SOFTWARE FOUNDATIONS

VOLUME 4: QUICKCHICK: PROPERTY-BASED TESTING IN COQ

TABLE OF CONTENTS INDEX

QUICKCHICKTOOL

THE QUICKCHICK COMMAND-LINE TOOL

Overview

In this chapter we will introduce the QuickChick command-line tool, which supports testing of larger-scale projects than the examples we have seen so far. The command-line tool offers several useful features:

- Batch processing, compilation and execution of tests
- Mutation testing
- Sectioning of tests and mutants

This chapter reads a bit differently than most SF chapters, as it deals with a command-line tool. The code that it discusses, a simple compiler from a high-level expression language to a low-level stack machine, can be found in the directory `stack-compiler`, broken up into two files: `Exp.v` containing the high-level expression languages and `Stack.v` containing the low-level stack machine and the compiler. The chapter's text will tell you what to type on the command line or where to look in the subdirectory as needed.

To get started, let's try the tool out and see it's output! Go to the `stack-compiler` subdirectory and run the following command:

```
quickChick -color -top Stack
```

The `-color` flag colors certain lines in the output for easier reading. The `-top` flag controls the namespace for the compilation and should be the same as the `-R` or `-Q` command in your `_CoqProject` file. Running this command should produce quite a bit of output in your terminal.

The output consists of four parts, delimited by (colored) headers such as

```
Testing base...
```

or

```
Testing mutant 2 (./Exp.v: line 20): Plus-copy-paste-error
```

Let's take a closer look at the first one.

```
Testing base...
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Exp.optimize_correct_prop...
+++ Passed 10000 tests (0 discards)
Checking Stack.compiles_correctly...
+++ Passed 10000 tests (0 discards)
```

As we will see later in this chapter, the `QuickChick` command-line tool gathers `QuickChick` tests from the sources and runs them together in one single, efficient extraction. To do that, it copies all the files (here `Exp.v` and `Stack.v`) in a new subdirectory that is a "sibling" of the current one (that is both the directory where you ran `quickChick` and the new directory are subdirectories of the same parent). This new directory is always named `_qc_<DIRNAME>.tmp`. `QuickChick` also produces a new file `QuickChickTop.v` that contains all the tests that will be run and more extraction directives.

Following the output of the `QuickChick` command-line tool, all it does is compile everything in `_qc_stack-compiler.tmp`, using the `Makefile` of the original development as is and a `_CoqProject` modified to include the new `QuickChickTop.v` file. This compilation leads to extracting all necessary files in *separate* OCaml modules, which are in turn compiled using `ocamlbuild`, and then run. The separate extraction into distinct ocaml modules allows us to reuse compilation effort across different mutants as well as different calls to `quickChick`, as we can identify whether newly extracted modules are actually different and recompile them or not accordingly!

The rest of the 3 parts of the output are similar, with the main difference being that instead of running all the tests on the unaltered, base development, they run the same tests on *mutated* code. We will see exactly what mutation testing is later in this chapter.

If all is well, the last line should be a reassuring success report:

```
All tests produced the expected results
```

Arithmetic Expressions

The code in the `stack-compiler` subdirectory consists of two modules, `Exp` and `Stack`, each containing a number of definitions and properties. After some `Imports` at the top, the `Exp` module begins with a *section declaration*:

```
(*! Section arithmetic_expressions *)
```

We will explain what quickChick sections are and how to use them later in this chapter.

It then defines a little arithmetic language, consisting of natural literals, addition, subtraction and multiplication.

```
Inductive exp : Type :=
  | ANum : nat → exp
  | APlus : exp → exp → exp
  | AMinus : exp → exp → exp
  | AMult : exp → exp → exp.
```

Since `exp` is a simple datatype, QuickChick can derive a generator, a shrinker, and a printer automatically.

```
Derive (Arbitrary, Show) for exp.
```

The `eval` function evaluates an expression to a number.

```
Fixpoint eval (e : exp) : nat :=
  match e with
  | ANum n ⇒ n
  | APlus e1 e2 ⇒ (eval e1) + (eval e2)
  | AMinus e1 e2 ⇒ (eval e1) - (eval e2)
  | AMult e1 e2 ⇒ (eval e1) * (eval e2)
  end.
```

(The actual definition in the file `Exp.v` contains a few more annotations in comments, defining a *mutant*. We will discuss these annotations later in this chapter.

