osnap! Painless and massive regression test generation for OCaml

Valentin Chaboche

Nomadic Labs, Paris, France valentin.chaboche@lambda-coins.com

29 july 2022

Bill is hired by a company to work on the blockchain `Taizosse`.

Bill is hired by a company to work on the blockchain `Taizosse`.

- `Taizosse` is a 20-year-old blockchain written in `OCaml` with little trafic on the network. But, suddenly, following the arrival of *CryptoCamel NFTs*, the number of users explodes putting the overall performance of the blockchain under high-stress.

Bill is hired by a company to work on the blockchain `Taizosse`.

- `Taizosse` is a 20-year-old blockchain written in `OCaml` with little trafic on the network. But, suddenly, following the arrival of *CryptoCamel NFTs*, the number of users explodes putting the overall performance of the blockchain under high-stress.
- Bill arrives as the saviour of the camel worshippers: he has to optimise the software to allow more camel worshippers to exchange their non-fungible tokens.

Bill is hired by a company to work on the blockchain `Taizosse`.

- `Taizosse` is a 20-year-old blockchain written in `OCaml` with little trafic on the network. But, suddenly, following the arrival of *CryptoCamel NFTs*, the number of users explodes putting the overall performance of the blockchain under high-stress.
- Bill arrives as the saviour of the camel worshippers: he has to optimise the software to allow more camel worshippers to exchange their non-fungible tokens.
- The problem is that if Bill makes a mistake, the blockchain collapses.

Bill then discovers the code written by his predecessors and is in charge of optimising it.

Bill then discovers the code written by his predecessors and is in charge of optimising it.
He manages to isolate a certain function `sum`:

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

(** /!\ Do not modify /!\ *)
let rec sum = function
  | Leaf x -> x
  | Node (a, b) -> sum a + sum b
```

He then decided to optimise the function using his extensive knowledge of OCaml:

```
let sum tree =
  let rec sum tree cont = match tree with
    | Leaf x -> cont x
    | Node (a, b) ->
      sum a (fun sum_a ->
        sum b (fun sum_a ->
          cont (sum_a + sum_a)))
  in sum tree (fun x -> x)
```

In order for the code to be accepted, he is nevertheless asked to write tests.

In order for the code to be accepted, he is nevertheless asked to write tests.

```
let test_sum =
  let tree1 = Node (Leaf 0, Leaf 0) in
  assert (sum tree1 = 0);
  let tree2 = Node (Leaf 1, Leaf 1) in
  assert (sum tree2 = 2);
  (* By induction other cases will work. *)
  ()
```

In order for the code to be accepted, he is nevertheless asked to write tests.

```
let test_sum =
  let tree1 = Node (Leaf 0, Leaf 0) in
  assert (sum tree1 = 0);
  let tree2 = Node (Leaf 1, Leaf 1) in
  assert (sum tree2 = 2);
  (* By induction other cases will work. *)
  ()
```

And the code is added in production.

However, Bill's modification is obviously wrong, and Bill will not pass his trial period. But how could he have done better?

- Write more unit tests: would he have found the inputs needed to detect the error case?
- Use random input generators to write property-based tests: which properties to test?

Why didn't he use osnap?

Why didn't he use osnap?

The objective with osnap is to use random scenario generation to produce regression tests.

Why didn't he use osnap?

The objective with osnap is to use random scenario generation to produce regression tests.

- Get inspiration from the property-based testing to randomly generate k scenarios.

Why didn't he use osnap?

The objective with osnap is to use random scenario generation to produce regression tests.

- Get inspiration from the property-based testing to randomly generate k scenarios.
- Store the results of these scenarios to create regression tests.

Why didn't he use osnap?

The objective with osnap is to use random scenario generation to produce regression tests.

- Get inspiration from the property-based testing to randomly generate k scenarios.
- Store the results of these scenarios to create regression tests.
- Re-run the test cases on new versions of the function to detect unwanted changes in results.

In order to generate `k` scenarios, we need to be able to generate a value for each parameter of a function.

```
type 'a spec = {
  gen : 'a gen;
    (** Generate random values for ['a]. *)
  printer : 'a printer option
    (** Optional printer to observe generated values. *)
}
```

Then, we consider the textual representation of the result to display it to the user.

```
type 'a result = 'a printer
```

Finally, we build the global specification from the parameter specs.

```
type ('fn, 'r) t =
  | Arrow : 'a spec * ('fn, 'r) t -> ('a -> 'fn, 'r) t
  | Result : 'r result -> ('r, 'r) t
```

Finally, we build the global specification from the parameter specs.

