration

Goldstein et al. (UIST '24)

Appendix:
extrinsic vs intrinsic typing

Some frequently asked questions

Why are Mica's symbolic expressions ***extrinsically*** typed, and not ***intrinsically*** typed?

Why do you have `expr` and not `'a expr` ?

Extrinsic

Symbolic expressions: Algebraic data type

```
type expr
type ty
```

Terms and types defined separately

(It is possible to construct representations of ill-typed terms!)

Auxiliary `value` type needed for interpreter

```
interp : expr → value
```

Intrinsic

Symbolic expressions: Parameterized GADT

```
type ‘a expr
```

Terms & types are intertwined

(By construction, only representations of well-typed terms are allowed)

No auxiliary `value` type needed

```
interp : ’a expr → ‘a
```

So why not use intrinsic typing instead?

Well, it is possible to write
an intrinsically-typed interpreter
for symbolic expressions …

```
module Interpret (M : S) = struct
  (** Both [value] & [expr] are now GADTs *)
  type _ value = …
  type _ expr = …

  (** [a] is a locally abstract type — [a] is instantiated
      w/ different concrete types in the function body *)
  let eval_value (type a) (v : a value) : a = …

  (** [interp] uses polymorphic recursion *)
  let rec interp : type a. a expr → a = …
```

Intrinsic typing is non-trivial

In OCaml, writing a QuickCheck generator for random inhabitants of GADTs is hard

```
let rec gen_expr ty =
  match ty with
  | IntT → return Empty
  | Bool →
    let%bind (e : int M.t expr) = gen_expr IntT in
    let b_expr : bool expr = Is_empty e in
    return b_expr
```

```
Error: This expression has type bool expr Generator.t
       but an expression was expected of type int M.t expr Generator.t
       Type bool is not compatible with type int M.t
```

Intrinsic typing is non-trivial

It is slightly easier in Haskell, but requires existential types & higher-kinded polymorphism

```
-- an existential type
— `f` is a type constructor of kind `* → *`
data Some f where
  Exists :: f t → Some f
```

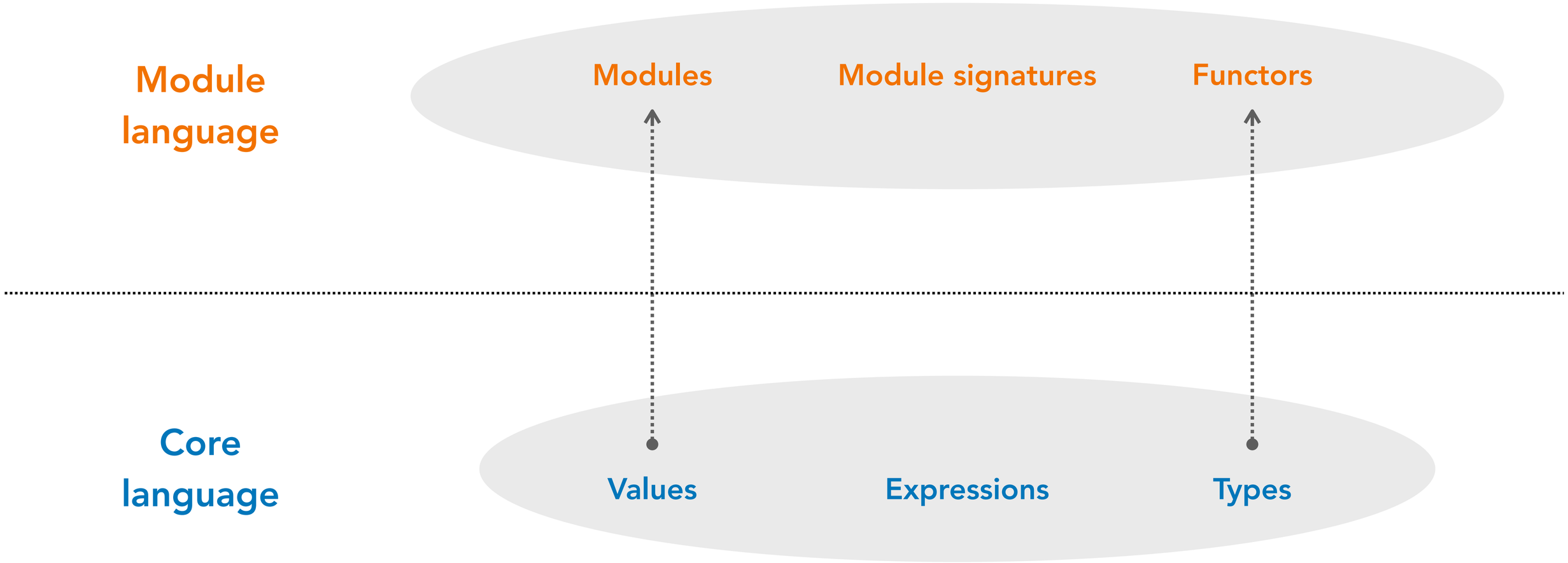
(idea due to Stephanie Weirich)

