FuzzChick

Beck, Calvin Huang, Jiani Li, Yishuai

December 5, 2018

What is this talk about?

What is this talk about?

FuzzChick!

What is this talk about?

FuzzChick!

What is FuzzChick...?

What is this talk about?

FuzzChick!

What is FuzzChick...?

FuzzChick is an experiment to improve QuickChick using ideas from fuzzing. This combines QuickChick with AFL [3].

What is this talk about?

FuzzChick!

What is FuzzChick...?

FuzzChick is an experiment to improve QuickChick using ideas from fuzzing. This combines QuickChick with AFL [3].

We'll come back to this...

QuickChick: A Brief Review

QuickChick is a properties based testing framework for Coq.

- You build (or derive) generators for data types.
- Using those generators you can feed data into test cases.
- These test cases can be any arbitrary predicate.

QuickChick: Pros and Cons

So what's great about QuickChick?

- Relatively easy to build / derive generators.
- Can generate lots of tests for specific properties automatically.

QuickChick: Pros and Cons

So what's great about QuickChick?

- Relatively easy to build / derive generators.
- Can generate lots of tests for specific properties automatically.

What's not so great about QuickChick?

QuickChick: Pros and Cons

So what's great about QuickChick?

- Relatively easy to build / derive generators.
- Can generate lots of tests for specific properties automatically.

What's not so great about QuickChick?

- Getting *good* generators can be hard!

What makes a good generator?

The basic idea of what makes a generator “good” can vary somewhat based on the context.

What makes a good generator?

The basic idea of what makes a generator “good” can vary somewhat based on the context.
In general you want good coverage. How can you achieve that with minimal work?

Finally, FuzzChick!

FuzzChick uses AFL to make the choices between constructors for building data types for tests.

Finally, FuzzChick!

FuzzChick uses AFL to make the choices between constructors for building data types for tests.
Why is this good?

FuzzChick Intuition

AFL uses DSE to attempt to get good coverage while fuzzing...

FuzzChick Intuition

AFL uses DSE to attempt to get good coverage while fuzzing...
Maybe we can utilize AFL's smarts to achieve better test coverage.

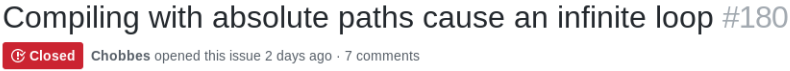
QuickChick: Now With Coverage!

We instrumented QuickChick using bisect_ppx to get coverage estimates!

QuickChick: Now With Coverage!

We instrumented QuickChick using bisect_ppx to get coverage estimates!

This *mostly* went smoothly...



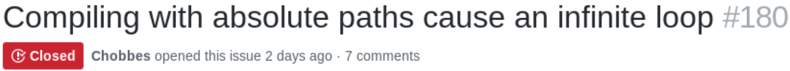
Compiling with absolute paths cause an infinite loop #180

Closed Chobbes opened this issue 2 days ago · 7 comments

QuickChick: Now With Coverage!

We instrumented QuickChick using bisect_ppx to get coverage estimates!

This *mostly* went smoothly...



Compiling with absolute paths cause an infinite loop #180

Closed **Chobbes** opened this issue 2 days ago · 7 comments

Maintainer fixed this issue promptly, which was *awesome*!

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

- Includes a lot of excessive extracted Coq code.

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

- Includes a lot of excessive extracted Coq code.
  - Like... All of QuickChick :(.
  - So, the percentages are a little off.

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

- Includes a lot of excessive extracted Coq code.
  - Like... All of QuickChick :(.
  - So, the percentages are a little off.
- QuickChick generates “random” files for each test, and the names aren’t all that useful
  - Modified QuickChick to include test case in name, but still not ideal.

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

- Includes a lot of excessive extracted Coq code.
  - Like... All of QuickChick :(.
  - So, the percentages are a little off.
- QuickChick generates “random” files for each test, and the names aren’t all that useful
  - Modified QuickChick to include test case in name, but still not ideal.

But...

QuickChick with coverage... Still needs some polish

QuickChick with coverage is cool, but it still does need some polish.

