BRIDGING FUZZING AND PROPERTY TESTING — 2023-07-10

It's been over three years since Fitzgen published: *"Announcing Better Support for Fuzzing with Structured Inputs in Rust"*, and a little over two years since arbitrary 1.0 was released. A few years agol wrote a property-testing crate based on `arbitrary` , but never ended up writing about it. So I wanted to take a moment to change that. We'll start by taking a look at automated testing, then cover the different approaches to it, explain how to work with structured inputs, and finally introduce heckcheck - a small property-testing library.

Update 2023-07-18: I recently learned about a different crate using a similar approach to mine: proptest-arbitrary-interop. The proptest crate is more mature than my `heckcheck` project, so if you're considering adopting the techniques described in this post - please check out the proptest interop crate too.

FUZZING VS PROPERTY TESTING

Both fuzzing and property testing are ways of automatically testing code. Where unit tests typically test some expected set of behavior, automated test have the ability to be more rigorous and exhaustive - making them far more likely to weed out bugs - especially those you didn't even conceive of. Fuzzing and property testing have a lot in common with each other. Where they tend to

differ is how tests are driven.

With fuzzing you typically use some external agent to test your program. Fuzzers usually have the ability to instrument the code under test, and make use of tools such as sanitizers to check whether invariants are violated. Fuzzers will also keep track of which input leads to which branches being hit in code, tailor their input to cover as many branches as possible. This process can take time, and is why it often pays off to run fuzzers for extended periods of time or even continuously. In Rust support for fuzzing is provided via the `cargo-fuzz` extension.

Property testing typically works the other way around: property testing is typically done by including a library in your test code which allows you to drive the tests yourself. Rather than looking for crashes or sanitizer failures the emphasis is more on checking your implementation using a strategy - more on those in a later section. While fuzzing tends to be more thorough, property testing tends to be faster to execute. Depending on what you're testing it's not uncommon to integrate property tests in your CI test suite - and run them every time you make a change. If you're writing unsafe Rust you can even combine property tests with miri to validate with more confidence whether the runtime behavior of your program matches Rust's operational semantics.

The final piece of both property testing and fuzzing is that they perform something called *test case shrinking* when they're done. When they encounter a failing test, what they'll do is try and simplify the input to the bare minimum to isolate the issue as much as possible. Once simplified it tends to be much easier to understand, and in turn can become easier to convert into unit tests.

There's a lot more to say about automated testing - including how to use it to mess with the ordering of concurrency primitives, or to intentionally mess with the network. But roughly speaking that should all just be a matter of wiring up drivers up to the same source of entropy coming from the automated testing drivers.

STRUCTURED INPUTS

In Fitzgen's post he shows how it's possible to combine fuzing with structured inputs using `arbitrary` crate. The way a fuzzer

typically works is that it generates random data which is passed to a program over some channel. Sometimes this data is generated from some corpus of data to try and weave in phrases we know the program might be looking for - such as `HTTP/1.1` or `GET` for an HTTP parser. But with `arbitrary` this randomness can be more clearly channeled: it can take the arbitrary stream of data, and use it to create structured types.

Part of the reason why Fitzgen did this work was to fuzz WASM programs. In *"Writing a Test Case Generator for a Programming Language"* he describes how using `cargo fuzz` and `arbitrary` he's able to generate endless amounts of new, valid WASM programs to pass to be used to test all sorts of WASM-related tools. This allows far more code to be exercised than solely using a corpus-based approach ever could.

To show an example of what it looks like to use cargo-fuzz, we can re-use the RGB parser/encoder example from Fitzgen's blog - but with the actual methods implemented. We start by defining a struct which takes red, green, and blue values:

```
use arbitrary::Arbitrary;

/// A color value encoded as Red-Green-Blue
#[derive(Arbitrary, Debug, PartialEq, Eq)]
pub struct Rgb {
    r: u8,
    g: u8,
    b: u8,
}

impl Rgb {
    /// Convert from RGB to Hexadecimal.
    pub fn to_hex(&self) -> String {
        format!("#{:02X}{:02X}{:02X}", self.r, self.g, self
    }

    /// Convert from Hexadecimal to RGB.
    pub fn from_hex(s: String) -> Self {
        let s = s.strip_prefix('#').unwrap();
        Rgb {
            r: u8::from_str_radix(&s[0..2], 16).unwrap(),
            g: u8::from_str_radix(&s[2..4], 16).unwrap(),
            b: u8::from_str_radix(&s[4..6], 16).unwrap(),
        }
```

```
    }
}
```

We can expose this to `cargo-fuzz` to perform a roundtrip-test by creating a new file containing the following:

```
// The fuzz target takes a well-formed `Rgb` as input!
libfuzzer_sys::fuzz_target!(|rgb: Rgb| {
    let hsl = rgb.to_hsl();
    let rgb2 = hsl.to_rgb();

    // RGB -> HSL -> RGB is the identity function. This
    // property should hold true for all RGB colors!
    assert_eq!(rgb, rgb2);
});
```

We can then execute it using `cargo fuzz` by running:

```
$ cargo fuzz run my_test
```

HECKCHECK: PROPERTY TESTING USING ARBITRARY

As we've covered the `arbitrary` crate works great when fuzzing. And because fuzzing and property testing are very similar, you'd assume that it should be possible to use `arbitrary` for property-based tests too right? I couldn't find a crate which did this, so I ended up writing one! Enter heckcheck, a really small property-testing library. If we take our earlier RGB example and adapt it to use heckcheck, it will look something like this:

```
/// Validate values can be converted from RGB to Hex and ba
#[test]
fn rgb_roundtrip() {
    heckcheck::check(|rgb: Rgb| {
        let hex = rgb.to_hex();
        let res = Rgb::from_hex(hex);
        assert_eq!(rgb, res);
        Ok(())
```

```
    });
}
```

And that's all we had to do. To test the code we just run `cargo test` as usual:

```
$ cargo test
```

The same code we would've written to interact with a fuzzer can now be used for property testing too. As I've said earlier: both fuzzing and property testing have more in common than not. The difference here compared to using a fuzzer is that this can be run as part of CI without a problem, executes really quickly, and is entirely portable. Fuzzers need to be installed on the system first, typically execute more slowly, and some platforms such as Windows have limited support for fuzzers.

