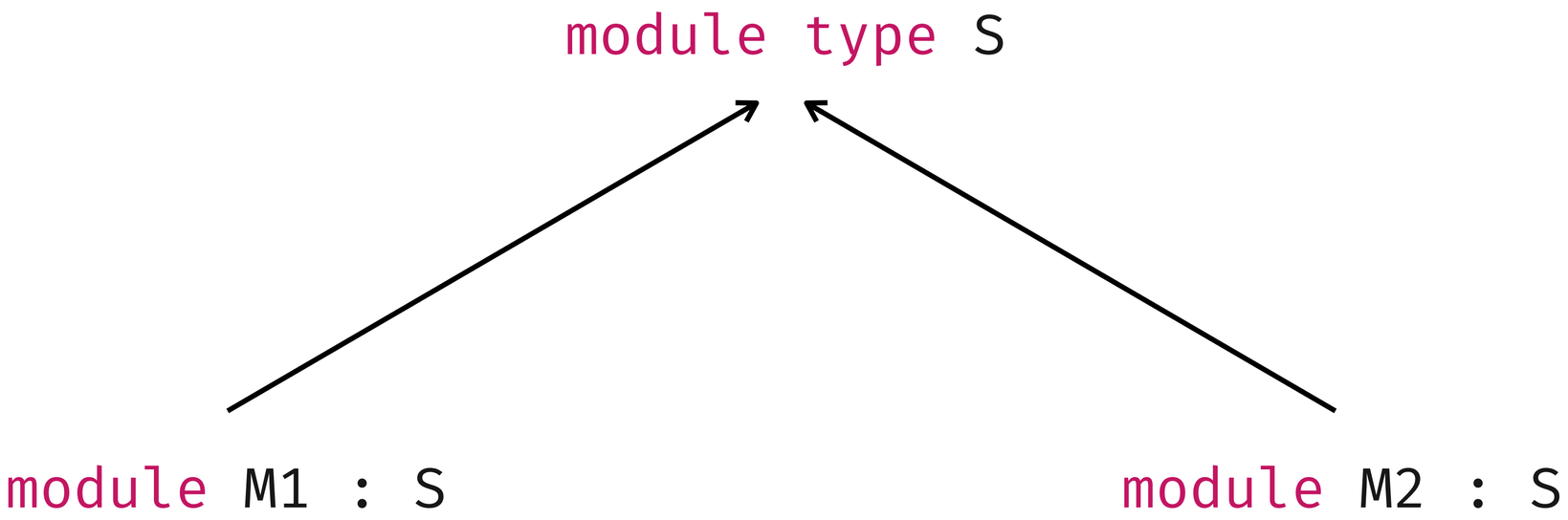
Mica

Automated Differential Testing for OCaml Modules

Ernest Ng
Harry Goldstein
Benjamin Pierce

Representation Independence



Example: Finite Sets

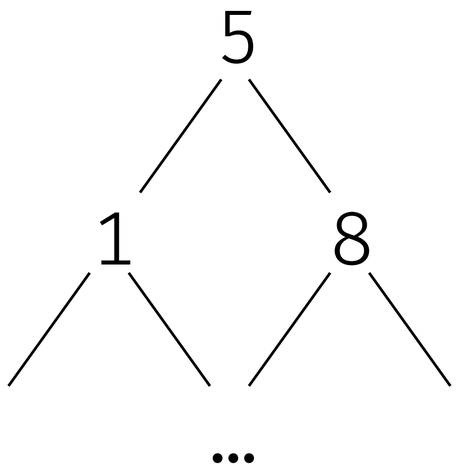
```
module type S = sig
  type 'a t
  val empty  : 'a t
  val insert : 'a → 'a t → 'a t
  …
end
```

```
module type S
```

{1, 5, 8, …}

```
[1; 5; 8; …]
```

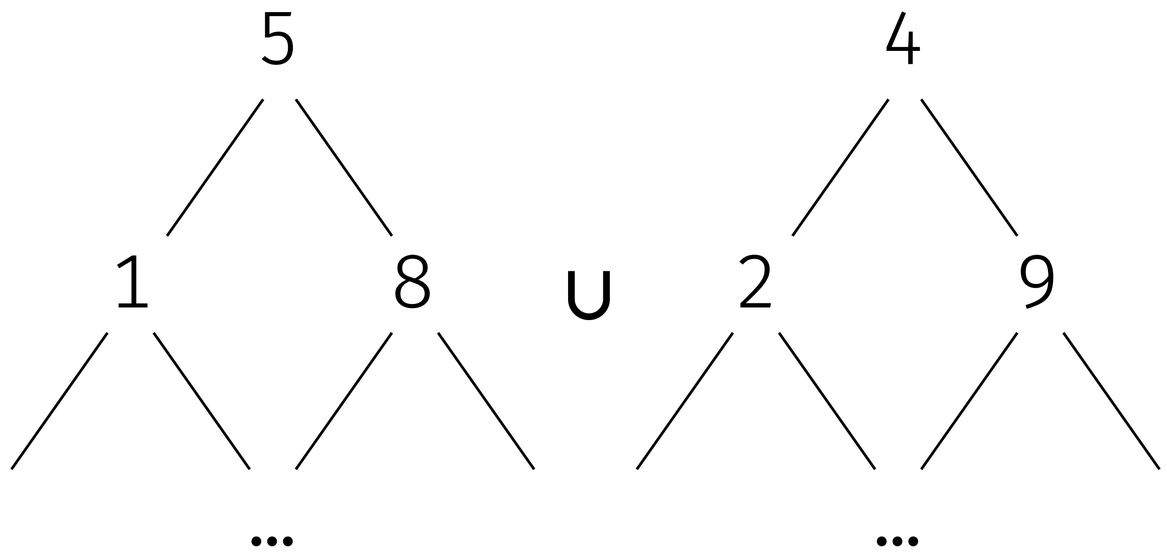
```
module ListSet : S
```




```
module BSTSet : S
```

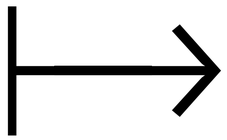
Are these ***equivalent***?

