# Multicoretests – Parallel Testing Libraries for OCaml 5.0

Jan Midtgaard

Olivier Nicole

Nicolas Osborne

**Tarides** 

#### 1 Introduction

Parallel and concurrent code is notoriously hard to test because of the involved non-determinism, yet it is facing OCaml programmers with the coming OCaml 5.0 multicore release. We present two related testing libraries to improve upon the situation:

- Lin a library to test for linearizability
- STM a state-machine testing library

Both libraries build on QCheck [18], a black-box, property-based testing library in the style of QuickCheck [5]. The two libraries represent different trade-offs between required user effort and provided guarantees and thereby supplement each other

In this document we will use OCaml's Hashtbl module as a running example.

## 2 The Lin library

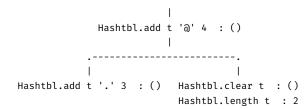
The Lin library performs a sequence of random operations in parallel, records the results, and checks whether the observed results are linearizable by reconciling them with a sequential execution. The library offers an embedded, combinator DSL to describe signatures succinctly. As an example, the required specification to test (parts of) the Hashtbl module is given in fig. 1.

The first line indicates the type of the system under test (SUT). In the above case we intend to test Hashtbls with char keys and int values. The bindings init and cleanup allow for setting up and tearing down the SUT. The api then contains a list of type signature descriptions using combinators in the style of Ctypes [23]. Different combinators unit, bool, int, list, option, returning, returning\_or\_exc, ... allow for a concise type signature description.

From the signature description the Lin library will iterate a number of test instances. Each test instance consists of a "sequential prefix" of calls to the specified operations, followed by a spawn of two parallel Domains that each call a sequence of operations.

For each test instance Lin chooses the individual operations arbitrarily and records the result received from each operation. The framework will then perform a search for a sequential interleaving of the same calls, and succeed if it finds one. Since Hashtbls are not safe for parallelism, the output produces the following:

Results incompatible with sequential execution



This describes that in one parallel execution, Lin received the response 2 from Hashtbl.length, despite having just executed Hashtbl.clear. It this case, it is not possible to interleave Hashtbl.add t '.' 3 with these two calls to explain this observed behaviour.

Underneath the hood, Lin does its best to schedule the two parallel Domains on top of each other. It also repeats each test instance, to increase the chance of triggering an error, and it fails if just one of the repetitions fail to find a sequential interleaving. Finally, upon finding an error it reduces the involved operation sequences to a local minimum, which is what is printed above.

Lin is phrased as an OCaml functor, Lin\_api.Make. The module resulting from Lin\_api.Make(HashtblSig) contains a binding lin\_test that can perform the above linearizability test over Domains, the basic unit of parallelism coming in OCaml 5.0. An alternative Lin mode works over Thread for testing concurrent but non-overlapping executions. This mode thus mimicks the above functionality by replacing Domain.spawn and Domain.join with Thread.create and Thread.join, respectively.

### 3 The STM library

Like Lin the STM library also performs a sequence of random operations in parallel and records the results. In contrast to Lin, STM then checks whether the observed results are linearizable by reconciling them with a sequential execution of a model description.

The model expresses the intended meaning of each tested operation. As such, the required STM user input is longer compared to that of Lin. The corresponding code to describe a Hashtbl test using STM is given in fig. 2.

Again this requires a description of the system under test, sut. In addition STM requires a type cmd for describing the tested operations. The hooks init\_sut and cleanup match init and cleanup from Lin, respectively.

```
module HashtblSig =
struct
  type t = (char, int) Hashtbl.t
  let init () = Hashtbl.create ~random:false 42
  let cleanup _ = ()
  open Lin api
  let a,b = char_printable,nat_small
  let api =
    [ val_ "Hashtbl.clear"
                              Hashtbl.clear
                                               (t a-> returning unit);
      val "Hashtbl.add"
                                               (t a-> a a-> b a-> returning unit);
                              Hashtbl.add
      val "Hashtbl.remove"
                              Hashtbl.remove
                                               (t @-> a @-> returning unit);
                                               (t a-> a a-> returning or exc b);
      val_ "Hashtbl.find"
                              Hashtbl.find
      val_ "Hashtbl.replace"
                              Hashtbl.replace (t @-> a @-> b @-> returning unit);
      val "Hashtbl.mem"
                              Hashtbl.mem
                                                (t a-> a a-> returning bool);
      val_ "Hashtbl.length"
                                               (t a-> returning int); ]
                              Hashtbl.length
end
```

Figure 1: Specification of selected Hashtbl functions for testing using Lin.