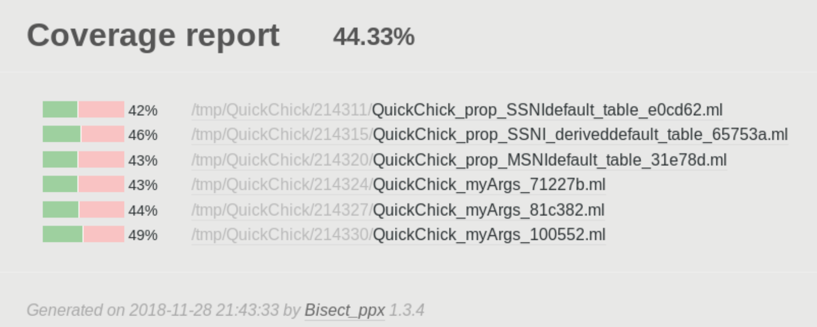
- Includes a lot of excessive extracted Coq code.
  - Like... All of QuickChick :(.
  - So, the percentages are a little off.
- QuickChick generates “random” files for each test, and the names aren’t all that useful
  - Modified QuickChick to include test case in name, but still not ideal.

But...

It works! We can measure stuff!

QuickChick Coverage: ifc-basic

Coverage with QuickChick in the ifc-basic example:



Coverage report 44.33%

42% /tmp/QuickChick/214311/QuickChick_prop_SSNIdefault_table_e0cd62.ml

46% /tmp/QuickChick/214315/QuickChick_prop_SSNI_deriveddefault_table_65753a.ml

43% /tmp/QuickChick/214320/QuickChick_prop_MSNIdefault_table_31e78d.ml

43% /tmp/QuickChick/214324/QuickChick_myArgs_71227b.ml

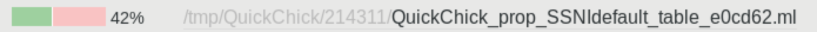
44% /tmp/QuickChick/214327/QuickChick_myArgs_81c382.ml

49% /tmp/QuickChick/214330/QuickChick_myArgs_100552.ml

Generated on 2018-11-28 21:43:33 by Bisect_ppx 1.3.4

QuickChick vs FuzzChick: ifc-basic

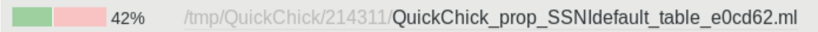
QuickChick:



42% /tmp/QuickChick/214311/QuickChick_prop_SSNIdefault_table_e0cd62.ml

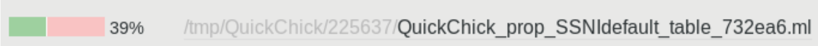
QuickChick vs FuzzChick: ifc-basic

QuickChick:



42% /tmp/QuickChick/214311/QuickChick_prop_SSNIdefault_table_e0cd62.ml

FuzzChick:



39% /tmp/QuickChick/225637/QuickChick_prop_SSNIdefault_table_732ea6.ml

For some reason it seems that FuzzChick actually gets worse coverage than QuickChick on this test case... At least in the time I let it run (I'm not terribly patient)

Why Worse Coverage?

- Just need to let it run longer?
    - AFL needs a while to “warm up”?

Why Worse Coverage?

- Just need to let it run longer?
  - AFL needs a while to “warm up”?
- QuickChick test already managed to get good coverage in this instance, so fuzzing doesn’t give us much on top of it?
  - Hard to tell what “good coverage” is due to the extraneous code extracted by QuickChick.

Why Worse Coverage?

- Just need to let it run longer?
  - AFL needs a while to “warm up”?
- QuickChick test already managed to get good coverage in this instance, so fuzzing doesn’t give us much on top of it?
  - Hard to tell what “good coverage” is due to the extraneous code extracted by QuickChick.
- Something’s not instrumented correctly?

Why Worse Coverage?

- Just need to let it run longer?
  - AFL needs a while to "warm up"?
- QuickChick test already managed to get good coverage in this instance, so fuzzing doesn't give us much on top of it?
  - Hard to tell what "good coverage" is due to the extraneous code extracted by QuickChick.
- Something's not instrumented correctly?
- This test case, for whatever reason, is fuzzer unfriendly?
  - Maybe extracted Coq could be fuzzer unfriendly? Lots of inefficient data types like like `nat` (basically a linked list whose length represents a number).
  - Could result in excessively long paths and hard to solve predicates for DSE?
  - Not sure that having pointers everywhere would be AFL's strength...

Some Further Coverage Testing...

Test case:

```
Extract Constant unlikely_branch =>
" fun i ->
  if (0 < i)
  then if (i mod 100 == 0)
       then if (i mod 1000 == 0)
            then if (i mod 10000 == 0)
                 then if (i mod 100000 == 0)
                      then if (i mod 1000000 == 0)
                           then if (i < 1000001)
                                then 42
                                else 0
                           else 0
                      else 0
                 else 0
            else 0
       else 0
  else 0
".


Definition always_zero := forAll (choose (0%Z, 9999999%Z)) (fun n =>
     unlikely_branch n =? 0).
```

Some Further Coverage Testing...