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

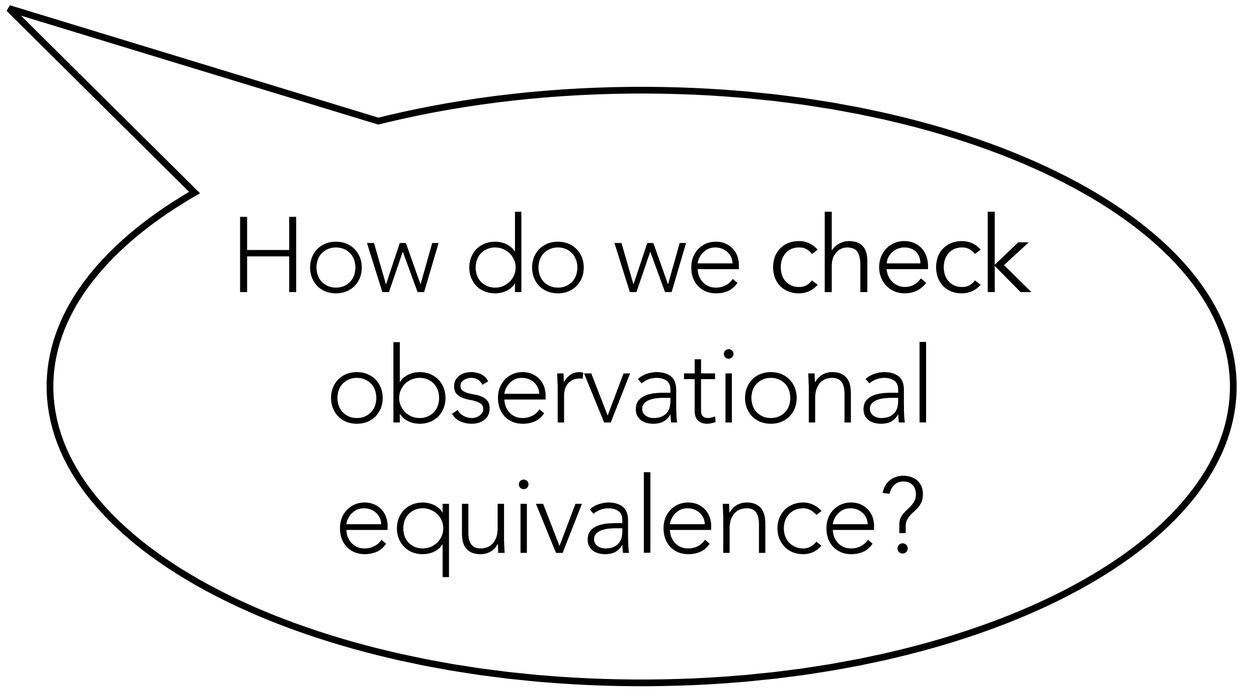


Observational Equivalence

equivalent
inputs



equivalent
outputs



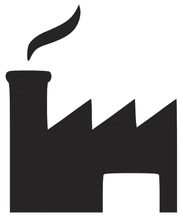
We can use
property-based testing!

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

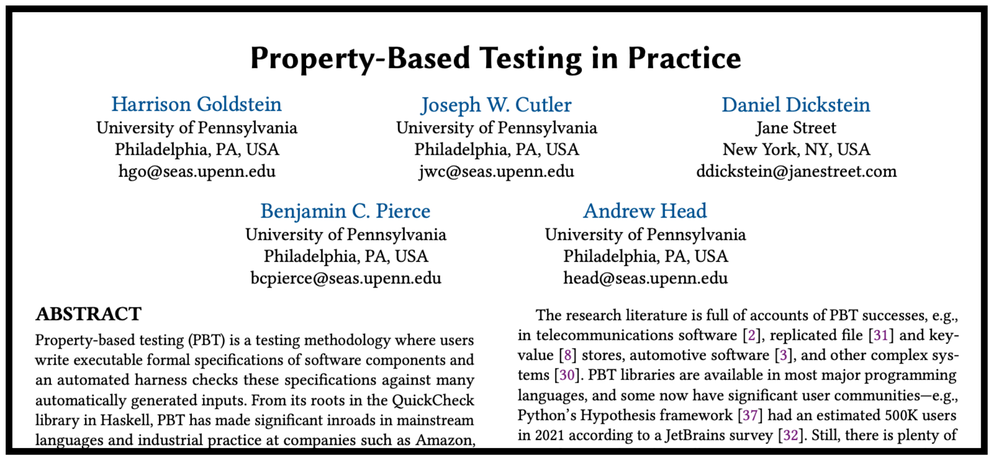
Property-Based Testing

QuickCheck

(Claessen & Hughes ’00)

PBT in Practice

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

PBT in Practice

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

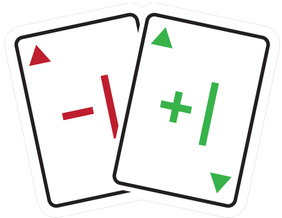
ABSTRACT

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

PBT tools are often used for differential testing!

~60% of developers wrote **differential properties**



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 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…