Why we used extrinsic typing

- Extrinsic typing is simpler & easier to get right
- Mica needs to derive property-based testing code automatically, for *any* possible module signature it might encounter

Other Appendix Slides

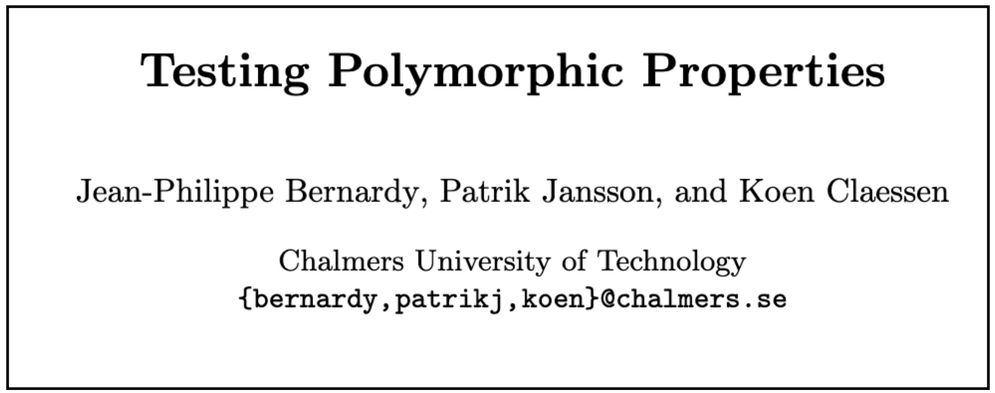
“OCaml is two languages in one”



Monomorphization

Heuristic: `'a ⇝ int`

Further reading:

