MICA: Automated Differential Testing for OCaml Modules

ERNEST NG*, University of Pennsylvania and Cornell University, USA
HARRISON GOLDSTEIN*, University of Pennsylvania and University of Maryland, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

1 INTRODUCTION

Suppose we are given two OCaml modules implementing the same signature. How do we check that they are *observationally equivalent*—that is, that they behave the same on all inputs? One established way is to use a *property-based testing* (PBT) tool such as QuickCheck [7]. Currently, however, this can require significant amounts of boilerplate code and ad-hoc test harnesses [21].

We present MICA, an automated tool for testing equivalence of OCaml modules. MICA is implemented as a PPX compiler extension [37], allowing users to supply minimal annotations to a module signature. These annotations guide MICA to automatically derive specialized PBT code that checks observational equivalence using Jane Street's `Core.Quickcheck` library [12]. A MICA prototype is available on GitHub;[1] we are currently reimplementing MICA's concrete syntax as a PPX extension (as described below).

2 DESIGN OF MICA

Suppose we have two modules `ListSet` and `BSTSet` that implement finite sets (signature `S`) using lists and binary search trees (BSTs), respectively. To test for observational equivalence, users invoke MICA by annotating `S` with the directive `[@@deriving mica]`. During compilation, MICA derives the definition for an inductively-defined algebraic data type (ADT) called `expr`, which represents *symbolic expressions*. Each declaration in `S` corresponds to a constructor for the `expr` ADT with the same name, arity and argument types. MICA also derives auxiliary ADTs that represent the possible *types* and *values* of symbolic expressions.

*Work done while at the University of Pennsylvania.
[1] https://github.com/ngernest/mica

```
(* User code *)

(* Signature for finite sets *)
module type S = sig
  type 'a t
  val empty  : 'a t
  val insert : 'a → 'a t → 'a t
  ...
end
[@@deriving mica, ...]

(* Modules under test *)
module ListSet : S = ...
module BSTSet  : S = ...

(* Users invoke Mica's test harness on
   the modules they wish to test *)
module T = TestHarness(ListSet)(BSTSet)
let () = T.run_tests ()
```

```
(* Code produced by Mica *)
(* Symbolic expressions *)
type expr = Empty | Insert of int * expr | ...
type ty = Int | IntT | ...

(* QuickCheck generator for [expr]s *)
let rec gen_expr : ty → expr Generator.t = ...

(* Interpretation functor *)
module Interpret (M : S) = struct
  type value = ValInt of int | ValIntT of int M.t | ...

  (* Interprets an [expr] over module [M] *)
  let rec interp : expr → value = ...
end

(* Functor for differential testing of [M1] & [M2] *)
module TestHarness (M1 : S) (M2 : S) = struct
   let run_tests : unit → unit = ...
```

Fig. 1. Left: User code (note the annotation on signature S). Right: PBT code automatically derived by MICA.

To generate random symbolic expressions, Mica derives a recursive QuickCheck generator `gen_expr` that is parameterized by the desired type of the expression. The type-directed nature of this generator ensures that only well-typed expressions are produced. Subsequently, to interpret symbolic expressions over a specific module `M` and produce concrete `values`, Mica produces an interpretation functor that is parameterized by an instance of `S`.

To check for observational equivalence, Mica produces a functor `TestHarness` which users instantiate with the desired modules. Crucially, Mica's test harness only compares the `value` of interpreted `exprs` at *concrete types*, for example `int`, not the abstract type `'a t`, since the internal representations of such values may differ arbitrarily.

To test modules with mutable internal state, the `expr` datatype is extended with a constructor `Seq`, where `Seq(e1, e2)` represents the *sequencing* of expressions `e1` and `e2`. Also, we are currently working on extending Mica with the ability to derive constructors that represent `let`-expressions. This addition will allow `exprs` to refer to previously generated data, encoding dependencies between successive function calls.

To test polymorphic functions, Mica instantiates all type variables `'a` with `int`, following well-known heuristics [2, 23]. Additionally, Mica offers support for generating unary anonymous functions. For example, to test the polymorphic higher-order function `map`, Mica derives the symbolic expression `Map of (int → int) * expr`, generating a random `int → int` function in the process using canonical techniques from the PBT literature [6, 12].

3 EDITOR INTEGRATION

We have integrated Mica with Tyche [22], an extension to VSCode for visualizing the behavior of PBT generators. Whenever Mica checks two modules for observational equivalence at type τ, Tyche plots numeric features regarding the random `exprs` of type τ that were used to test the two modules. For example, Tyche serializes the individual `exprs` generated and also visualizes the distribution of their depth, thereby giving users greater insight into the effectiveness of Mica's testing. We refer the reader to Goldstein et al.'s work for further details regarding Tyche.