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]
```

```
type expr = …
let gen_expr ty = …
let interp expr = …
```

Jane Street

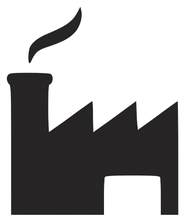
QUICKCHECK

Mica derives the following automatically:

Types

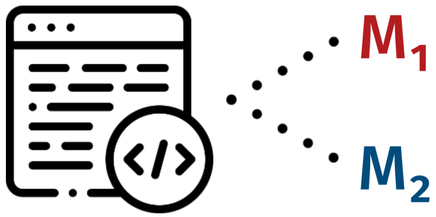
```
expr
ty
value
```

QuickCheck Generator

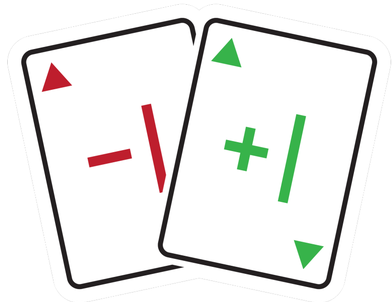


$\rightsquigarrow$ e_1 e_2 …

Interpreter

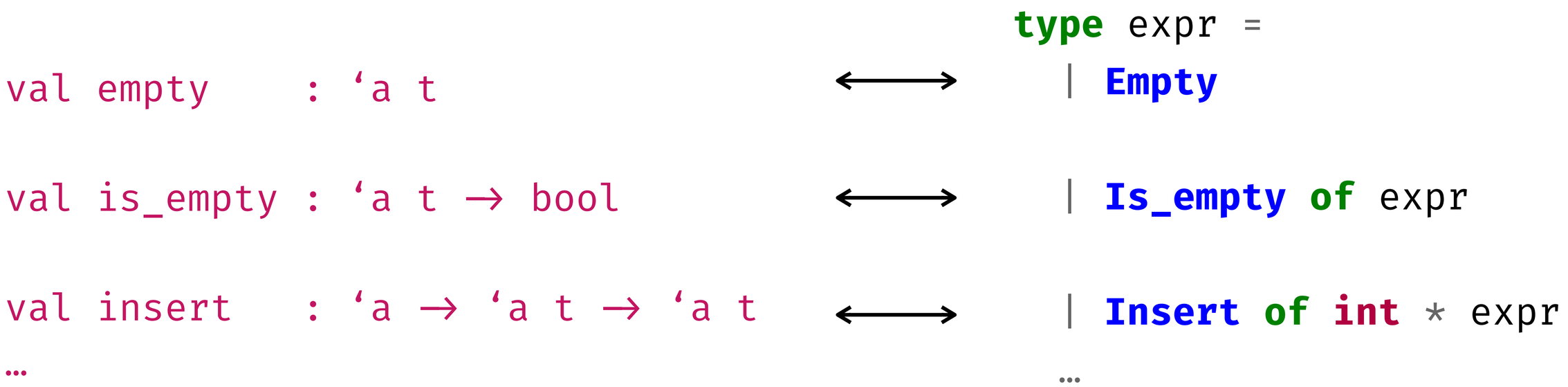


Test Harness



Symbolic Expressions

Inductively-defined algebraic data type



Symbolic Expressions

types

```
type ty = …
```

values

```
type value = …
```

QuickCheck Generator

random, **well-typed** symbolic expressions

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

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

✓

```
Is_empty (Size Empty)
```

X

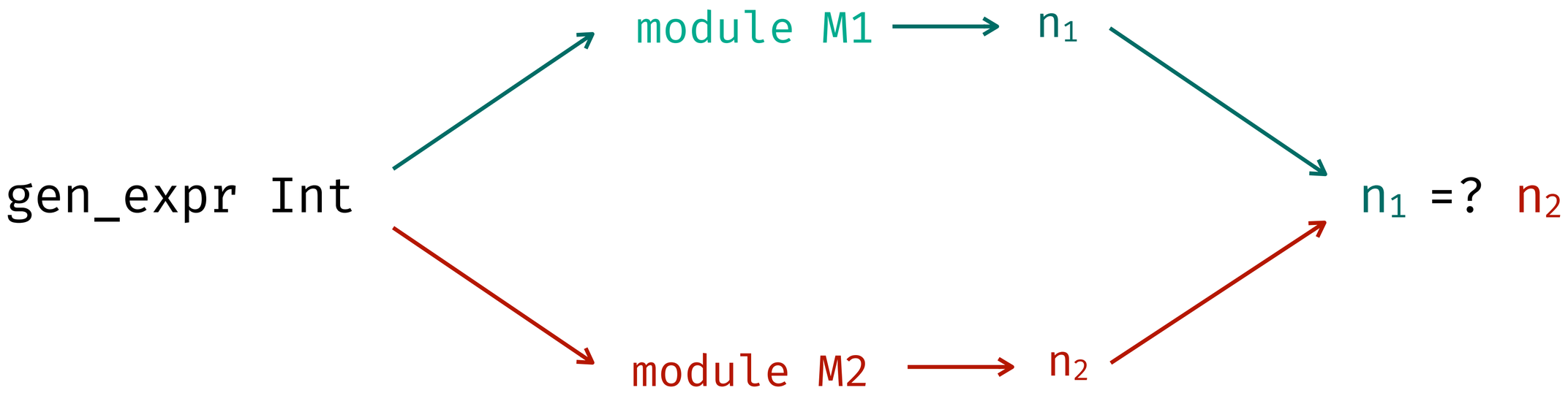
Interpretation Functor

```
module Interpret (M : S) = …
```

```
expr    ⟶    value

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

Test Harness Functor

Checks observational equivalence at **concrete** types



Test Harness Functor

Checks observational equivalence at **concrete** types

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

`int` ✓

`'a t` ✗

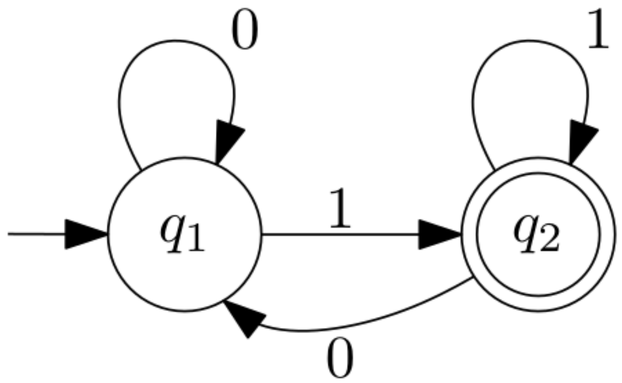
Case Studies

Regular Expressions

Brzozowski Derivatives

$$
\begin{aligned}
D_c(\emptyset) &= \emptyset \\
D_c(\epsilon) &= \emptyset \\
D_c(c') &= \begin{cases} \epsilon_c & \text{if } c = c' \\ \emptyset & \text{if } c \neq c' \end{cases} \\
D_c(L_1 \cup L_2) &= D_c(L_1) \cup D_c(L_2) \\
D_c(L_1 \circ L_2) &= \begin{cases} D_c(L_1) \circ L_2 & \text{if } \epsilon \notin [\![L_1]\!] \\ (D_c(L_1) \circ L_2) \cup D_c(L_2) & \text{if } \epsilon \in [\![L_1]\!] \end{cases}
\end{aligned}
$$