```
type ('fn, 'r) t =
  | Arrow : 'a spec * ('fn, 'r) t -> ('a -> 'fn, 'r) t
  | Result : 'r result -> ('r, 'r) t

let repeat n c = String.make n c

let spec_repeat : (int -> char -> string, string) t =
  Spec.(int ^> char ^>> Printer.string)
```

Finally, we build the global specification from the parameter specs.

```
type ('fn, 'r) t =
  | Arrow : 'a spec * ('fn, 'r) t -> ('a -> 'fn, 'r) t
  | Result : 'r result -> ('r, 'r) t

let repeat n c = String.make n c

let spec_repeat : (int -> char -> string, string) t =
  Spec.(int ^> char ^>> Printer.string)

val ( ^> ) : 'a spec -> ('b, 'c) t -> ('a -> 'b, 'c) t
```

Finally, we build the global specification from the parameter specs.

```
type ('fn, 'r) t =
  | Arrow : 'a spec * ('fn, 'r) t -> ('a -> 'fn, 'r) t
  | Result : 'r result -> ('r, 'r) t

let repeat n c = String.make n c

let spec_repeat : (int -> char -> string, string) t =
  Spec.(int ^> char ^>> Printer.string)

val ( ^> ) : 'a spec -> ('b, 'c) t -> ('a -> 'b, 'c) t
val ( ^>> ) : 'a spec -> 'b result -> ('a -> 'b, 'b) t
```

We can then define a specification for the sum function.

```
let gen_tree = ... and printer_tree = ...

let spec_tree = Spec.{ gen = gen_tree; printer = Some printer_tree}

let spec_sum : (tree -> int, int) Spec.t =
  Spec.(spec_tree ^>> Printer.int)
```

We can then define a specification for the sum function.

```
let gen_tree = ... and printer_tree = ...

let spec_tree = Spec.{ gen = gen_tree; printer = Some printer_tree}

let spec_sum : (tree -> int, int) Spec.t =
  Spec.(spec_tree ^>> Printer.int)
```

Then, we can generate the regression tests.

```
let _ =
  let test = Test.make ~spec:spec_sum sum in
  Runner.(run_tests ~encoding:Marshal [ test ])
```

```
+ {
+  name = sum;
+  scenarios = [
+    N (N (N (N (N (L 2, L 87), N (L 4000, L 7)), L 6), L 0), L 63)    =  4165
+    N (L 4, N (L 217, L 97))    =    318
+    N (L 6, L 505)    =    511
+    N (L 2, L 8)    =    10
+    N (L 80, N (L 7, L 69))    =    156
+    L 83    =    83
+    L 4    =    4
+    N (N (N (L 2, L 674), N (L 42, L 456)), L 90)    =    1264
+    L 7    =    7
+    L 504    =    504
+  ]
+ }

Do you want to promote this new snapshot? [Y\n]
```

```
+ {
+  name = sum;
+  scenarios = [
+    N (N (N (N (N (L 2, L 87), N (L 4000, L 7)), L 6), L 0), L 63)    =  4165
+    N (L 4, N (L 217, L 97))    =    318
+    N (L 6, L 505)    =    511
+    N (L 2, L 8)    =    10
+    N (L 80, N (L 7, L 69))    =    156
+    L 83    =    83
+    L 4    =    4
+    N (N (N (L 2, L 674), N (L 42, L 456)), L 90)    =    1264
+    L 7    =    7
+    L 504    =    504
+  ]
+ }
```

```
Do you want to promote this new snapshot? [Y\n]
```

We now have the regression tests.

Let's now come back to Bill's optimisation.

```
let sum tree =
  let rec sum tree cont = match tree with
    | Leaf x -> cont x
    | Node (a, b) ->
      sum a (fun sum_a ->
        sum b (fun sum_a ->
          cont (sum_a + sum_a)))
  in sum tree (fun x -> x)
```

Let's now re-run the regression tests.

```
--- failure -------------------------------------------------------------

@@ -1,14 +1,14 @@
 {
   name = sum;
   scenarios = [
-    N (N (N (N (N (L 2, L 87), N (L 4000, L 7)), L 6), L 0), L 63)    =    4165
-    N (L 4, N (L 217, L 97))    =    318
-    N (L 6, L 505)    =    511
-    N (L 2, L 8)    =    10
-    N (L 80, N (L 7, L 69))    =    156
+    N (N (N (N (N (L 2, L 87), N (L 4000, L 7)), L 6), L 0), L 63)    =    126
+    N (L 4, N (L 217, L 97))    =    388
+    N (L 6, L 505)    =    1010
+    N (L 2, L 8)    =    16
+    N (L 80, N (L 7, L 69))    =    276
     L 83    =    83
     L 4    =    4
-    N (N (N (L 2, L 674), N (L 42, L 456)), L 90)    =    1264
+    N (N (N (L 2, L 674), N (L 42, L 456)), L 90)    =    180
     L 7    =    7
     L 504    =    504
   ]


