Automatically Generating QuickCheck Tests

Ben Holtz and Fraser Brown

June 5 and 12, 2014

1 Introduction

Occasionally, functions look like this:

```
-- This function should return an uppercase character
giveChar :: Char
giveChar = ...
```

Much of the function's desired behavior is presented in its comment–in fact, the comment loosely describes what the function means to do. In this project, we use this information to automatically check that functions are performing according to their expectations. We:

1. Extract function names, prototypes, and comments
2. Comb through the comments looking for candidate properties to QuickCheck
3. Prioritize the list of candidate properties using NLP strategies
4. Generate QuickCheck test cases for the most likely properties.

2 Extracting names, prototypes, comments

Given a source file to test, we read in the file and use regular expressions to gather comments and type information for each function. We store this information in (comment, prototype, function name) tripples; this way we can easily grab the comment information for test generation.

3 Identifying candidate properties

Given a function comment, we want to identifying possible QuickCheckable properties that should apply to that function. We initially handle this problem in three steps:

1. We consider every word in the function a possible property. This isn't always enough, however–sometimes, a property is described in two or three words (or more): "is prime" or "is not lowercase," for example. To handle this, we take consider each single word, each pair of words, and each tuple of words. The comment "value is prime" would generate "value," "is," "prime," "value is," "is prime," "value is prime." This is a stupid way to do things, but it works.
2. We look up each candidate property on Hoogle. This lookup generates a list of search results, which we then filter by function prototype–`function return type -> Bool`. The filter step eliminates about 90% of the possible properties; I don't imagine there's a good `Int -> Bool` function that corresponds to a Hoogle lookup of "value is." I actually just checked, and there isn't.
3. At this point, we have a list of properties and a corresponding list of possible Hoogle functions. Now it's time to rank the likelihood that they are, in fact, a reasonable properties to check.

3.1 Hoogle Lookups

In the service of documentationless future generations, we wanted to talk a little about the Hoogle API. In order to use Hoogle, you need a database to search. Obviously, in our case, the more extensive the database the better, but the size of the "all" database of Hackage is prohibitive: on the order of a gigbyte. In order to perform a search, the entire database to be searched needs to be loaded into memory, so a big machine is recommended for programmatic Hoogle use. A word to the wise– when using Hoogle to download databases (`hoogle data all`), the files are sneakily secreted off to `.cabal/share/hoogle/databases`.

In order to search a database, Hoogle builds a query out of plaintext you supply, using some simple parsing (the `parseQuery` function unsurprisingly handles this) to decide whether the query is intended to search types or names. We chose to search names, then to filter out the results by type. The results of a Hoogle search are vaguely unpleasant: they are a triple of locations (URLs and package names), function names, and corresponding comments. Of course, names and comments are preformatted for conversion to HTML, so they have to be shown with `showTagText`.

4 Ranking the property list

Given a list of properties and a corresponding list of Hoogle functions, we want to decide which functions most likely encapsulate a correct QuickCheck property.

We use the Stanford NLP toolkit for all of our NLP work, since it's far more established than anything we could make. In particular, we are using the Stanford parser, which can generate a dependency parse of English sentences and phrases. Given a sentence like "This function should not return an uppercase character," the parser generates the following dependency parse:

```
det(function-2, This-1)
nsubj(return-5, function-2)
aux(return-5, should-3)
neg(return-5, not-4)
root(ROOT-0, return-5)
det(character-8, an-6)
amod(character-8, uppercase-7)
dobj(return-5, character-8)
```