Test case:

```
Extract Constant unlikely_branch =>
" fun i ->
  if (0 < i)
  then if (i mod 100 == 0)
       then if (i mod 1000 == 0)
            then if (i mod 10000 == 0)
                 then if (i mod 100000 == 0)
                      then if (i mod 1000000 == 0)
                           then if (i < 1000001)
                                then 42
                                else 0
                           else 0
                      else 0
                 else 0
            else 0
       else 0
  else 0
".


Definition always_zero := forAll (choose (0%Z, 9999999%Z)) (fun n =>
     unlikely_branch n =? 0).
```

Trying to give AFL a good chance to find the failing branch...

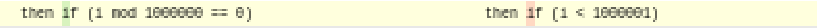
Results

In the equivalent C code AFL does quite well...

Results

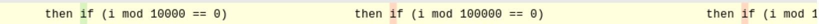
In the equivalent C code AFL does quite well...

QuickChick:



```
   then if (i mod 1000000 == 0)                                      then if (i < 1000001)
```

FuzzChick:

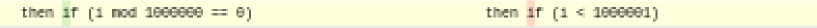


```
      then if (i mod 10000 == 0)                     then if (i mod 100000 == 0)                     then if (i mod 1
```

Results

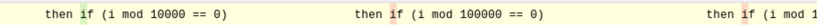
In the equivalent C code AFL does quite well...

QuickChick:



```
  then if (i mod 1000000 == 0)                              then if (i < 1000001)
```

FuzzChick:



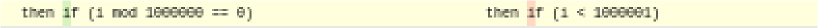
```
     then if (i mod 10000 == 0)                    then if (i mod 100000 == 0)                    then if (i mod 1
```

Here not so much? FuzzChick doesn't make it as far...

Results

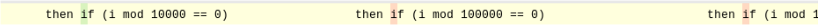
In the equivalent C code AFL does quite well...

QuickChick:



```
  then if (i mod 1000000 == 0)                          then if (i < 1000001)
```

FuzzChick:



```
    then if (i mod 10000 == 0)                    then if (i mod 100000 == 0)                        then if (i mod 1
```

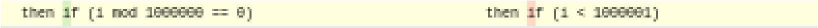
Here not so much? FuzzChick doesn't make it as far...

Well, in fairness, it does eventually, but it takes a good 30 minutes. QuickChick was much faster.

Results

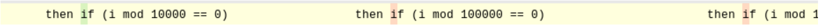
In the equivalent C code AFL does quite well...

QuickChick:



```
  then if (i mod 1000000 == 0)                    then if (i < 1000001)
```

FuzzChick:



```
     then if (i mod 10000 == 0)                  then if (i mod 100000 == 0)                       then if (i mod 1
```

Here not so much? FuzzChick doesn't make it as far...

Well, in fairness, it does eventually, but it takes a good 30 minutes. QuickChick was much faster.

Suggests maybe the extracted ocaml is harder for AFL to analyze? The C branches were discovered very quickly by AFL.

Performance

- Fuzzing is an order of magnitude slower than random testing.
- Performance bottleneck: disk access.
- Experiments to see whether the instrumentation overhead is worth it are still in preliminary stages.

Fuzzing a Large Project

How do QuickChick and FuzzChick perform on a large scale project?

Fuzzing a Large Project

How do QuickChick and FuzzChick perform on a large scale project?

We tried them on Apache's httpd,

Fuzzing a Large Project

How do QuickChick and FuzzChick perform on a large scale project?

We tried them on Apache's httpd,

And unfortunately they performed not so well...

Fuzzing a Large Project

Setting up the experiment:

coq $\xrightarrow{???}$ C (apache)

Fuzzing a Large Project

Setting up the experiment:

$$\text{coq} \xrightarrow{???} \text{C (apache)}$$

$$\text{coq} \xrightarrow{\textit{Extract}} \text{Ocaml} \xrightarrow{\textit{Unixcall}} \text{C (apache)}$$

Fuzzing a Large Project

Good news:

- Both Quickchick and FuzzChick successfully run on the apache server.

Fuzzing a Large Project

Good news:

- Both Quickchick and FuzzChick successfully run on the apache server.

Bad news:

- We failed to go deeper than the client side on the server.