-------------------------------------------------------------------------------
failure: ran 1 test (1 error(s))
```

Now, Bill can use the regression tests of the sum function until he converges to a correction of his optimisation.

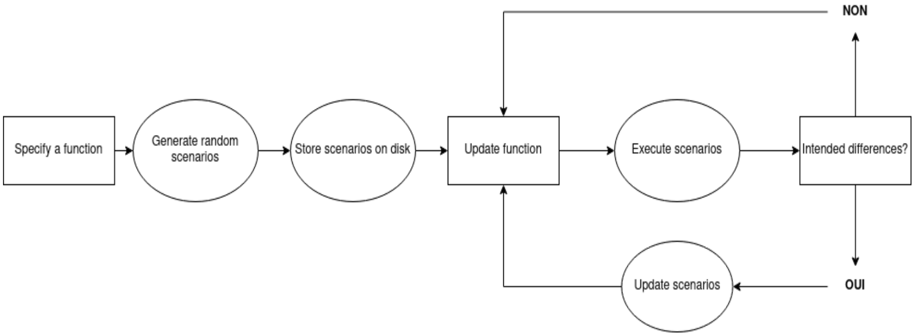
```
@@ -12,6 +12,6 @@ let sum tree =
     | Leaf x -> cont x
     | Node (a, b) ->
       sum a (fun sum_a ->
-          sum b (fun sum_a ->
-              cont (sum_a + sum_a)))
+          sum b (fun sum_b ->
+              cont (sum_a + sum_b)))
   in sum tree (fun x -> x)
```

Now, Bill can use the regression tests of the `sum` function until he converges to a correction of his optimisation.

```
@@ -12,6 +12,6 @@ let sum tree =
     | Leaf x -> cont x
     | Node (a, b) ->
       sum a (fun sum_a ->
-          sum b (fun sum_a ->
-              cont (sum_a + sum_a))
+          sum b (fun sum_b ->
+              cont (sum_a + sum_b)))
   in sum tree (fun x -> x)
```

The tests will now prove him right.

```
---------------------------------------------------------------
success: ran 1 test (1 passed)
```


NON
Specify a function
Generate random scenarios
Store scenarios on disk
Update function
Execute scenarios
Intended differences?
Update scenarios
OUI

Conclusion

Inspired by unit tests and property-based testing, we were able to:

Conclusion

Inspired by unit tests and property-based testing, we were able to:

- Automatically generate `k` scenarios using random generators

Conclusion

Inspired by unit tests and property-based testing, we were able to:

- Automatically generate `k` scenarios using random generators
- Abstract the expertise needed to extract properties from the code.

Conclusion

Inspired by unit tests and property-based testing, we were able to:

- Automatically generate `k` scenarios using random generators
- Abstract the expertise needed to extract properties from the code.
- Save and version the state of a function to prevent unwanted changes in the future.

Future works

We have several ways to improve the tool.

Future works

We have several ways to improve the tool.

- Better integration with the development environment, notably by connecting the tool to existing framework such as QCheck or ppx_expect.

Future works

We have several ways to improve the tool.

- Better integration with the development environment, notably by connecting the tool to existing framework such as `QCheck` or `ppx_expect`.
- Regression tests are not very resilient to change: old scenarios cannot be re-applied to a function if its signature changes.

Future works

We have several ways to improve the tool.

- Better integration with the development environment, notably by connecting the tool to existing framework such as `QCheck` or `ppx_expect`.
- Regression tests are not very resilient to change: old scenarios cannot be re-applied to a function if its signature changes.
- Generating a large number of scenarios does not ensure a large coverage of generators. We would then like to incrementally improve the code's coverage of the scenarios, for example, by using coverage tools such as `bisect_ppx`.

Thanks for listening!

https://github.com/vch9/osnap

Références

- Code coverage for OCaml and ReScript.
  https://github.com/aantron/bisect_ppx
- Marshaling of data structures.
  https://ocaml.org/api/Marshal.html
- QuickCheck inspired property-based testing for OCaml.
  https://github.com/c-cube/qcheck
- Expect-test - a cram like framework for OCaml.
  https://github.com/janestreet/ppx_expect

Example of tree generator's implementation using QCheck.

```
let gen_tree : tree QCheck.Gen.t =
  let open QCheck.Gen in
  sized @@ fix (fun self -> function
    | 0 -> map (fun x -> Leaf x) nat
    | n ->
      oneof [
        map (fun x -> Leaf x) nat;
        map2 (fun x y -> Node (x, y))
          (self (n / 2)) (self (n / 2));
      ])
```