Mica: Automated Property-Based Testing for OCaml Modules

ERNEST NG, University of Pennsylvania, USA

1 PROBLEM AND MOTIVATION

"Modules matter most", as Harper [2011] once proclaimed. Indeed, OCaml's module system is indispensable for enforcing data abstractions and managing the complexity of large software systems. One key benefit of modularity is that a *signature*, or *interface*, can support multiple implementations, with the signature specifying the interface's behavior while hiding implementation details from clients. To check that two modules implementing the same signature behave identically, one can rely on *property-based testing* (PBT) tools such as QuickCheck [Claessen and Hughes 2000].

As a motivating example, consider an OCaml module signature `SetIntf` for a finite set data structure. Suppose there exist two modules `ListSet` and `BSTSet` that both implement this interface using lists and binary search trees (BSTs) respectively. A natural question arises: are these two modules *observationally equivalent*? That is, given equivalent commands, do the modules produce equivalent outputs? To test for observational equivalence, one can generate random sequences of symbolic commands and execute these commands over the two modules.

However, defining QuickCheck generators for this sequence of commands requires significant programmer effort, as the abstract type `'a t` is instantiated differently in the two modules. Moreover, due to encapsulation, clients of a module cannot access the module's internal implementation – they only have access to the interface.

To address this issue, we propose Mica, a tool that examines if two OCaml modules with the same signature are behaviorally equivalent by *automatically* producing PBT code specialized to the interface.

```
module type SetIntf = sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val intersect: 'a t -> 'a t -> 'a t
  val size: 'a t -> int
  val is_empty : 'a t -> bool
  val invariant: 'a t -> bool
  ...
end

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

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

module BSTSet : SetIntf = struct
  type 'a t = 'a tree
  (* BST invariant *)
  let invariant s = ...
  ...
end
```

2 BACKGROUND AND RELATED WORK

PBT was popularised by Haskell's QuickCheck library [Claessen and Hughes 2000], which allows users to test executable properties of programs on large numbers of randomly generated inputs. The notion of *model-based testing* was introduced to QuickCheck via an extension for testing monadic code [Claessen and Hughes 2002]. This testing framework examines whether the system under test is observationally equivalent to an abstract model when executing commands.

This approach was extended in QuviQ's Erlang QuickCheck library [Hughes 2016], which uses a finite state machine as the model. This framework was later adopted by the QCSTM OCaml state-machine testing framework [Midtgaard 2020], which is based on the QCheck PBT library [Cruanes 2017]. In QCSTM, symbolic commands are represented as algebraic data types (ADTs), while the testing harness features state-dependent command generators and functions that interpret commands over the implementation. Our work builds on QCSTM by utilising a similar ADT-based

Author's address: Ernest Ng, University of Pennsylvania, Philadelphia, Pennsylvania, USA, ngernest@seas.upenn.edu.
ACM member number: 4784536. Entry for ICFP'23 SRC, Graduate Category. Advised by Benjamin C. Pierce.

representation for symbolic commands, adding support for checking invariants and testing binary operations on abstract types (e.g. `intersect` in the finite set example from section 1).

Model_quickcheck [Dumond 2020] is a similar model-based PBT framework for testing imperative OCaml code, based on Jane Street's Base_quickcheck PBT library [Eastlund 2015]. Each command is represented as an individual module containing separate functions for executing the command over the model and the implementation. Our work furthers this approach by providing the capability to automatically derive functions for interpreting symbolic commands over the two modules.

Articheck [Braibant et al. 2014] is a randomized testing tool for OCaml module signatures. Articheck uses generalized algebraic data types (GADTs) to represent well-typed sequences of commands, and relies on fuzzing tools to enumerate inhabitants of abstract types when simulating function calls. Our work is inspired by this approach in that we also generate type safe sequences of function calls in the interface, but our tool completely automates this process.