A distinguishing feature is type state = (char  $\star$  int) list describing with a pure association list the internal state of a hashtable. next\_state is a simple state transition function describing how the state changes across each cmd. For example, Add (k,v) appends the key-value pair onto the association list.

arb\_cmd is a generator of cmds, taking state as a parameter. This allows for state-dependent cmd generation, which we use to increase the chance of producing a Remove 'c', Find 'c', ... following an Add 'c'. Internally arb\_cmd uses combinators Gen.return, Gen.map, and Gen.map2 from QCheck to generate one of 7 different operations. For example, Gen.map (fun k -> Mem k) char creates a Mem command with the result obtained from the char generator. arb\_cmd further uses a derived printer show\_cmd to be able to print counterexamples.

run executes the tested cmd over the SUT and wraps the result up in a result type res offered by STM. Combinators unit, bool, int, ... allow to annotate the result with the expected type. postcond then expresses a post-condition by matching the received res, for a given cmd with the corresponding answer from the model description. For example, this compares the Boolean result r from Hashtbl.mem with the result from List.mem\_assoc. Similarly precond expresses a cmd pre-condition.

STM is also phrased as an OCaml functor. The module resulting from STM.Make(HashtblModel) thus includes a binding agree\_test for running sequential tests comparing the SUT behaviour to the given model. Another binding agree\_test\_par instead runs parallel tests that make a similar comparison over a sequential prefix and two parallel Domains, this time also searching for a sequential interleaving of cmds. For example, one execution of agree\_test\_par produced the following output. Note how no interleaving of Remove from the first parallel cmd sequence can make the association list model return -1 from Length:

Results incompatible with linearized model

#### 4 Status

Both libraries are open source and available for download on GitHub from https://github.com/jmid/multicoretests. As the APIs are still unstable and under development, we have not made a public release yet. Interested users can nevertheless easily install the libraries with opam.

During development we have used examples such as Hashtbl to confirm that the approach indeed works as intended. The behaviour is continuously confirmed by running GitHub Actions of the latest trunk compiler. As further testament to the usability of the approach, we have used the libraries to test parts of OCaml's Stdlib, as well as the Domainslib and lockfree libraries. In doing so, we have been able to find and report a number of issues which have either already been fixed or have fixes underway:

- In\_channel and Out\_channel unsafety [1, 3]
- · MacOSX crash [21]
- Buffer unsafety [22, 15]