Finite Automata



Unsigned Integer Arithmetic

andrenth/**ocaml-stdint**

Various signed and unsigned integers for OCaml

15 Contributors · 21 Issues · 83 Stars · 15 Forks

yallop/**ocaml-integers**

Various signed and unsigned integer types for OCaml

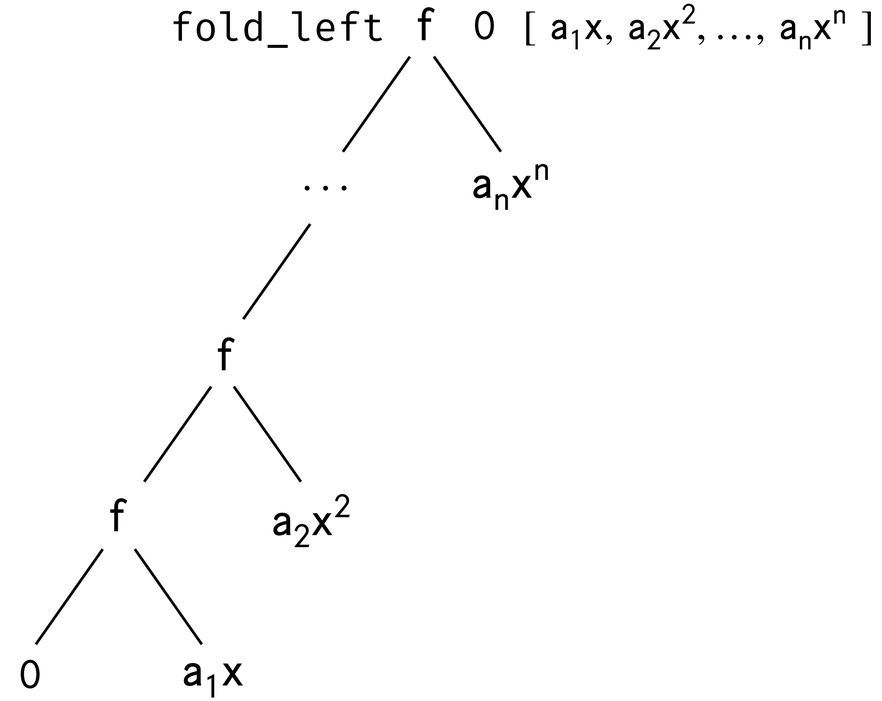
8 Contributors · 5 Issues · 2 Discussions · 60 Stars · 20 Forks