Notice here that the root of the comment is "return," that it is negated by "not," and that the direct object of "return" is "character," and "character" is modified by "uppercase." From this, it is fair to assume that the function does not return an uppercase character. Obviously, this is a simplification. This example from the wild generates the parse below: "Initialize an environment with a 'StateStore,' an input map, a preexisting 'DataCache,' and a seed for the random number generator":

```
root(ROOT-0, Initialize-1)
det(environment-3, an-2)
dobj(Initialize-1, environment-3)
prep(Initialize-1, with-4)
det(StateStore-7, a-5)
pobj(with-4, StateStore-7)
det(map-12, an-10)
nn(map-12, input-11)
conj(StateStore-7, map-12)
det(DataCache-17, a-14)
amod(DataCache-17, preexisting-15)
conj(StateStore-7, DataCache-17)
cc(StateStore-7, and-20)
det(seed-22, a-21)
conj(StateStore-7, seed-22)
prep(seed-22, for-23)
```

```
det(generator-27, the-24)
amod(generator-27, random-25)
nn(generator-27, number-26)
pobj(for-23, generator-27)
```

This is not so useful for determining the expected behaviour of the the function. Note that a "collapsed" version of the parse, which we have not had time to fully explore, does a slightly better job, correctly identifying (note the `prep_with`) the need to initialize the environment with a StateStore, a map, a DataCache, and a seed.

```
root(ROOT-0, Initialize-1)
det(environment-3, an-2)
dobj(Initialize-1, environment-3)
det(StateStore-7, a-5)
prep_with(Initialize-1, StateStore-7)
det(map-12, an-10)
nn(map-12, input-11)
prep_with(Initialize-1, map-12)
conj_and(StateStore-7, map-12)
det(DataCache-17, a-14)
amod(DataCache-17, preexisting-15)
prep_with(Initialize-1, DataCache-17)
conj_and(StateStore-7, DataCache-17)
det(seed-22, a-21)
prep_with(Initialize-1, seed-22)
conj_and(StateStore-7, seed-22)
det(generator-27, the-24)
amod(generator-27, random-25)
nn(generator-27, number-26)
prep_for(seed-22, generator-27)
```

More complicated comments aside, we can apply this parsing to get a sense of which functions found from searching Hoogle are more likely to match the desired behavior. Namely, we prioritize words modifying the direct object of return, or a synonym of return, as the most likely candidate for a testable property. For example, from "This function should not return an uppercase character," we should prioritize "uppercase."

Here's another example, for the comment "Takes an array and outputs the first element":

```
root(ROOT-0, Takes-1)
det(array-3, an-2)
dobj(Takes-1, array-3)
cc(array-3, and-4)
conj(array-3, outputs-5)
det(element-8, the-6)
amod(element-8, first-7)
nsubj(Takes-1, element-8)
```

We see that the adjective modifying the direct object here is "first", so we prioritize results returned by Hoogle containing the word first. Of course, we still want to hold onto all the results form all words in the comment, as the parser may be incorrect or the comment may not be perfectly descriptive. Any horribly incorrect functions found by Hoogle still have a chance at being filtered out in the next stage.

5 Generating QuickChecks

Once we have our list of candidate properties, it's time to generate QuickChecks. Right now, we have a template that we plug tester functions into, generating a string that can be evaluated at runtime. The failure of many runtime evaluations is an excellent hint that the property should not be used as a test.

After generating and QuickChecking our properties, we rank them again: a QuickCheck property that does not get past the first few tests is likely wrong. A QuickCheck property that passes all tests is likely right–assuming that people typically QuickCheck functions that they believe to work (or nearly work). To this end we use `quickCheckResult`, which returns information regarding the number of tests that pass, the number of shrinks and shrink attempts, and the nature of the test cases. In the future, we might take shrinking into account because of the implications of the complexity of the inputs that pass and fail, i.e. if a test case is shrinkable, it is likely less trivial, so multiple shrinks imply failure on more complex test cases. We adjust our rankings according to this information, and go on to print the best tests to a file for the user to modify if necessary. We want to supply the user with a nice mix of tests that pass and nearly-pass; all they need to do is look at our results to get a sense of how their functions perform.