I wrote `heckcheck` mostly to see whether it would be possible to write - and the implementation is maybe ~400 lines or so in total. I bet it could be improved, since in particular our shrinking code is less than ideal, while good approaches are known.

Compared to other popular property testing libraries `heckcheck` is more similar to `quickcheck` than it is to `proptest` . It has a simple interface, can generate complex values quickly, and uses external shrinking strategies. The proptest docs contain a comparison between proptest and quickcheck, and that same comparison could probably apply between proptest and heckcheck. The only place where the comparison may differ is that the `arbitrary` crate has support for customizing generated values. Say we wanted to restrict the values generated in our RGB implementation to only valid RGB values, we could change the implementation to the following:

```
use arbitrary::{Arbitrary, Unstructured};

/// A color value encoded as Red-Green-Blue
#[derive(Arbitrary, Debug, PartialEq, Eq)]
pub struct Rgb {
    #[arbitrary(with = color_range)]
    r: u8,
```

```
    #[arbitrary(with = color_range)]
    g: u8,
    #[arbitrary(with = color_range)]
    b: u8,
}

/// Generate a valid RGB color value
fn color_range(u: &mut Unstructured) -> arbitrary::Result<u
    u.int_in_range(0..=255)
}
```

If I could change one thing about the `arbitrary` crate it's that I'd like to see it support pattern-type notation inside the attributes. Patterns allow primitive values to be restricted to a subset of their possible inputs, which should be a fairly common use. With that we could imagine the RGB type could instead be written like this:

```
//! ⚠️ This is purely speculative and not currently support

use arbitrary::{Arbitrary, Unstructured};

/// A color value encoded as Red-Green-Blue
#[derive(Arbitrary, Debug, PartialEq, Eq)]
pub struct Rgb {
    #[arbitrary(pattern = 0..=255)]
    r: u8,
    #[arbitrary(pattern = 0..=255)]
    g: u8,
    #[arbitrary(pattern = 0..=255)]
    b: u8,
}
```

AUTOMATED TESTING STRATEGIES

Now that we've covered what the different ways of automated testing are, it's worth briefly touching on some common testing strategies. Because once you've understood that automated testing might be good, there often is a bit of a gap to actually implement it in your own code. And that's what testing strategies are for. These are some common testing strategies:

- **roundtrip testing**: generate a message, pass it to the encoder, then pass the encoder's output to the decoder. The

decoder's output should be the same as the original message.
- **differential testing**: test the program against a "known good" implementation of a similar program (also known as an "oracle"). This is useful if, for example, you're re-implementing an existing protocol or algorithm.
- **invariant testing**: test that a certain property *always* holds. This can be a universal property like: "my program didn't crash". But invariants can be specific too. Rain recently gave a great example that if you're for example testing a sorting function, you can check its correctness by making iterating over each number and checking it's bigger than the last number.

We've already covered roundtrip testing earlier in this post, and I believe invariant testing should be fairly straight forward. So as we're closing out here I just briefly wanted to show an example of how to perform differential testing using an approach Tyler Neely introduced me to a few years ago.

The way this works is that you use your source of entropy to generate a sequence of operations which you then apply to both your oracle and your program under test. Should your test trigger a failure, the shrinker will kick in and try and remove operations trying to uncover the minimal sequence of operations needed to trigger the bug. Say we in the process of building the smallvec crate; the implementation differs from the stdlib, but the observable behavior should be the same. Which means we can use the stdlib's `Vec` as our oracle for the test, which could look something like this:

```
/// A single operation we can apply
#[derive(Arbitrary)]
enum Operation {
    Insert(usize, usize),
    Fetch(usize),
}

#[test]
fn differential_smallvec_test() {
    heckcheck::check(|operations: Vec<Operation>| {
        // Setup both our subject and the oracle
        let mut subject = smallvec::SmallVec::new();
        let mut oracle = vec![];
```

```
        // Apply the same operations in-order to both the s
        // comparing outputs whenever we get any.
        for operation in operations {
            match operation {
                Insert(index, value) => {
                    oracle.insert(index, value);
                    subject.insert(index, value);
                }
                Fetch(index) => {
                    assert_eq!(oracle.get(index), subject.g
                }
            }
        }
    });
}
```

CONCLUSION

In this post we've covered what fuzzing and property testing are, how they're similar, and how they differ. We've introduced heckcheck, a small property-testing library and shown how you can use it to implement various testing strategies with.

I believe one of the greatest strengths Rust has is its ability to provide uniform experiences. And I believe that automated testing is one of the best tools we have to ensure the correctness of the implementations of our Rust programs. I'd love to one day see Rust provide facilities out of the box for automated testing - including a standard `Arbitrary` interface.

REFERENCES

▸ View all references