Property-based testing for Fred

I wrote this assuming you would read my 499 writeup first. It's over here if you want to look at it: https://github.com/ysthakur/fred/blob/main/writeup/writeup.pdf.

The goal of my final project here was to gain a little more faith in my implementation of the runtime for Fred, as well as the algorithm itself. This hasn't really happened, due to how limited my tests appear to be. However, I did uncover a couple of bugs while working on this (although manual testing would also have uncovered them at some point).

Testing strongly-connected components

My first test was for testing whether or not the compiler correctly created strongly-connected components of the types in the program it was given. My implementation used Tarjan's algorithm for doing this, since it not only creates strongly-connected components but also sorts them. However, I did not understand the algorithm at all, and so I blindly ported the pseudocode on Wikipedia to Scala. So I definitely needed to test this.

I made a generator to create a list of product types. Each type was represented a list of integers, where each integer represented a field pointing to the type at that index in the list of generated types. The generator is basically just this, followed by some code to generate types:

```
Gen.sequence(0.until(numTypes).map { i => Gen.listOf(Gen.choose(0, numTypes - 1)) })
```

Then, I get the strongly-connected components from this list of types. Two properties are checked:

- Are the strongly-connected components sorted? Types in later components cannot have references to types in previous components.
- Are the components strongly-connected? One problem with this check is that I had to make a `reachable()` method to check that every type in a given component was reachable from every other type in that component, and this method itself is not tested at all. Given how simple it is, it's proooobably fine.

This test is in `SCCTests.scala`.

Testing runtime behavior

Now that we have types, we can generate variables. After that, we can throw in assignments, like `set foo.field bar` and `set bar.field foo`, to create cycles between objects. Finally, we can insert `processAllPCRs(); VALGRIND_DO_CHANGED_LEAK_CHECK;` here and there, to remove all garbage at that point and then look for leaks using Valgrind. All of that can be put into a single `main` method, after which we can run the generated code and make sure there's no errors. This is more fuzzing than property-based testing, but whatever.

Before we generate variables, though, we need to modify the types a bit so that cycles can actually be formed at runtime. Suppose you have generated types like this:

```
// First strongly-connected component: T0 and T1
data T0 = T0 { f0: T1, f1: T2 }
data T1 = T1 { f0: T0, f1: T1 }
// Second strongly-connected component: Just T2
data T2 = T2 { f0: T2 }
```

You can't actually instantiate them. So what I did was generate optional types for every type, then insert mutable references to these optional types (note that `T0.f1` didn't have to be updated because `T2` is in a different strongly-connected component from `T0`):

```
data T0 = T0 { mut f0: OptT1, f1: T2 }
data OptT0 = SomeT0 { value: T0 } | NoneT0 {}
```

```
data T1 = T1 { mut f0: OptT0, mut f1: OptT1 }
data OptT1 = SomeT1 { value: T1 } | NoneT1 {}
data T2 = T2 { mut f0: OptT2 }
data OptT2 = SomeT2 { value: T2 } | NoneT2 {}
```

This lets you actually construct these types, e.g., `T2 { f0: NoneT2 {} }`. Now we can generate a randomly-chosen number of variables of each type.

Generating variables

There is one complication: objects will need to have references to other objects, e.g. when creating a `T0`, we'll need to give it a `T2`. Initially, I dealt with this by recursively creating variables, e.g. `genVars(T0)` would call `genVars(T2)` to get a variable of type `T2`. I needed to put in a recursion limit to get it to work, which wasn't great. Also, it created a bunch of trees, but it would be better to create all sorts of directed acyclic graphs.

So I got rid of that and took advantage of the fact that I already had a way to get strongly-connected components. We know that objects of types in earlier strongly-connected components will need references to objects of types in later strongly-connected components, but not backwards. So we will start by generating variables for types from the very last strongly-connected component and then move backwards to the top.

For example, with the types above, we would first generate some variables of type `T2`. It doesn't have any fields to fill in, aside from `f0`, which is a reference to a type in the same strongly-connected component and will therefore just be a `NoneT2`. After generating these variables, we would have a list of variables of type `T2`.

Then we would generate variables of type `T0` and `T1`. For `T0`, we would randomly pick a variable of type `T2` and use that for field `f1`.

Creating cycles

Now we can go back to all those optional fields we filled in with `NoneTX` earlier and put actual objects into them, causing cycles.

Here's some almost-pseudocode for how I generated assignment expressions to cause cycles:

- For every type `Ti`
  - For every field `fj` in `Ti` whose type is in the same strongly-connected component as `Ti`:
    - Randomly choose some variables of type `Ti`
    - For each variable `v`:
      - Get the type of the field. This should be something like `OptTk`
      - Randomly pick a variable `ref` of type `Tk`
      - Create the assignment expression `set v.fj (Some{value: ref})`

I have one test that simply generates code like this and then runs it with Valgrind to make sure that there's no memory issues. I didn't bother with putting in shrinking for that one. This test found one bug, but there's no way I wouldn't have found it myself within a few days with just manual testing.

Inserting Valgrind checks in between

Currently, Fred code only collects all the PCRs once, at the end of the `main` method. This is entirely because I couldn't be bothered to find and implement smarter ways to do it. This means that the test above will only run `processAllPCRs()` (the function that runs mark scan on all the PCRs) once, then check for leaks at the very end.

I also wanted to make sure that if `processAllPCRs()` were run in the middle of the program, there wouldn't be any leaks. So for my next test, I inserted after every statement, with probably 0.1, the C

code `processAllPCRs(); VALGRIND_DO_CHANGED_LEAK_CHECK;`. This should make sure that even while the program is running, there are no memory leaks. I couldn't figure out how to get Valgrind to ignore still-reachable blocks, but the test works.

However, I'm realizing as I'm typing out this report, 0.1 is very high given how small my generated programs are. Putting in so many `processAllPCRs()` calls means that I may not be giving my generated programs the opportunity to form big, complex structures. My first test should sort of account for that, but I can also play around with reducing the probability of inserting a Valgrind check.

Another problem with this test is that the programs it tests are not the same as the ones generated by the compiler. Right now, the compiler only decrements refcounts for local variables at the end of their scope. This didn't work for me for the fuzz tests, because it would mean that all of the objects would stay live until the end of `main`. So I currently have the generator insert code to decrement the refcount of every variable right after its last usage. The test also removes the compiler-generated refcount decrementing at the end of `main`, because that would cause use-after-frees.

The fact that inside this test, every variable is considered done after its last usage, while in normal Fred programs, every variable is considered done at the end of the block, could mean that it doesn't pick up on bugs that normal programs would run into. This is not great, but I can modify the compiler itself so that it considers variables done right after their last usage.

This test did catch one bug, but again, I probably would have caught it sooner or later using manual testing.

Shrinking

For shrinking, I just shrink the list of assignment expressions and Valgrind checks. I didn't get around to removing variables and removing fields from types, although this should be pretty doable.

Conclusion

My trust in my compiler and runtime haven't increased much. My current fuzzing covers a somewhat narrow range of behavior. The generated code also isn't the same as normal programs. So, the fact that my tests are currently passing doesn't mean all that much. I do plan on increasing generation sizes to see if I hit any new bugs.

I could also completely arbitrarily generate expressions. These programs would be less likely to hit interesting behavior, but it would also mean that if I let it run forever, all or nearly all behavior would eventually be explored. It wouldn't be terribly hard to set up, since I can just use my compiler to filter out ill-typed programs.