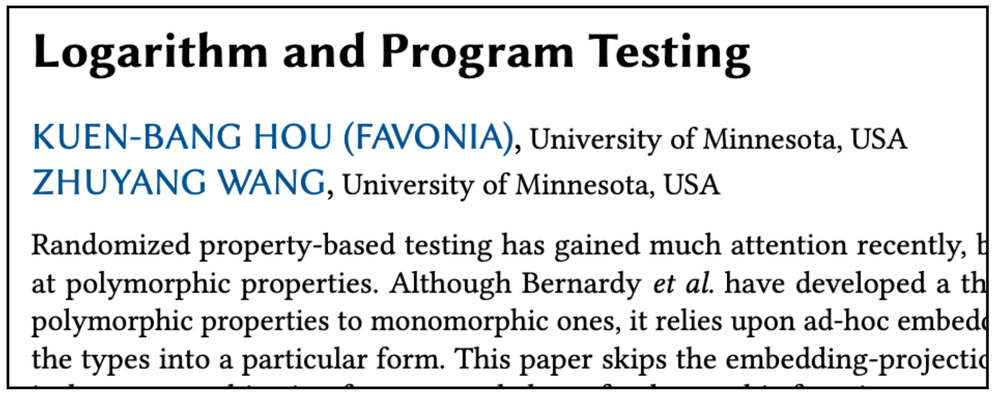


Testing Polymorphic Properties

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen

Chalmers University of Technology
{bernardy,patrikj,koen}@chalmers.se

ESOP 2010



Logarithm and Program Testing

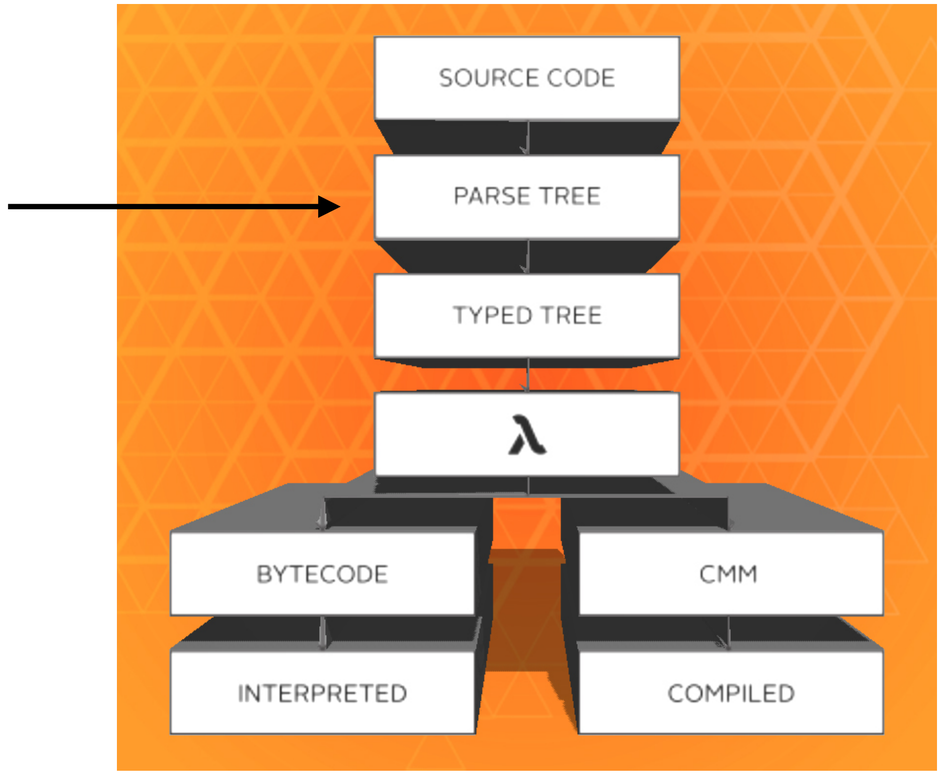
KUEN-BANG HOU (FAVONIA), University of Minnesota, USA
ZHUYANG WANG, University of Minnesota, USA

Randomized property-based testing has gained much attention recently, b
at polymorphic properties. Although Bernardy *et al.* have developed a th
polymorphic properties to monomorphic ones, it relies upon ad-hoc embedd
the types into a particular form. This paper skips the embedding-projectio

POPL 2022

PPXes (PreProcessor Extensions)

- Preprocessors that operate on the untyped AST (`Parsetree`) produced by the OCaml compiler
- PPX derivers = `ast_node` → `ast_node` functions
- Similar facilities exist in Haskell / Rust

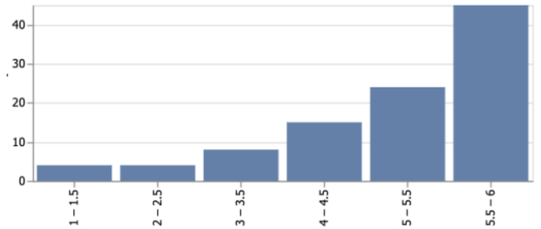


Interacting with **Mica** + **Tyche**

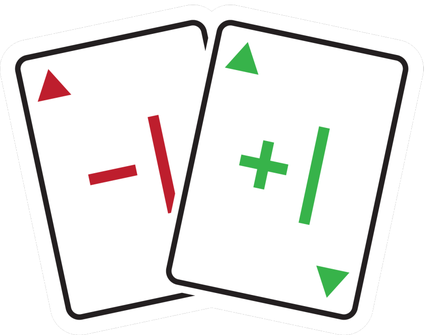
1. Annotate module signature & invoke Mica test harness

```
module type S = …
[@@deriving mica]
```