Fuzzing a Large Project

Background:

- We are using the patch which is mentioned in AFL, so that the apache server will take in a string as its argument.

Fuzzing a Large Project

Background:

- We are using the patch which is mentioned in AFL, so that the apache server will take in a string as its argument.
- But this patch used pthread to launch the server and what is happening at the backend is obscure (aka. neither humanbeing nor AFL knows what it is doing).

Fuzzing a Large Project

Background:

- We are using the patch which is mentioned in AFL, so that the apache server will take in a string as its argument.
- But this patch used pthread to launch the server and what is happening at the backend is obscure (aka. neither humanbeing nor AFL knows what it is doing).
- The pthread is not AFL instrumented and AFL cannot analysis the underlying path in the blackbox and do its state driven fuzzing technique. (GDB results said so :p)

Fuzzing a Large Project

Since I don't know what is the string that will make Apache run normally. (Thanks to the lack of documentation and the pthread function the patch is using.) I want the fuzzers to help me capture what is a string that will make the patched apache run successfully (exit with 0).

Fuzzing a Large Project

Quickchick:

- **Pros:** Quickchick runs pretty fast at generating test cases.
- **Cons:** Quickchick fails to capture the successful case I want when we generate 10000 random strings. (That sounds natural I guess).

Fuzzing a Large Project

FuzzChick:

- **Pros:** FuzzChick runs AFL and AFL does not generate random string, but it can cheat on having some testscript that people wrote.
- **Cons:** It runs pretty slowly (1.2s per testcase). Maybe the string it comes up with is meaningful to the server.

Fuzzing a Large Project

What if we specify a richer spec to coq?

Fuzzing a Large Project

What if we specify a richer spec to coq?

Instead of getting the label that whether the apache exit normally ot not, we tried to obtain the stdout string from apache to coq.

Fuzzing a Large Project

What if we specify a richer spec to coq?

Instead of getting the label that whether the apache exit normally ot not, we tried to obtain the stdout string from apache to coq.

It's too messy to specify in the ocaml library that the extracted ocaml function is using. We cannot compile the program. :(

Fuzzing a Large Project

What if we specify a richer spec to coq?

Instead of getting the label that whether the apache exit normally ot not, we tried to obtain the stdout string from apache to coq.

It's too messy to specify in the ocaml library that the extracted ocaml function is using. We cannot compile the program. :(

Aborted.

Fuzzing a Large Project

Technical difficulties:

- The linkage between C to Ocaml and Ocaml to Coq is pretty messy. Code working in Ocaml does not necessarily compile when it is extracted to Coq.
- When we try to obtain more information from both of stderr and stdout in Ocaml, it is nontrivial to not stuck the terminal.

Fuzzing a Large Project

Takeaway:

It is not yet very practical to fuzz large real world project with coq and ocaml.

Honourable Mentions: Some Other Stuff We Did

Honourable Mentions: Some Other Stuff We Did

- Honggfuzz!

Honourable Mentions: Some Other Stuff We Did

- Honggfuzz!
  - Finally got this working...
  - Didn't really work out because it takes a long time to find bugs by fuzzing.
  - Decided it wasn't really a great comparison to FuzzChick which is a properties based testing tool anyway.
  - Some useful scripts / documentation to get this running in our git repo. [1]

Honourable Mentions: Some Other Stuff We Did

- Honggfuzz!
  - Finally got this working...
  - Didn't really work out because it takes a long time to find bugs by fuzzing.
  - Decided it wasn't really a great comparison to FuzzChick which is a properties based testing tool anyway.
  - Some useful scripts / documentation to get this running in our git repo. [1]
- Plain AFL!

Honourable Mentions: Some Other Stuff We Did

- Honggfuzz!
  - Finally got this working...
  - Didn't really work out because it takes a long time to find bugs by fuzzing.
  - Decided it wasn't really a great comparison to FuzzChick which is a properties based testing tool anyway.
  - Some useful scripts / documentation to get this running in our git repo. [1]
- Plain AFL!
  - Similar story to Honggfuzz.

Conclusion! Questions?

Whew! Questions?

References

- Calvin Beck, Jiani Huang, and Yishuai Li. *Quick700*. 2018. URL: https://github.com/Quick700/Quick700 (visited on 11/29/2018).
- Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. "Generating Good Generators for Inductive Relations". In: ().
- Michal Zalewski. *AFL*. URL: http://lcamtuf.coredump.cx/afl/ (visited on 11/29/2018).

These are all good resources! You should look at them!