Monolith [Pottier 2021] is a model-based testing framework that uses AFL-based coverage guidance to examine if two modules are observationally equivalent when executing the same sequence of commands. In the same vein as Articheck, Monolith provides users with a GADT-based DSL for declaring functions to be tested across both modules. Our work builds on Monolith's approach to testing two modules for observational equivalence. Moreover, our tool automatically produces the requisite PBT code and obviates the need for users to learn a specialised DSL.

3 APPROACH AND UNIQUENESS

We present Mica, a tool that parses an OCaml module signature containing abstract types, and *automatically* derives specialized PBT code which utilises Jane Street's Core.QuickCheck library [Madhavapeddy and Minsky 2022]. Suppose a developer takes the n-th version of a module (known to be correct) and optimizes the implementation, producing the $(n+1)$-th version. By feeding the n-th and $(n+1)$-th versions of the module to Mica, any behavioral discrepancies due to erroneous optimizations can be swiftly identified, with minimal programmer effort required. Additionally, by feeding two copies of the same module to Mica, one can check if a representation invariant specified in the signature is upheld.

Specifically, Mica generates definitions for three ADTs: `expr`, `ty`, `value`. Together, these types represent sequences of calls to the module's functions. The type `expr` represents *expressions* consisting of *symbolic commands*, whereas `ty` and `value` respectively denote the possible OCaml types and associated values that `exprs` can return. Mica also produces a functor `ExprToImpl` that takes as input a module `M` implementing the signature under test, and produces a test harness containing a function `interp` that interprets symbolic commands (`exprs`) over the module `M` and produces the corresponding `value`. Note that the type `value` is defined in the module that is returned by the `ExprToImpl` functor, as it depends on the abstract type `M.t`.

For example, for the finite set interface `SetIntf` presented in section 1, the automatically derived definitions for these datatypes and functions are presented on the following page. Notably, all of the code displayed in this section is *automatically* produced by Mica, without any programmer intervention.[1]

Note that each declaration in the signature `SetIntf` corresponds to a constructor with the same name in the `expr` ADT. Functions of arity 2, for example `add x s`, are represented using constructors whose arguments share the same type as the function arguments. The abstract type `'a t` in the module signature corresponds to the constructor `T` in the type `ty`, with values of this abstract type corresponding to the `ValT` constructor of the `value` ADT.

Our tool also produces the definition of a QuickCheck generator for `exprs` (`gen_expr`), whose argument `ty` denotes the return type of the generated command sequence. Successive calls to the

[1]Implementation: https://github.com/ngernest/module_pbt (repository contains the automatically generated PBT code)

```
type expr =
  | Empty
  | Add of int * expr
  | Intersect of expr * expr
  | Size of expr
  | Is_empty of expr
  | Invariant of expr
  ...

type ty = Bool | Int | T
```

```
module ExprToImpl (M : SetIntf) = struct
  type value = ValBool of bool | ValInt of int | ValT of int M.t

  let rec interp (expr : expr) : value =
    match expr with
    | Empty -> ValT (M.empty)
    | Add (x, e) -> match interp e with
      | ValT v -> ValT (M.add x v)
       ...
    | Intersect (e1, e2) -> match (interp e1, interp e2) with
      | (ValT e1', ValT e2') -> ValT (M.union e1' e2')
       ...
end
```

generator `gen_expr` produce sequences of symbolic commands that are type safe, e.g. `Intersect (Add 2 Empty) Empty`, but not `Is_empty (Size Empty)`. Moreover, since the return type of the command has to be specified as an argument in each recursive call to `gen_expr`, the generated sequences of function calls are guaranteed to be well-typed.