Now let's write a simple optimization: whenever we see an unnecessary operation (adding/subtracting 0) we optimize it away.

```
Fixpoint optimize (e : exp) : exp :=
  match e with
  | ANum n ⇒ ANum n
  | APlus e (ANum 0) ⇒ optimize e
  | APlus (ANum 0) e ⇒ optimize e
  | APlus e1 e2 ⇒ APlus (optimize e1) (optimize e2)
  | AMinus e (ANum 0) ⇒ optimize e
  | AMinus e1 e2 ⇒ AMinus (optimize e1) (optimize e2)
  | AMult e1 e2 ⇒ AMult (optimize e1) (optimize e2)
  end.
```

Again, the actual definition in `Exp.v` contains again a few more annotations in comments (another section annotation and another mutant).

We can now write a simple correctness property for the optimizer, namely that evaluating an optimized expression yields the same number as evaluating the original one.

```
Definition optimize_correct_prop (e : exp) := eval (optimize e) =
eval e?.


  (*! QuickChick optimize_correct_prop. *)

 QuickChecking optimize_correct_prop
 +++ Passed 10000 tests (0 discards)
```

QuickChick Test Annotations

In earlier chapters, we have included `QuickChick` commands in comments, with an invitation to the reader to uncomment and execute them. This has been done to avoid executing each and every test when compiling the volume as a whole. If we were to leave the `QuickChick` commands uncommented, then for each test we would extract the entire volume up to that point, compile the extracted OCaml, execute the test (up to 10000 tests by default for successes), and report the outcome. While this process is often adequate for small developments, it quickly becomes intractable for large Coq files, multi-file developments, or large numbers of properties that need to be tested.

One main feature of the command line tool is to gather all QuickChick commands, perform a *single* extraction and compilation pass, and report the results for all tests. This is achieved with special QuickChick annotations.

Notice that this QuickChick comment, just like all QuickChick-specific annotations in the file, begin with an exclamation mark. *Comments that begin with an exclamation mark are special to the QuickChick command-line tool parser and signify a test, a section, or a mutant*.

The annotation above defines a test of the property `optimize_correct_prop`. For simplicity, each test annotation requires a *named* property, like `optimize_correct_prop`. That is, while inline one could successfully execute a command like the one below, the command-line tool requires a defined constant in the test annotation.

```
QuickChick (fun e ⇒ eval (optimize e) = eval e?).
```

Sections

When in the middle of a large development, it is useful to be able to concentrate your tests in the parts of the development you are actively changing. For example, if you are playing with the optimizer for your high-level language it is not ideal to spend time re-running the (successful) tests of the code generator. This is where QuickChick's sections come in.

Sections are contiguous blocks of code within modules, and are allowed to depend on earlier ones. They contain sets of tests (and later on mutants) that correspond to a single aspect of the development and that are meant to be run together.

Thare are two kinds of section declarations in QuickChick. The first section declaration in the `Exp` module simply defines the start of a new block that can be identified by the name "arithmetic_expressions".

```
(*! Section arithmetic_expressions *)
```

The second also includes an `extends` clause.

```
(*! Section optimizations *)
(*! extends arithmetic_expressions *)
```

This signifies that this new block (until the end of the file, in this case, since there are no further section headers), also contains all tests and mutants from the

`arithmetic_expressions` section as well.

To see sectioning in action, execute the following command from the `stack-compiler` directory:

```
quickChick -color -top Stack -s optimizations
```

```
Testing base...
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Exp.optimize_correct_prop...
+++ Passed 10000 tests (0 discards)
... etc ...
```

In addition to the standard arguments (`-color`, `-top Stack`) we also specified that we only care about the `optimizations` section with the `-s` flag. Therefore this time, when testing the base development, only the single test in `optimizations` was executed.