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 monomials



Character Sets

Standard Library

```
Set.Make(Char)
```

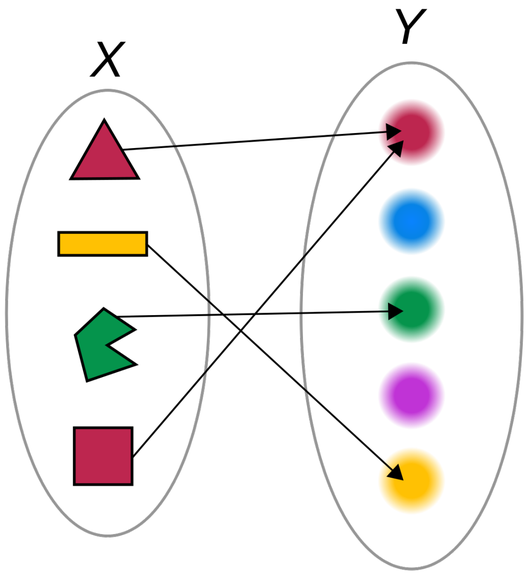
Compiler Intrinsics

yallop/**ocaml-charset**

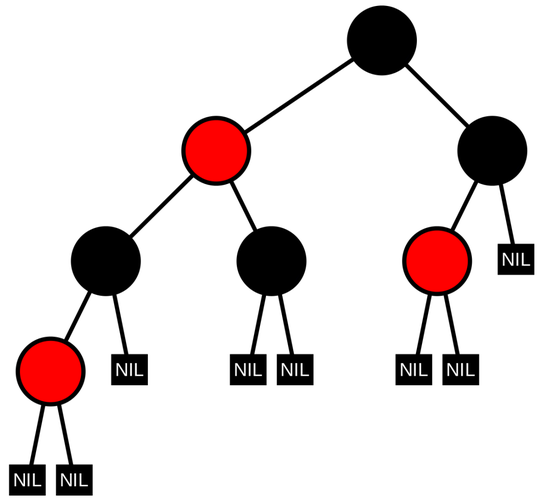
Fast char sets

Persistent Maps

Association Lists



Red-Black Trees



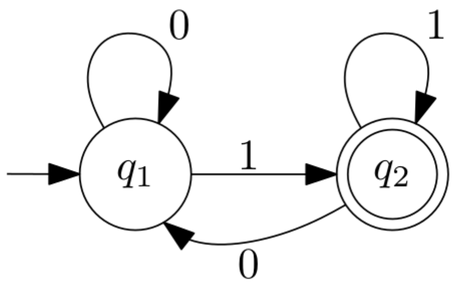
Ephemeral Queues

`Base.Queue`

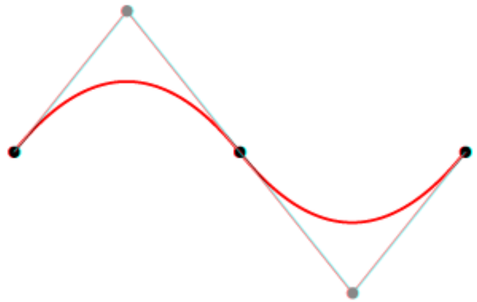
`Base.Linked_queue`

Case Studies

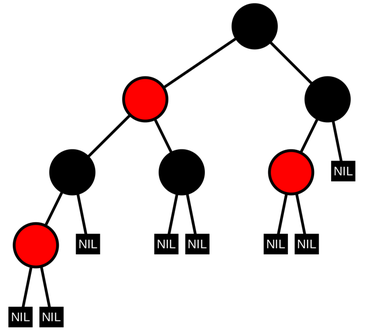
Regex Matchers



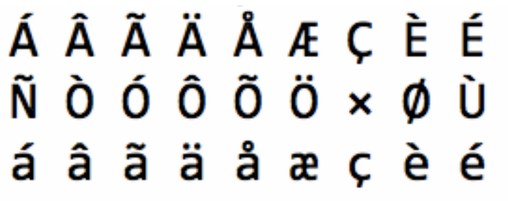
Polynomials



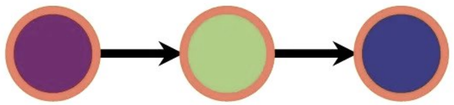
Persistent Maps



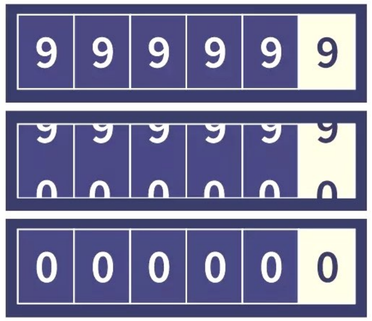
Character Sets



Ephemeral Queues

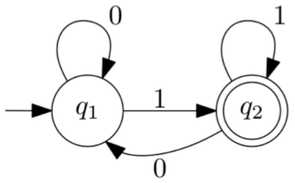


Unsigned Integers

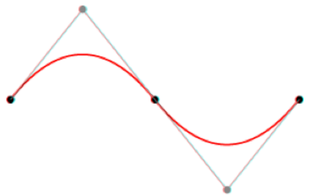


35 manually-inserted bugs caught

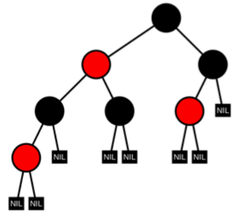
Regex Matchers



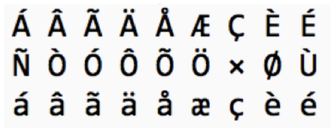
Polynomials



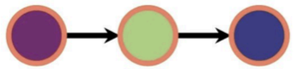
Persistent Maps



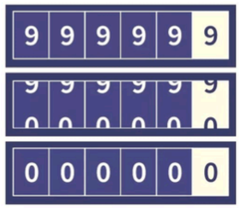
Character Sets



Ephemeral Queues



Unsigned Integers



6 real-world OCaml libraries

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

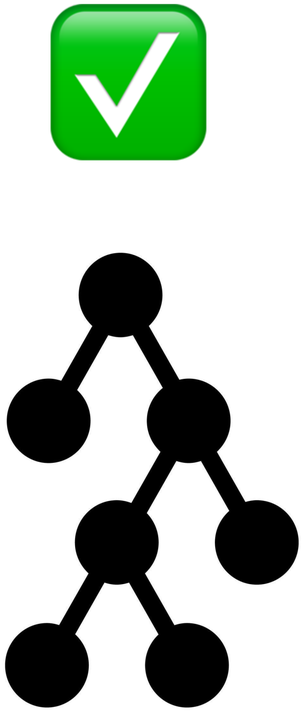
John Hughes

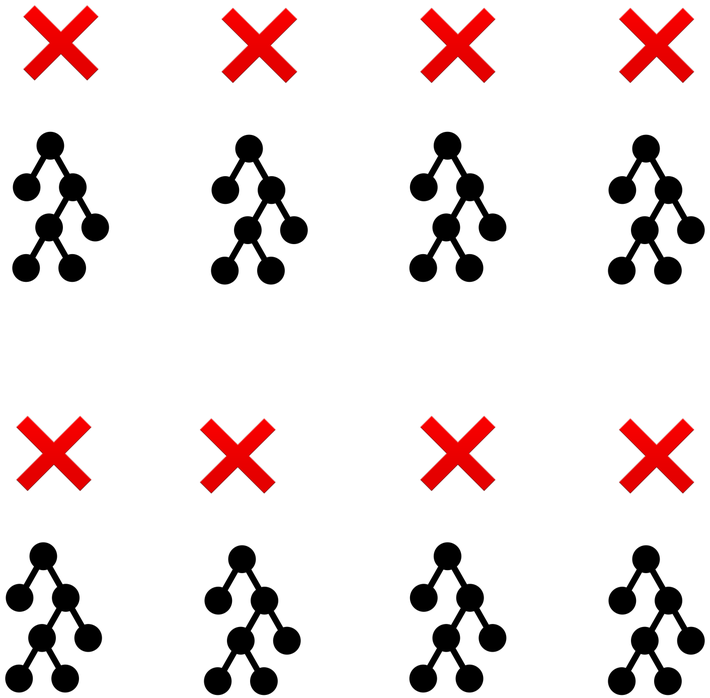
How to Specify it!

A Guide to Writing Properties of Pure Functions.

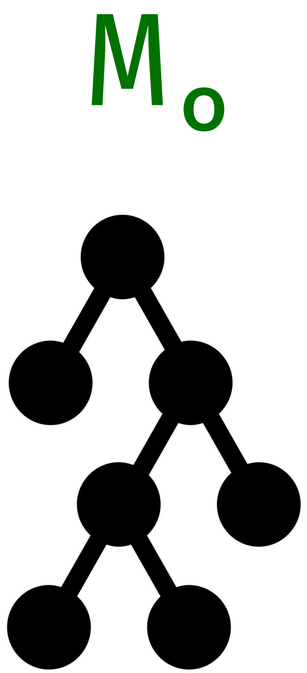
(TFP '19)