6 Example

We wanted to include a toy example, since the example we presented in class didn't have all the NLP ranking hooked up. This is the function we're quick checking:

```
-- Takes a number and result is prime
nextPrime :: Int -> Int
nextPrime x = x^2 + x + 41
```

From this function, we generate bookkeeping info:

```
(nextPrime, Int -> Int, "Takes a number and result is prime")
```

From the bookkeeping info, we generate a list of candidate search terms:

```
["takes", "a", "number", "and", "result", "is", "prime"]
```

We note the dependency parse incorrectly connects "prime" as a modifier for "number" (it should modify "result").

```
root(ROOT-0, Takes-1)
det(number-3, a-2)
nsubj(Takes-1, number-3)
cc(number-3, and-4)
conj(number-3, result-5)
cop(prime-7, is-6)
rcmod(number-3, prime-7)
```

Next:

1. for each candidate property in the list, we do a Hoogle lookup for that property and filter the results by type signature. The only result is the function `isPrime` from the primes module.
2. we plug the functions into our template and run each of them as a QuickCheck property. We end up with the result that we fail after 81 tests with no shrinks.
3. Finally, we output the lone performing property, and any imports required for the property, to file for the user to compile, run, and examine. The file looks something like this:

```
import Test.QuickCheck
import <My Module>
import Data.Numbers.Primes

main :: IO ()
main = do
    putStrLn "Running auto-generated testing for nextPrime:is_prime"
    let propis_prime x = isPrime $ nextPrime x
          where types = x :: Int
    quickCheck propis_prime
```

7 Related Work

We were inspired by the *Macho: Programming with Man Pages* paper and a firey desire to QuickCheck the world.

The Macho system tries to automatically generate code from maual pages; for example, it managed to generate a moderately successful `pwd` program in Java. The system incorporates NLP, Database lookups, and type signature checks to generate these programs. "Usually," the authors explain, "the structure of the [man page] sentence can be directly transformed to requested computation: verbs imply action, nouns imply objects, and two nouns linked by a preposition imply some sort of converstion code" (Macho). Luckily, we weren't trying to do anything as complicated as whole-function generation; we just wanted to determine, from natural language, what properties should hold on the return value of a function. To this end, we drew even more inspiration from Macho: they found that a field of many function options could be radically pared down by filtering by type signature.

8 Why Haskell?

Haskell might not be an obvious choice for this project; in fact, if we'd been more familiar with the language when we started, we might not have chosen it. For one thing, Haskell code seems virtually commentless in the wild, and the comments that do exist are often explanatory–the user might have to add comments specially tailored for our system after the fact. That's exactly what we were trying to avoid. Still, Haskell was awesome for a few reasons:

- Organizing functions into pure and impure code is easy. It's simpler to organize code that has to read and write from disk, create and destroy files, and make calls to other languages when the bulk of the project is actually pure. Furthermore, the pure nature of Haskell makes it so great for simple testing like QuickCheck; we probably *couldn't* have done this project for Java.
- In the future, we might consider SafeHaskell: the arbitrary code pulled in from Hoogle, while likely not malicious, is still not technically trusted.

9 What's holding us back

Clearly, our tool has some limitations; for one thing, it only does lookups on Hoogle. Plenty of Haskell code is divorced from the Hoogle world. For another, our project can only handle unary functions. In the future, we would hope to take a Macho-like appoach to generating multi-part checks, stitching together functions by return type. This works best with a database of lots and lots of code. Hopefully, as Haskell gets more popular, a large database like this would become more feasible. Finally, it seems that Haskell programmers are not huge fans of traditional comments (though the Literate Haskell movement might disagree). In the future, syncing our comment reading up with Haddock may help.

10 References

We drew inspiration from a couple papers for this project:

Sridhara, Giriprasad. Automatic Generation Of Descriptive Summary Comments For Methods In Object-Oriented Programs. Diss. University of Delaware, 2012.

Cozzie, Anthony, Murph Finnicum, and Samuel T. King. "Macho: Programming With Man Pages." HotOS XII (2011): Web. 10 June 2014.