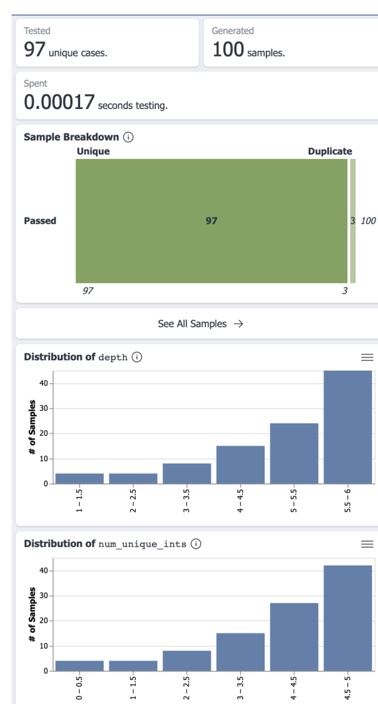

Fig. 2. The Tyche user interface, displaying Mica's test results

4 CASE STUDIES

To examine Mica's efficacy as a testing tool, we applied it with various module signatures that admit multiple implementations, including:

- Regular expression matchers (Brzozowski derivatives, deterministic finite automata) [14, 20]
- Jane Street's imperative `Base.Queue` and `Base.Linked_queue` modules [29]
- Character sets, implemented respectively using the standard library's `Set.Make(Char)` module and the `charset` library (a specialized implementation that uses compiler intrinsics for efficiency purposes) [41]
- Polynomials (Horner schema, monomials) [15, 19]
- Finite maps (red-black trees, association lists) [8, 33]
- Unsigned 32 & 64-bit integer arithmetic (the `stdint` and `ocaml-integers` libraries) [32, 40]

	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

MICA was able to found 35 manually-inserted bugs inserted across these modules without any user input required.

We have also replicated a case study from John Hughes's *How to Specify It* [25], an extended tutorial on Haskell QuickCheck which uses BSTs representing finite maps as its running example. (Hughes's paper is a well-known benchmark in the PBT literature [10, 13, 36, 38].) The paper's accompanying artifact [24] contains one correct BST implementation and eight erroneous ones. For example, one bug results in a singleton tree being returned during BST insertion, while another bug reverses key comparison when deleting a key-value pair from the tree.[2] We ported these implementations to OCaml as nine separate modules. Subsequently, we found that MICA was able to successfully detect divergent behavior between the correct and erroneous modules.

Specifically, we evaluate MICA by measuring the average number of tests required to provoke failure in each observational equivalence test. We measure this average by executing the PBT code derived by MICA for 1000 times, each time with a different random seed. Following a technique established in prior work [10], in all our tests, we generate keys uniformly at random from the range 0 to `size`, where `size` is the internal size parameter of MICA's QuickCheck generator. As Figure 3 shows, MICA was able to detect all bugs in Hughes's repository without any user intervention.

In addition to this case study, we are currently working on using MICA for differential testing of pairs of real-world OCaml libraries that implement the same signature, in particular libraries for Patricia trees [16, 28], sequences [5, 29], and ropes [17, 39]. We are also examining how compilation times using MICA's PPX extension scale with respect to module signature size.

5 RELATED WORK

MONOLITH [35] and ARTICHECK [3] are differential testing frameworks for ML modules that provide users with GADT-based DSLs to represent well-typed sequences of function calls. Using these DSLs, users declare functions to be tested across modules; both libraries use coverage-guided fuzzers to enumerate inhabitants of abstract data types during testing. Like these tools, MICA generates well-typed symbolic expressions, but it obviates the need for users to learn specialized DSLs, automatically producing specialized PBT code instead.

Model-based testing is a similar style of testing which examines whether the system under test is observationally equivalent to an abstract model. Model-based testing was pioneered in the PBT community by QUVIQ's Erlang QuickCheck library [1], which uses finite state machines as abstract models. This approach was brought to OCaml via the state-machine based PBT library QCSTM [30]. In QCSTM, symbolic expressions are represented as algebraic data types (ADTs), while the testing harness features state-dependent QuickCheck generators for symbolic expressions, along with functions that interpret expressions over both the model and target implementations. Our work builds on QCSTM by utilizing a similar ADT-based representation for symbolic expressions and adding support for testing binary operations over abstract types.

For testing ML modules more broadly, one can utilize GOSPEL [4], a specification language for ML modules, along with ORTAC [18], a runtime assertion-checking tool that checks GOSPEL specifications. Notably, ORTAC offers a plugin that supports QCHECK-STM [31], a variant of QCSTM

[2]We refer the reader to Hughes's paper [25] for detailed descriptions of all eight bugs.

adapted for testing parallel Multicore OCaml code. In this setup, in addition to generating random symbolic expressions, the test harness also checks whether pre- and post-conditions expressed using Gospel are satisfied in-between function calls [34].