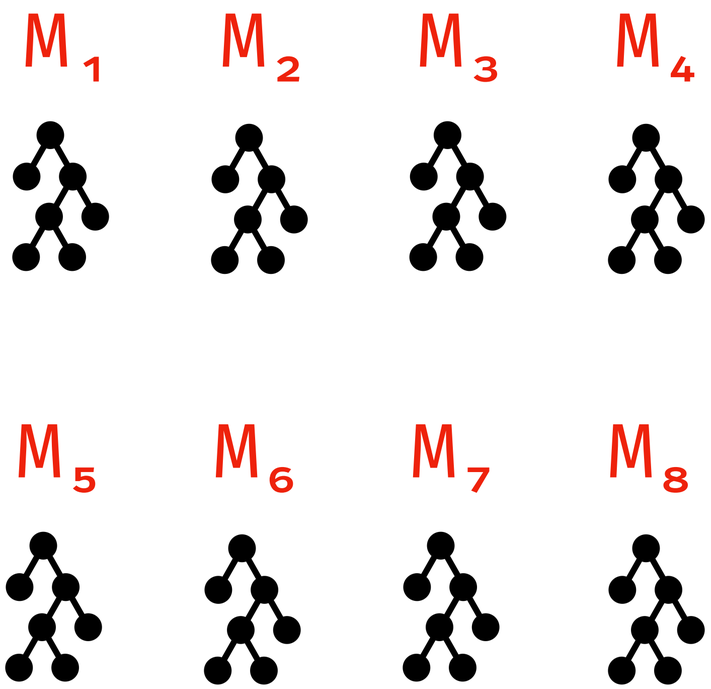
Case study: **BSTs done 9 ways**





Case study: **BSTs done 9 ways**





Each bug caught within ~170 random inputs (on average)

VS Code Integration with **Tyche**

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

TYCHE: Making Sense of Property-Based Testing Effectiveness

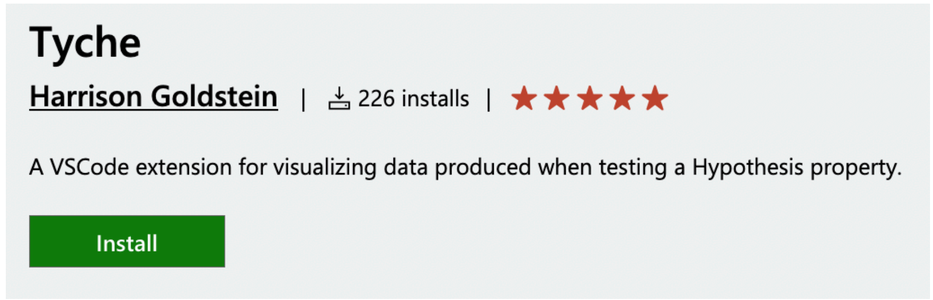
Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

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

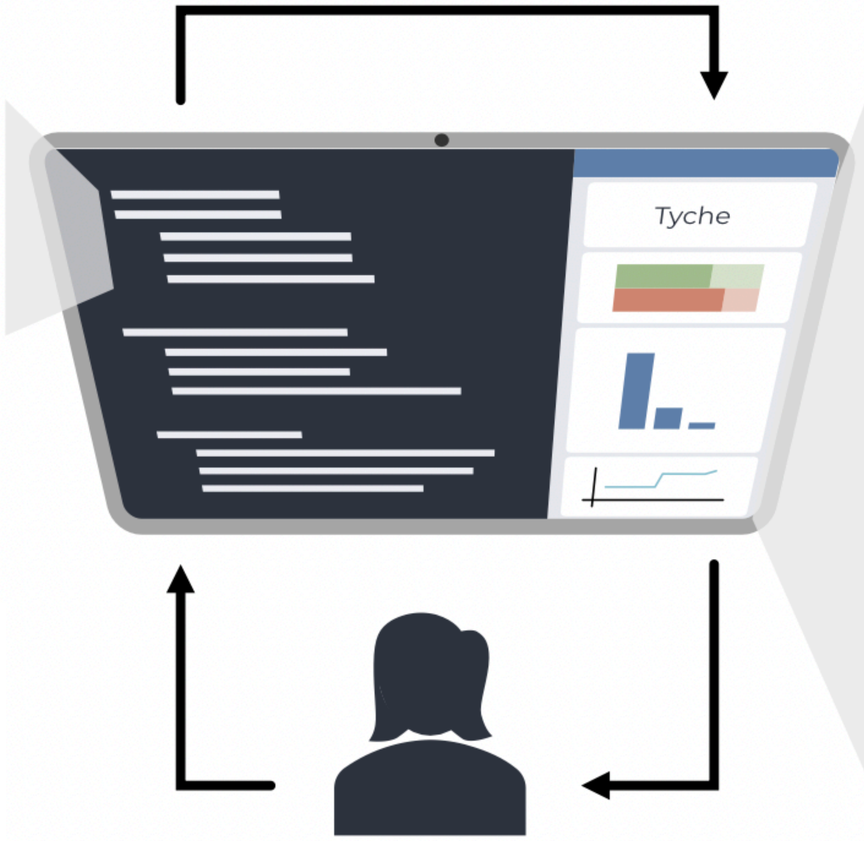
Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

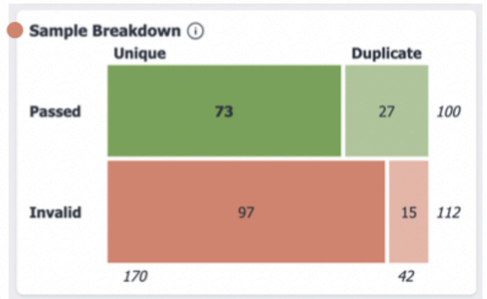


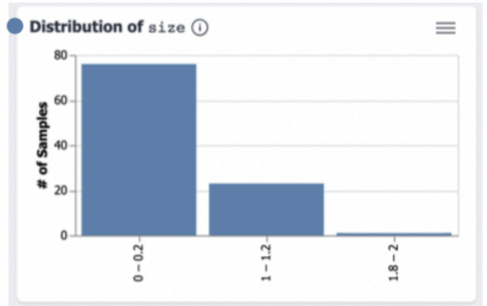
VS Code Integration with **Tyche**

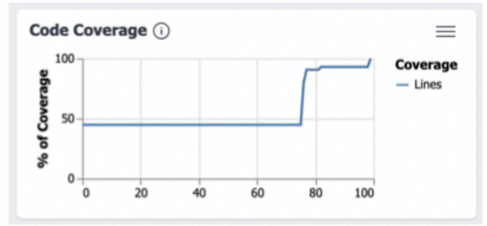
```
module type S = … [@@deriving mica]

TestHarness(M1)(M2).run_tests ();;
```