2. Tyche visualizes test statistics



3. Examine test results



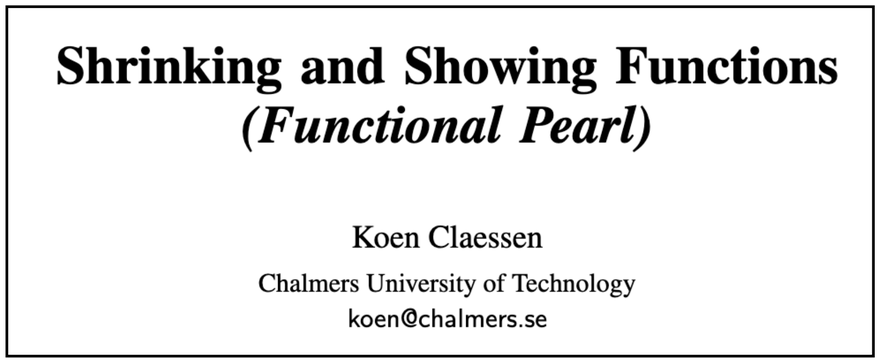
4. Update module implementations

```
module M1 : S = …
module M2 : S = …
```

Representing Higher-Order Functions Using Symbolic Expressions

```
map : ('a → 'b) → 'a t → 'b t
```

```
type expr = Map of (int → int) * expr | …
```

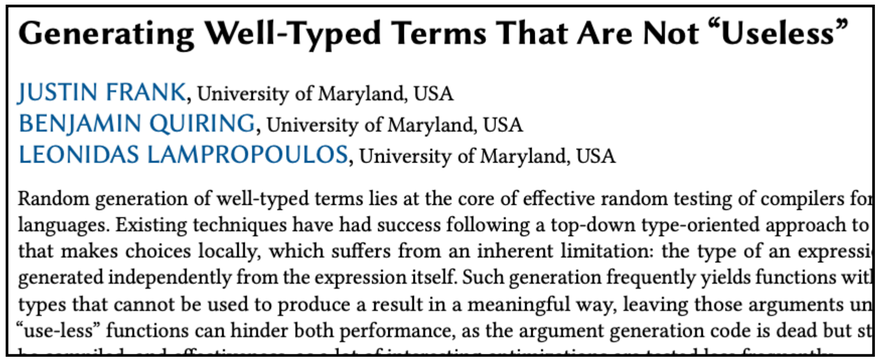


Shrinking and Showing Functions
(Functional Pearl)

Koen Claessen

Chalmers University of Technology
koen@chalmers.se

Haskell Symposium 2012



Generating Well-Typed Terms That Are Not "Useless"

JUSTIN FRANK, University of Maryland, USA
BENJAMIN QUIRING, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Random generation of well-typed terms lies at the core of effective random testing of compilers for languages. Existing techniques have had success following a top-down type-oriented approach to that makes choices locally, which suffers from an inherent limitation: the type of an expressi generated independently from the expression itself. Such generation frequently yields functions with types that cannot be used to produce a result in a meaningful way, leaving those arguments un "use-less" functions can hinder both performance, as the argument generation code is dead but st