#### 5 Related Work

QuickCheck [5] originally introduced property-based testing within functional programming with combinator-based generators, properties, and test-case reduction. It has since been ported to over 30 other programming languages, including Quviq QuickCheck [19]—a commercial port to Erlang.

```
module HashtblModel =
                                                           let next state (c:cmd) (s:state) = match c with
                                                             | Clear
                                                                             -> []
struct
 type sut = (char, int) Hashtbl.t
                                                             | Add (k,v)
                                                                             -> (k,v)::s
 type state = (char * int) list
                                                             | Remove k
                                                                             -> List.remove_assoc k s
 type cmd =
                                                             | Find
                                                                             -> s
   | Clear
                                                             | Replace (k,v) -> (k,v)::(List.remove_assoc k s)
                                                             | Mem _
   | Add of char * int
   | Remove of char
                                                             | Length
                                                                             -> s
    | Find of char
   | Replace of char * int
                                                           let run (c:cmd) (h:sut) = match c with
   | Mem of char
                                                             | Clear
                                                                             -> Res (unit, Hashtbl.clear h)
                                                                             -> Res (unit, Hashtbl.add h k v)
    | Length [@@deriving show { with_path = false }]
                                                             | Add (k,v)
                                                                             -> Res (unit, Hashtbl.remove h k)
                                                             l Remove k
 let init_sut () = Hashtbl.create ~random:false 42
                                                             | Find k
                                                                             -> Res (result int exn,
 let cleanup (_:sut) = ()
                                                                                      protect (Hashtbl.find h) k)
                                                             | Replace (k,v) -> Res (unit, Hashtbl.replace h k v)
 let arb_cmd (s:state) =
                                                             l Mem k
                                                                             -> Res (bool, Hashtbl.mem h k)
    let char =
                                                             | Length
                                                                             -> Res (int, Hashtbl.length h)
      if s = []
      then Gen.printable
                                                           let init_state = []
      else Gen.(oneof [oneofl (List.map fst s);
                       printable]) in
                                                           let precond (_:cmd) (_:state) = true
                                                           let postcond (c:cmd) (s:state) (res:res) =
   let int = Gen.nat in
   QCheck.make ~print:show cmd
                                                             match c,res with
                                                                              Res ((Unit,_),_)
     (Gen.oneof
                                                             | Clear,
                                                             | Add (_,_),
                                                                              Res ((Unit,_),_)
       [Gen.return Clear;
        Gen.map2 (fun k v \rightarrow Add (k,v)) char int;
                                                                              Res ((Unit,_),_) -> true
                                                             Remove _,
        Gen.map (fun k -> Remove k) char;
                                                             | Find k,
                                                                              Res ((Result (Int,Exn),_),r) ->
        Gen.map (fun k
                         -> Find k) char;
                                                                 r = (try 0k (List.assoc k s)
        Gen.map2 (fun k v -> Replace (k,v)) char int;
                                                                      with Not_found -> Error Not_found)
        Gen.map (fun k
                         -> Mem k) char;
                                                             | Replace (_,_), Res ((Unit,_),_) -> true
        Gen.return Length;
                                                             | Mem k,
                                                                              Res ((Bool,_),r) \rightarrow r = List.mem_assoc k s
       ])
                                                             | Length,
                                                                              Res ((Int,_),r) \rightarrow r = List.length s
                                                             | _ -> false
```

Figure 2: Description of a Hashtbl test using STM.

Model-based testing was initially suggested as a method for testing monadic code with Haskell's QuickCheck [6]. An explicit framework was later proposed in the GAST property-based testing library for Clean [10]. The commercial Quviq QuickCheck [19] was later extended with a state-machine model framework for testing stateful systems [2]. This approach was extended further to test parallel code for data races [7]. This general approach for parallel testing has since been adopted in other ports, such as Erlang's open source Proper [14], Haskell Hedgehog [9], ScalaCheck [20], and Kotlin's propCheck [17]. STM continues this adoption tradition. qcstm [11] is a previous OCaml adoption, also building on QCheck. It was missing the ability to perform parallel testing though. STM seeks to remedy this limitation.

Crowbar [8] is another QuickCheck-style testing framework with combinator-based generators. In contrast to QuickCheck, it utilizes AFL-based coverage guidance to effectively guide the generated input towards unvisited parts of the SUT. Crowbar does not come with a state-machine framework. Monolith [16] is a model-based testing framework also building on AFL-based coverage guidance. In contrast to STM, Monolith's models are oracle implementations

with operations matching the type signatures of the tested operations. Neither Crowbar nor Monolith come with skeletons to perform parallel or concurrent testing. Furthermore the AFL-based coverage-guidance underlying both Crowbar and Monolith works best for deterministic, sequential code.

ParaFuzz [13] is another approach to fuzz test multicore OCaml programs. It simulates parallelism in OCaml through concurrency, enabling scheduling order to be controlled by AFL, which helps to trigger and find scheduling-dependent bugs. A caveat is that ParaFuzz assumes data race freedom.

Ortac can extract Monolith-based tests from a formal specification written in Gospel, a specification language for OCaml [12]. Gospel specifications include models, preconditions, and post-conditions close to those of STM. The extracted tests however inherit Monolith's and AFL's focus on sequential code.