6 FUTURE WORK

We plan to extend Mica to support OCaml functors and modules with multiple abstract types, and add the ability to generate a wider variety of higher-order functions. Furthermore, inspired by recent tools that combine coverage-guided fuzzing and PBT [11, 27], we plan on investigating whether coverage information could be used to tune Mica's generator of random `exprs` so that newly generated `exprs` tend to exercise previously untested code.

Finally, although the PBT code derived by Mica currently uses Jane Street's `Core.Quickcheck` library, Mica's design is library-agnostic. We leave it as future work to adapt Mica to support other OCaml PBT frameworks (e.g. QCheck [9]), building on recent work that uses the Etna PBT evaluation platform [38] to compare the efficacy of different OCaml PBT frameworks [26].

ACKNOWLEDGEMENTS

A prototype of Mica was presented at the ICFP '23 Student Research Competition; we thank the ICFP SRC program committee for their feedback. We also thank Jeremy Yallop, Carl Eastlund, David Vulakh, and Jan Midtgaard for comments and suggestions. Lastly, we thank the University of Pennsylvania's PLClub for support and encouragement. Development of Mica was supported by the National Science Foundation under grant NSF #2402449, *SHF: Medium: Usable Property-Based Testing*.

REFERENCES

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing Telecoms Software with QuviQ QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang* (Portland, Oregon, USA) *(ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. https://doi.org/10.1145/1159789.1159792

[2] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. Testing Polymorphic Properties. In *Proceedings of the 19th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer International Publishing, 125–144. https://doi.org/10.1007/978-3-642-11957-6_8

[3] Thomas Braibant, Jonathan Protzenko, and Gabriel Scherer. 2014. ArtiCheck: well-typed generic fuzzing for module interfaces. In *OCaml Workshop 2014*. http://gallium.inria.fr/~scherer/doc/articheck-long.pdf

[4] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. 2019. GOSPEL—Providing OCaml with a Formal Specification Language. In *Formal Methods – The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 484–501. https://doi.org/10.1007/978-3-030-30942-8_29

[5] Arthur Charguéraud, François Pottier, and Émilie Guermeur. 2020. Sek: An efficient OCaml implementation of ephemeral and persistent sequences. https://gitlab.inria.fr/fpottier/sek. GitLab repository.

[6] Koen Claessen. 2012. Shrinking and showing functions: (functional pearl). In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 73–80. https://doi.org/10.1145/2364506.2364516

[7] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[8] Michael R. Clarkson, Robert L. Constable, Nate Foster, Michael D. George, Dan Grossman, Daniel P. Huttenlocher, Dexter Kozen, Greg Morrisett, Andrew C. Myers, Radu Rugina, and Ramin Zabih. 2021. Functional Data Structures. In *OCaml Programming: Correct + Efficient + Beautiful*. Cornell University, Chapter 5.6. https://cs3110.github.io/textbook/chapters/modules/functional_data_structures.html#sets

[9] Simon Cruanes. 2013. QCheck: QuickCheck inspired property-based testing for OCaml. https://github.com/c-cube/qcheck.

[10] Joseph Cutler, Harrison Goldstein, Koen Claessen, John Hughes, and Benjamin C. Pierce. 2022. Functional Pearl: Holey Generators! https://www.seas.upenn.edu/~jwc/assets/holey.pdf. Unpublished manuscript.

[11] Stephen Dolan and Mindy Preston. 2017. Testing With Crowbar. https://github.com/stedolan/crowbar. Presented at the OCaml Workshop 2017.

[12] Carl Eastlund. 2015. Quickcheck for Core. Jane Street Tech Blog. https://blog.janestreet.com/quickcheck-for-core/

[13] Sólrún Halla Einarsdóttir, Nicholas Smallbone, and Moa Johansson. 2021. Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages* (Canterbury, United Kingdom) *(IFL '20)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/3462172.3462192

[14] Conal Elliott. 2021. Symbolic and automatic differentiation of languages. *Proc. ACM Program. Lang.* 5, ICFP, Article 78 (Aug 2021), 18 pages. https://doi.org/10.1145/3473583

[15] Jean-Christophe Filliâtre. 2009. Polynomials with coefficients in any ring. https://www.lri.fr/~filliatr/ftp/ocaml/ds/poly.ml.html.

[16] Jean-Christophe Filliâtre. 2020. Ptmap: Implementation of maps of integers with Patricia trees. https://github.com/backtracking/ptmap. GitHub repository.