POPL 2024

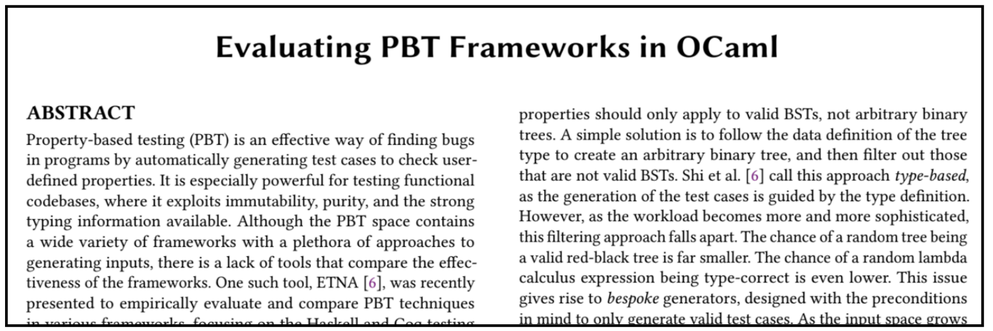
Supporting other PBT libraries besides `Core.Quickcheck`

- Mica's design is ***library-agnostic***: developers can write other backends that support other OCaml PBT libraries (e.g. QCheck, Crowbar, …)
  - (We picked `Core.Quickcheck` just because we were most familiar with it)
- It'd be interesting to build on recent work extending **Etna** (an evaluation platform for different PBT frameworks) for comparing the efficacy of different OCaml PBT libraries

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

ICFP 2023



Evaluating PBT Frameworks in OCaml

ABSTRACT

Property-based testing (PBT) is an effective way of finding bugs in programs by automatically generating test cases to check user-defined properties. It is especially powerful for testing functional codebases, where it exploits immutability, purity, and the strong typing information available. Although the PBT space contains a wide variety of frameworks with a plethora of approaches to generating inputs, there is a lack of tools that compare the effectiveness of the frameworks. One such tool, ETNA [6], was recently presented to empirically evaluate and compare PBT techniques in various frameworks, focusing on the Haskell and Coq testing

properties should only apply to valid BSTs, not arbitrary binary trees. A simple solution is to follow the data definition of the tree type to create an arbitrary binary tree, and then filter out those that are not valid BSTs. Shi et al. [6] call this approach *type-based*, as the generation of the test cases is guided by the type definition. However, as the workload becomes more and more sophisticated, this filtering approach falls apart. The chance of a random tree being a valid red-black tree is far smaller. The chance of a random lambda calculus expression being type-correct is even lower. This issue gives rise to *bespoke* generators, designed with the preconditions in mind to only generate valid test cases. As the input space grows

PLDI 2024 SRC

Compilation Times + How long it takes Mica's tests to run

In practice, we haven't found compilation / test runtimes to be an issue!

Module Signature	Compilation Times (using a Mica prototype)	Runtime of PBT test harness
Sets	309.25 µs	2.55 ns
Stacks	361.08 µs	2.54 ns
Polynomials	302.82 µs	2.57 ns
Maps	262.84 µs	2.56 ns
Regexes	266.61 µs	2.57 ns

(Measured using `Core_bench` on an M1 Mac)

How to Specify It (BST Case Study) Stats

Bug revealed only in one branch of a pattern-match:
coverage information would help us here!

	Bug #1	Bug #2	Bug #3	Bug #4	Bug #5	Bug #6	Bug #7	Bug #8
Min	6	8	504	7	42	10	17	20
Mean	20	62	553	20	286	44	163	229
Max	118	262	765	94	546	238	312	438

Fig. 3. Average mean no. of trials required to provoke failure in an observational equivalence test

Invoking QuickCheck generators for opaque types

- For any user-defined type `t`, the user should provide a QuickCheck generator called `quickcheck_generator_t`
- Mica will then invoke this generator by calling the appropriate directive from `ppx_quickcheck` in the derived code

```
let rec gen_expr (ty : ty) : expr Generator.t =
  match ty, QC.size with
  | (T, _) → …
    let%bind t = [%quickcheck.generator: t] in …
```

Related Work

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

Related Work

QCSTM
(Midtgaard 2020)

Model_quickcheck
(Dumont 2020)

- Algebraic data types for representing symbolic expressions
- Mica adds support for binary operations on abstract types

Future Work (Engineering)

Contact us if you're interested in contributing to Mica!

- Shrinking
- Modules with multiple abstract types
- Compute "module coverage" for tests
- Support other OCaml PBT libraries

ernest@cs.cornell.edu