Lastly, our tool also produces an executable that invokes the aforementioned PBT code and tests two modules for observational equivalence. For example, the code to the right examines whether the list and BST implementations of the set interface from section 1 behave identically when executing randomly generated command sequences that return `int`.

```
let rec gen_expr (ty : ty) : expr Generator.t =
  let%bind k = QC.size in
  match ty, k with
  | (T, 0) -> return Empty
  | (T, _) ->
    let intersect =
      let%bind e1 =
        QC.with_size ~size:(k / 2) (gen_expr T)
      and e2 =
       QC.with_size ~size:(k / 2) (gen_expr T) in
      QC.return @@ Intersect (e1, e2) in
    ...
    QC.union [intersect; ...]
  | (Int, _) -> ...

(* Executable code *)
module M1 = ExprToImpl(ListSet)
module M2 = ExprToImpl(BSTSet)

QC.test (gen_expr Int) ~f:(fun e ->
  match (M1.interp e, M2.interp e) with
    | (ValInt n1, ValInt n2) ->
      [%test_eq: int] n1 n2
    ...)
```

4 RESULTS AND CONTRIBUTIONS

As a proof-of-concept, we are currently testing Mica with three different signatures that each have two different implementations, namely finite sets [Sergey 2021; Zdancewic and Weirich 2022], stacks [Alekseyev 2022; Clarkson et al. 2021] and polynomials [Filliâtre 2009; Fletcher 2017]. The executable that is automatically produced by Mica can detect when the two modules' behavior diverge due to a bug in one implementation.

For example, suppose one module implementing the `SetIntf` interface erroneously defines `is_empty Empty = false`, whereas the other module defines `is_empty Empty = true`. This discrepancy is identified when the executable is run, printing an error message containing the expression that evaluates differently. We are currently working on a empirical evaluation of Mica on different module signatures, and we hope to share our analysis in the future. Ultimately, we hope that through its capability to automatically derive PBT code for generic module signatures, Mica can make PBT more accessible to the OCaml community.

Future work. We intend to incorporate ideas from the existing PBT literature to tune our generator for `expr`s so that it can be maximally useful. Inspired by techniques from QuviQ QuickCheck [Hughes 2016] and others, we would want generated expressions to re-use previously generated data and preconditions for function calls to be enforced by construction in the generator. For instance, for the Set example above, we would like to generate `expr`s where we add and subsequently remove the *same* element. We believe that encoding dependencies between successive function calls in our `expr` generator is a promising idea that merits further research.

REFERENCES

Arseniy Alekseyev. 2022. Base: Standard library for OCaml - Stack Module. https://github.com/janestreet/base/blob/master/src/stack.ml. GitHub repository.

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

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

Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* 37 (06 2002). https://doi.org/10.1145/636517.636527

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

Simon Cruanes. 2017. QuickCheck Inspired Property-Based Testing for OCaml. https://github.com/c-cube/qcheck/.

Jesse Dumond. 2020. Model_quickcheck: Model-based testing for imperative OCaml code. https://github.com/suttonshire/model_quickcheck.

Carl Eastlund. 2015. Quickcheck for Core. https://blog.janestreet.com/quickcheck-for-core/.

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

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

Robert Harper. 2011. *Modules Matter Most.* https://existentialtype.wordpress.com/2011/04/16/modules-matter-most/

John Hughes. 2016. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Vol. 9600. 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

Anil Madhavapeddy and Aaron Minsky. 2022. Property Testing with QuickCheck. In *Real World OCaml* (2nd ed.). Cambridge University Press, Cambridge, United Kingdom, Chapter 18.3, 339–343.

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

François Pottier. 2021. Strong automated testing of OCaml libraries. In *Journées Francophones des Langages Applicatifs (JFLA)*.

Ilya Sergey. 2021. Representing Sets via Binary Search Trees. https://ilyasergey.net/YSC2229/week-11-bst.html.

Steve Zdancewic and Stephanie Weirich. 2022. Modularity and Abstraction: Finite Sets. CIS 1200 lecture notes, University of Pennsylvania. https://www.seas.upenn.edu/~cis120/23su/files/120notes.pdf Chapter 10, Section 10.1.