[17] Jean-Christophe Filliâtre. 2020. Ropes: the rope data structure. https://github.com/backtracking/ocaml-bazaar/blob/main/rope.mli. GitHub repository.

[18] Jean-Christophe Filliâtre and Clément Pascutto. 2021. Ortac: Runtime Assertion Checking for OCaml (Tool Paper). In *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 244–253. https://doi.org/10.1007/978-3-030-88494-9_13

[19] Shayne Fletcher. 2017. Polynomials over rings. https://blog.shaynefletcher.org/2017/03/polynomials-over-rings.html.

[20] Harrison Goldstein. 2022. PL Theory Discussion: Regular Expressions, Derivatives, and DFAs. YouTube. https://www.youtube.com/watch?v=QaMU0wMMczU

[21] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[22] Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2023. Tyche: In Situ Analysis of Random Testing Effectiveness. In *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23 Adjunct)*. Association for Computing Machinery, New York, NY, USA, Article 96, 3 pages. https://doi.org/10.1145/3586182.3615788

[23] Kuen-Bang Hou (Favonia) and Zhuyang Wang. 2022. Logarithm and program testing. *Proc. ACM Program. Lang.* 6, POPL, Article 64 (jan 2022), 26 pages. https://doi.org/10.1145/3498726

[24] John Hughes. 2019. The code used in *How to Specify It*. https://github.com/rjmh/how-to-specify-it.

[25] John Hughes. 2020. How to Specify It!. In *Trends in Functional Programming*, William J. Bowman and Ronald Garcia (Eds.). Springer International Publishing, Cham, 58–83.

[26] Nikhil Kamath. 2024. Evaluating PBT Frameworks in OCaml. Poster presented at the PLDI '24 Student Research Competition. https://github.com/nikhil-kamath/etna

[27] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. https://doi.org/10.1145/3360607

[28] Dorian Lesbre and Matthieu Lemerre. 2024. Patricia Tree data structure in OCaml for maps and sets. https://github.com/codex-semantics-library/patricia-tree. GitHub repository.

[29] Jane Street Group LLC. 2022. Base: Standard library for OCaml. https://github.com/janestreet/base. GitHub repository.

[30] Jan Midtgaard. 2020. A Simple State-Machine Framework for Property-Based Testing in OCaml. In *OCaml Workshop*. https://janmidtgaard.dk/papers/Midtgaard%3AOCaml20.pdf

[31] Jan Midtgaard, Olivier Nicole, and Nicolas Osborne. 2022. Multicoretests - Parallel Testing Libraries for OCaml 5.0.

[32] Andre Nathan, Jeff Shaw, Markus Weissmann, and Florian Pichlmeier. 2015. `ocaml-stdint`: Various signed and unsigned integers for OCaml. https://github.com/andrenth/ocaml-stdint/tree/master. GitHub repository.

[33] Chris Okasaki. 1999. Red-black trees in a functional setting. *Journal of Functional Programming* 9, 4 (1999), 471–477. https://doi.org/10.1017/S0956796899003494

[34] Nicolas Osborne and Samuel Hym. 2023. Ortac/QCheck-STM. https://ocaml-gospel.github.io/ortac/ortac-qcheck-stm/index.html.

[35] François Pottier. 2021. Strong Automated Testing of OCaml Libraries. https://cambium.inria.fr/~fpottier/publis/pottier-monolith-2021.pdf. In *32es Journées Francophones des Langages Applicatifs (JFLA '21)* (Saint Médard d'Excideuil, France).

[36] Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. *Proc. ACM Program. Lang.* 7, PLDI, Article 178 (June 2023), 20 pages. https://doi.org/10.1145/3591292

[37] Nathan Rebours, Jeremie Dimino, Xavier Clerc, and Carl Eastlund. 2019. The future of OCaml PPX: towards a unified and more robust ecosystem. https://icfp19.sigplan.org/details/ocaml-2019-papers/8/The-future-of-OCaml-PPX-towards-a-unified-and-more-robust-ecosystem#the-future-of-ocaml-ppx-towards-a-unified-and-more-robust-ecosystem. Presentation delivered at the OCaml Workshop 2019.

[38] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 218 (aug 2023), 17 pages. https://doi.org/10.1145/3607860

[39] Chris Troestler. 2017. Ropes ("heavyweight strings") for OCaml. https://github.com/Chris00/ocaml-rope. GitHub repository.

[40] Jeremy Yallop. 2013. `ocaml-integers`: Various signed and unsigned integer types for OCaml. https://github.com/yallop/ocaml-integers. GitHub repository.

[41] Jeremy Yallop. 2022. Fast char sets for OCaml. https://github.com/yallop/ocaml-charset/. GitHub repository.