Tyche visualizes test metadata produced by Mica



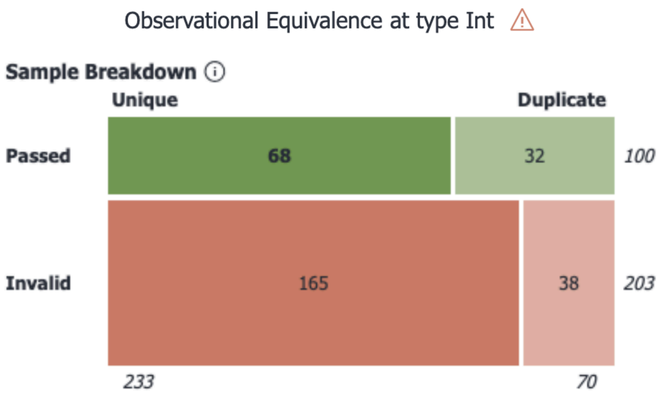




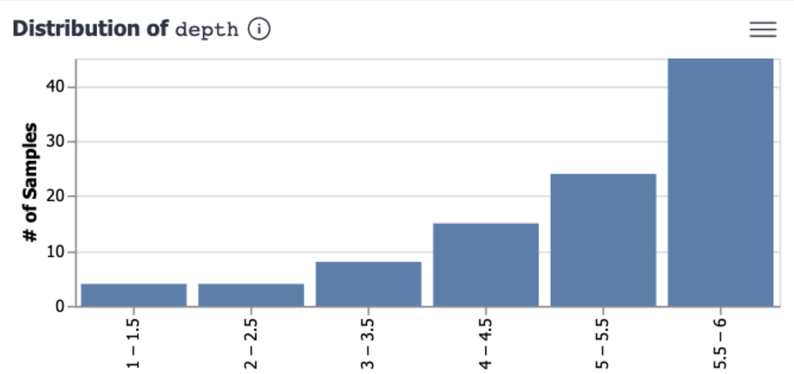


Using **Tyche** to visualize **Mica**'s test results

Observational equivalence test results

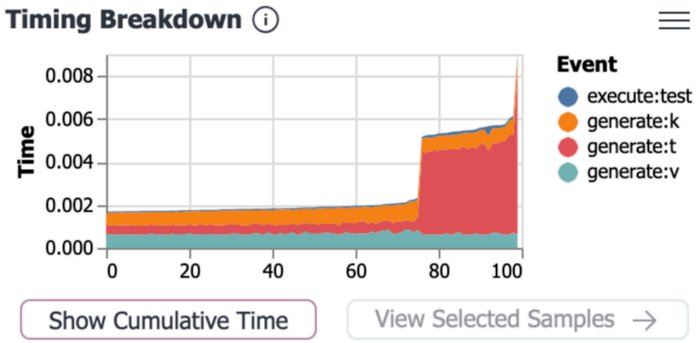


Distribution of symbolic expressions




```
(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

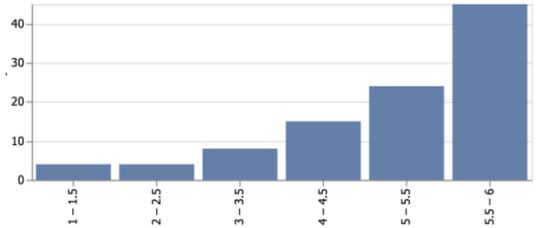


Interacting with **Mica** + **Tyche**

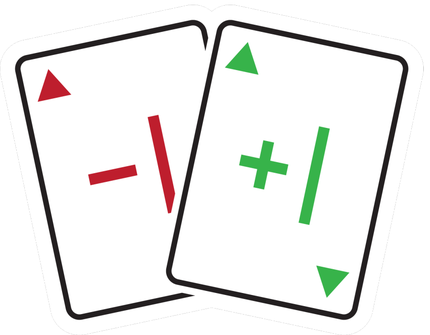
1. Annotate module signature & invoke Mica test harness

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

2. Tyche visualizes test statistics



3. Examine test results



4. Update module implementations

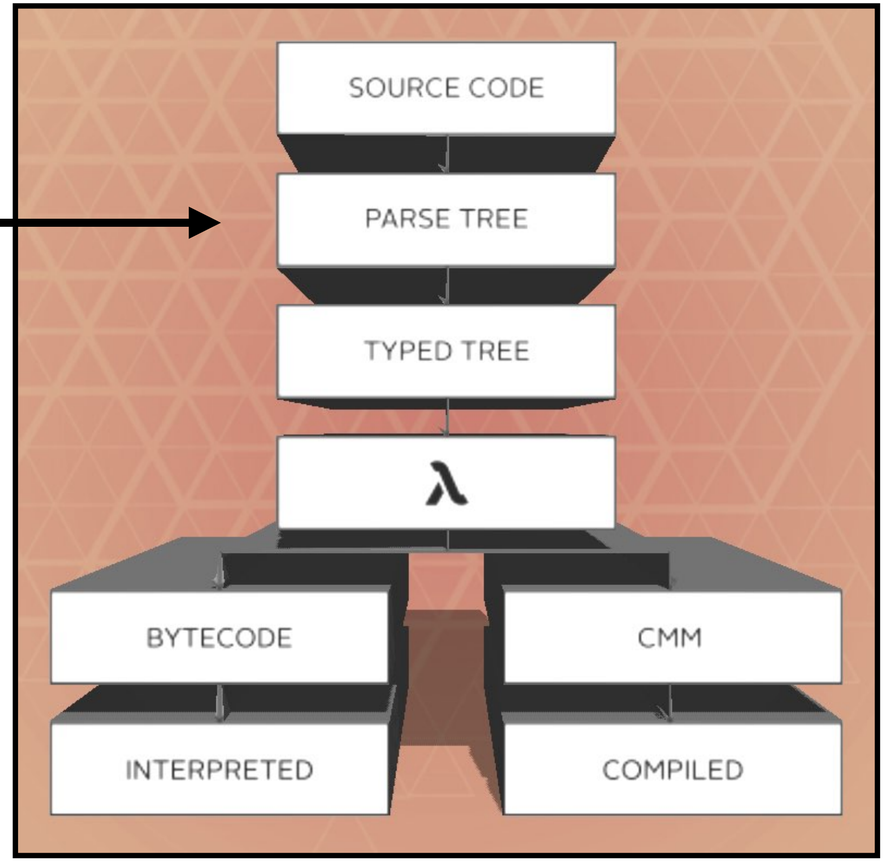
```
module M1 : S = …
module M2 : S = …
```

Engineering

`Ppxlib`

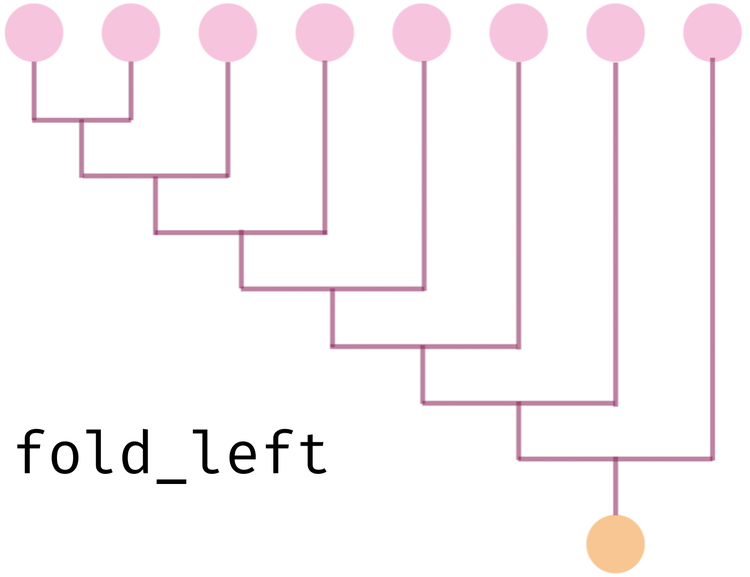
Produce new AST nodes

~3500 LoC



Future Work

Support more **higher-order functions**



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)

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

eyn5@cornell.edu

Mica is:

a ***PPX extension***

`[@@deriving mica]`

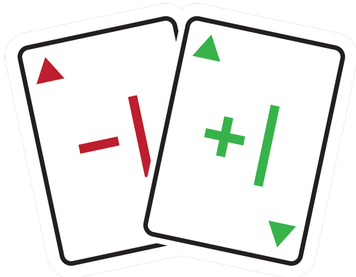
that ***automatically*** derives

PBT code

QUICKCHECK

for testing

module observational equivalence



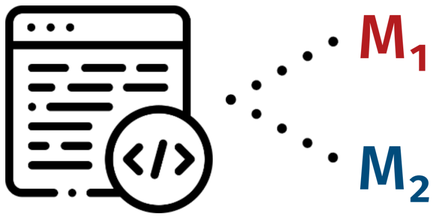
ngernest/mica

Thank you!

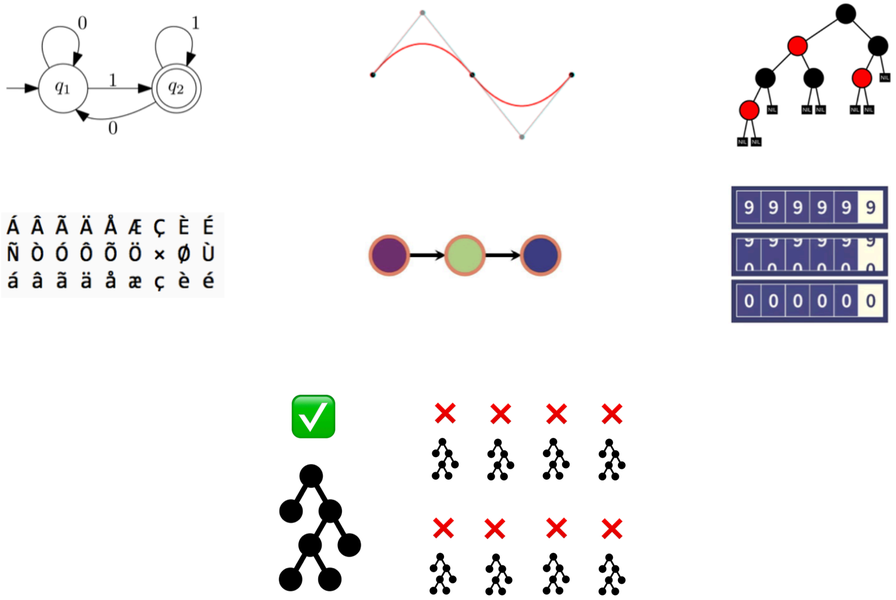
PPX Extension

[@@deriving **mica**]

QUICKCHECK



Case Studies



VS Code Integration

Goldstein et al. (UIST '24)

Appendix

Monomorphization + Higher-Order Functions for Symbolic Expressions

