Property-Based Testing For OCaml through Coq

Paaras Bhandari
University of Maryland College Park
paaras99@umd.edu

Leonidas Lampropoulos
University of Maryland College Park
leonidas@umd.edu

***Abstract*—We will present a property-based testing framework for OCaml that leverages the power of QuickChick, a popular and mature testing plugin for the Coq proof assistant, by automatically constructing a extraction-based shim between OCaml and Coq. That gives OCaml programmers access to the advanced automation and fuzzing facilities that QuickChick provides.**

I. Introduction

Property-based random testing is a popular testing technique used to ensure that a program conforms to an executable specification. Popularized by Haskell's QuickCheck [9], property-based testing libraries have found their way to most languages, including OCaml with straightforward translations like QCheck [1] and ocaml-quickcheck[2], or more ambitious frameworks incorporating coverage-guided fuzzing like Crowbar [10].

In such frameworks, users write *properties* of the program under test, and the tool is in charge of generating a large number of random inputs, executing the property, and, if a counterexample is found, minimizing it to report it to the user. For example, a property that a list `reverse` function should satisfy is that it preserves the length of the input list:

```
let reverse_prop (l : int list) =
  length l = length (reverse l)
```

To test such a property, a property-based testing framework generates random lists of integers, executes `reverse_prop`, and checks if the result is `false`. If it is, it will try to *shrink*—i.e., minimize—the list (e.g. by dropping elements or by shrinking the elements themselves), before presenting a minimal counterexample to the user.

But how does a tool know these lists are to be generated and shrunk? In many cases (including QCheck, ocaml-quickcheck, and Crowbar), the user has to write programs that perform this generation and minimization, usually relying on a comprehensive library of expressive combinators. However, this process is tedious, error-prone, and hinders adoption of property-based testing. To that end, a lot of work has been to semi- or fully- automate various aspects of property-based testing: in Haskell's QuickCheck [3] and Coq's QuickChick[8], typeclasses are used to automatically dispatch the appropriate generators and meta-programming is used to automatically derive such generators for user-defined datatypes. Moreover, researchers have added support for advanced features in such frameworks, from generating *constrained* inputs (e.g. sorted lists, red-black trees, well-typed lambda terms), to AFL-style coverage-guided fuzzing [5, 6, 7].

Instead of developing or extending an OCaml tool for property-based testing by duplicating years of research and engineering effort, in this work we show how we can leverage prior work on QuickChick by building a bridge between OCaml and Coq through extraction.

II. Property-Based Testing in OCaml, through Coq

Coq is a proof assistant whose functional language, Gallina, is very similar to OCaml. Programs in Gallina can be run via extraction to OCaml, which can be highjacked to allow for hybrid Gallina/OCaml programs. We first show how this can be achieved manually, and then discuss our work on automating it.

Linking OCaml and Coq

Consider a polymorphic version of the `reverse_prop` property of the previous section, implemented in a `List` module:

```
let reverse_prop (l : 'a list) =
  length l = length (reverse l)
```

A QuickChick user can test this OCaml property by simply adding the definition as an axiom in Coq and ensuring it extracts to its OCaml counterpart:

```
Axiom test_reverse_prop :
  forall {A}, list A -> bool.

Extract Constant test_reverse_prop =>
 "List.test_reverse_prop".

QuickChick test_reverse_prop.
```

QuickChick would use typeclass resolution to figure out that it needs to generate lists (and defaults to lists of integers for polymorphic properties), extract the code necessary to test `test_reverse_prop`, and when the property itself needs to be executed, call the specified OCaml function. To ensure that this function can be found, a user can specify either a concrete path:

```
QCInclude "list.ml".
```

or an opam library to link at compile time:

```
QCOpen "list".
```

Handling User-Defined Types

In addition to typeclass resolution, QuickChick users enjoy a powerful derivation mechanism that provides generators, printers, and shrinkers for free. For example, in order to test a property of a QCheck user would have to roll their own generator:

```
type tree =
  | Leaf of int
  | Node of tree * tree

let leaf x = Leaf x
let node x y = Node (x,y)

let tree_gen = QCheck.Gen.(sized @@ fix
 (fun self n -> match n with
  | 0 -> map leaf nat
  | n ->
   frequency
    [1, map leaf nat;
     2, map2 node (self (n/2)) (self (n/2))
    ]))
```

The particular details of this generator are not important—it's a simple recursive generator that picks between generating a `Leaf` or a `Node` to generate a tree of up to a given size. What is important is that a user has to write it.

On the other hand, QuickChick provides such generators for free through OCaml meta-programming:

```
Inductive tree :=
| Leaf : int -> tree
| Node : tree -> tree -> tree.

Derive (Arbitrary, Show) for tree.
```

The snippet above does not only create a generator similar in behavior to the QCheck one, but also a printer and a minimizer.

So how can an OCaml user take advantage of this automation support? Well, the user only needs to extract the Coq definition of `tree` to the OCaml one! Assuming that the OCaml implementation lies in a `Tree` module, this is as simple as:

```
Extract Inductive tree =>
  "Tree.tree"
  ["Tree.Leaf" "Tree.Node"].
```

Automatically Creating the Extraction Shim

Naturally, if we required an OCaml user to perform all of this manually, we'd just be exchanging one boilerplate for another. To that end, we automated this process to provide a seamless experience to OCaml users.

A user need only specify the name of the property under test and its location in their OCaml code:

```
let main =
  quickChick "list" reverse_prop
```

This will generate a Coq file with the appropriate boilerplate, compile it, extract the relevant property to a temporary OCaml file, and execute it. To generate the boilerplate, we recursively check (using regular expressions) which user-defined types are involved in the property and convert them to Coq type definitions using coq-of-ocaml [4]. These Coq type definitions are then used to define the types in the Coq script. Finally, we `Derive` the generator, printer, and minimizer before declaring the property-test as an axiom.

III. Presentation

The proposed presentation will be an OCaml demo of the tool, covering its base workflow, and showcasing the features of QuickChick that it grants OCaml users access to.

References

[1] *c-cube/qcheck: QuickCheck inspired property-based testing for OCaml.* https://github.com/c-cube/qcheck.

[2] *camlunity/ocaml-quickcheck: OCaml QuickCheck*. https://github.com/camlunity/ocaml-quickcheck.

[3] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2000, pp. 268–279. URL: http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf.

[4] Clarus. *clarus/coq-of-ocaml*. URL: https://github.com/clarus/coq-of-ocaml.

[5] Leonidas Lampropoulos. "Random Testing for Language Design". PhD thesis. University of Pennsylvania, 2018.

[6] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. "Coverage Guided, Property Based Testing". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. Oct. 2019.

[7] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. "Generating good generators for inductive relations". In: *PACMPL* 2.POPL (2018), 45:1–45:30. DOI: 10.1145/3158133. URL: http://doi.acm.org/10.1145/3158133.

[8] Zoe Paraskevopoulou et al. "Foundational Property-Based Testing". In: *6th International Conference on Interactive Theorem Proving (ITP)*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 325–343. ISBN: 978-3-319-22101-4. URL: http://prosecco.gforge.inria.fr/personal/hritcu/publications/foundational-pbt.pdf.

[9] *QuickCheck: Automatic testing of Haskell programs*. https://hackage.haskell.org/package/QuickCheck.

[10] *stedolan/crowbar: Property fuzzing for OCaml*. https://github.com/stedolan/crowbar.