ArtiCheck [4] tests random combinations of OCaml calls from type signature descriptions, similarly to Lin. Whereas Lin and STM target impure interfaces, ArtiCheck targets persistent (pure) interfaces. ArtiCheck furthermore targets sequential rather than parallel or concurrent tests.

#### 6 Conclusion

We have presented two libraries, Lin and STM for testing parallel and concurrent code for OCaml 5.0. Despite still being under development, we believe both libraries could be helpful to developers of OCaml 5.0 programs.

### References

- [1] Add (Failing) {In,Out}\_channel Linearization Tests. URL: https://github.com/jmid/multicoretests/pull/
- [2] Thomas Arts et al. "Testing Telecoms Software with Quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (Erlang 2006).* 2006, pp. 2–10.
- [3] Audit Stdlib for Mutable State (Comment). URL: https: //github.com/ocaml/ocaml/issues/10960 # issuecomment-1087660763.
- [4] Thomas Braibant, Jonathan Protzenko, and Gabriel Scherer. "Well-Typed Smart Fuzzing". In: ML Familiy Workshop (ML 2014). 2014. URL: https://github.com/braibant/articheck/blob/master/ml-workshop-talk/talk.pdf.
- [5] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000). 2000, pp. 268–279.
- [6] Koen Claessen and John Hughes. "Testing Monadic Code with QuickCheck". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell 2002)*. 2002, pp. 65–77.
- [7] Koen Claessen et al. "Finding Race Conditions in Erlang with QuickCheck and PULSE". In: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009). 2009, p. 12.
- [8] Stephen Dolan and Mindy Preston. "Testing with Crowbar". In: OCaml Users and Developers Workshop. 2017.
- [9] Hedgehog. URL: https://github.com/hedgehogqa/ haskell-hedgehog.
- [10] Pieter W. M. Koopman and Rinus Plasmeijer. "Testing Reactive Systems with GAST". In: Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming (TFP 2003). Vol. 4. Trends in Functional Programming. 2003, pp. 111–129.
- [11] Jan Midtgaard. "A Simple State-Machine Framework for Property-Based Testing in OCaml". In: *OCaml Users and Developers Workshop*. 2020.

- [12] Nicolas Osborne and Clément Pascutto. "Leveraging Formal Specifications to Generate Fuzzing Suites". In: OCaml Users and Developers Workshop. 2021. URL: https://hal.inria.fr/hal-03328646.
- [13] Sumit Padhiyar, Adharsh Kamath, and KC Sivaramakrishnan. "Parafuzz: Coverage-guided Property Fuzzing for Multicore OCaml Programs". In: OCaml Users and Developers Workshop. 2021.
- [14] Manolis Papadakis and Konstantinos Sagonas. "A PropEr Integration of Types and Function Specifications with Property-Based Testing". In: *Proceedings of* the 2011 ACM SIGPLAN Erlang Workshop. 2011, pp. 39– 50.
- [15] Parallel Access to Buffer Can Trigger Segfaults. URL: https://github.com/ocaml/ocaml/issues/11279.
- [16] François Pottier. "Strong Automated Testing of OCaml Libraries". In: Journées Francophones Des Langages Applicatifs (JFLA 2021). Feb. 2021.
- [17] propCheck. URL: https://github.com/1Jajen1/ propCheck.
- [18] *QCheck*. URL: https://github.com/c-cube/qcheck.
- [19] Quviq QuickCheck. URL: http://quviq.com/documentation/eqc/index.html.
- [20] ScalaCheck. URL: https://github.com/typelevel/ scalacheck.
- [21] Segfault on MacOSX with Trunk. URL: https://github.com/ocaml/ocaml/issues/11226.
- [22] STM Clean-Up. URL: https://github.com/jmid/ multicoretests/pull/63.
- [23] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. "A modular foreign function interface". In: Science of Computer Programming 164 (2018), pp. 82-97. URL: https://www.sciencedirect.com/science/ article/pii/S0167642317300709.