```
'a ⇝ int
```

```
map : ('a → 'b) → 'a t → 'b t
```

```
Map of (int → int) * expr
```

BST Case Study Statistics

	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

Intrinsically-typed Symbolic Expressions via GADTs: Attempt

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

```
module Interpret (M : S) = struct

  (** GADT for intrinsically typed symbolic expressions *)
  type _ expr =
    | Empty : int M.t expr
    | Is_empty : int M.t expr → bool expr
    …

end
```

Intrinsically-typed Symbolic Expressions via GADTs: Attempt

```
type _ expr =
  | Empty : int M.t expr
  | Is_empty : int M.t expr → bool expr
  …
```

```
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
```

Interpreting GADT-based intrinsically-typed exprs

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

```
module Interpret (M : S) = struct
  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 =
    match v with
    | ValInt x → x
    | ValIntT intT → intT
    …

  (** [interp] uses polymorphic recursion *)
  let rec interp : type a. a expr → a =
   fun expr →
       match expr with
       | Value v → eval_value v
       | Empty → M.empty
       | Is_empty e →
         let b = M.is_empty (interp e) in
         eval_value (ValBool b)
       …
```

Invoking QuickCheck generators for opaque types

```
module type MapInterface = 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%bind xs = [%quickcheck.generator: AssocList.t] in
    G.return @@ From_list xs
  …
```

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