Mutation Testing

The last major feature of the QuickChick command-line tool is *mutation testing*.

A question that naturally arises when random testing a software artifact is whether the testing is actually good. Does our property succeed for 10000 tests because everything is correct, or because we are not covering enough of the input space? How much testing is enough? Mutation testing can be used to answer such questions, by intentionally introducing bugs in either the artifacts or the properties we are testing and then checking that the tests can detect them.

Here is an excerpt of the `eval` function with a simple mutant annotation:

```
(*! *)
  | APlus e1 e2 ⇒ (eval e1) + (eval e2)
(*!! Plus-copy-paste-error *)
(*! | APlus e1 e2 => (eval e1) + (eval e1) *)
```

Let's break it down into its parts.

QuickChick mutants come in three parts. First, an annotation that signifies the beginning of a mutant. That is always the same:

```
(*! *)
```

This is followed by the correct code. In our example, by the evaluation of `APlus` e_1 e_2.

Afterwards, we can include an optional (but recommended) name for the mutant, which begins with two exclamation marks to help the parser.

```
(*!! Plus-copy-paste-error *)
```

Finally, we include mutations of the original code, each in a QuickChick single-exclamation-mark annotation. The code of the mutation is meant to be able to verbatim replace the original code. Here, a copy-paste error has been made to evaluate e_1 twice as both operands in an addition.

Similarly, in the `optimize` function we encounter the following mutant.

```
(*! *)
  | AMinus e (ANum 0) ⇒ optimize e
(*!! Minus-Reverse *)
(*!
  | AMinus (ANum 0) e => optimize e
*)
```

This bug allows the optimization of `0-e` to `e` instead of `e-0` to `e`.

To see these mutants in action, let's look at the rest of the output of the last `quickChick` command we ran:

```
quickChick -color -top Stack -s optimizations
```

```
Testing mutant 0 (./Exp.v: line 35): Minus-Reverse
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
AMinus (ANum 0) (ANum 1)
Checking Exp.optimize_correct_prop...
*** Failed after 13 tests and 4 shrinks. (0 discards)

Testing mutant 1 (./Exp.v: line 20): Plus-copy-paste-error
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
APlus (ANum 1) (ANum 0)
Checking Exp.optimize_correct_prop...
*** Failed after 5 tests and 3 shrinks. (0 discards)
All tests produced the expected results
```

After running all the tests for `base` (the unmutated artifact), the `quickChick` tool proceeds to run the single test in the `optimizations` section for each of the mutants it finds. Since

the `optimizations` section *extends* the `arithmetic_expressions` section, the mutants from both sections will be included. As expected, the optimize property fails in both cases.

A Low-Level Stack Machine

The second module of our development is `Stack.v`, which describes a low-level stack machine for arithmetic expressions. It contains a single section, `stack_instructions` which extends `arithmetic_expressions` but not `optimizations`. This allows us to independently run tests for the compiler or the optimzer without worrying about extra tests or mutants.

We begin by defining a low-level stack machine instruction set, which closely corresponds to the high-level expression language.

```
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SPlus : sinstr
| SMinus : sinstr
| SMult : sinstr.
```

Then we describe how to execute the stack machine.

```
Fixpoint execute (stack : list nat) (prog : list sinstr) : list nat
:=
  match (prog, stack) with
  | (nil, _ ) ⇒ stack
  | (SPush n::prog', _ ) ⇒ execute (n::stack) prog'
  | (SPlus::prog', m::n::stack') ⇒ execute ((m+n)::stack') prog'
  | (SMinus::prog', m::n::stack') ⇒ execute ((m-n)::stack') prog'
  | (SMult::prog', m::n::stack') ⇒ execute ((m*n)::stack') prog'
  | (_::prog', _ ) ⇒ execute stack prog'
  end.
```

Given the current `stack` (a list of natural numbers) and the `prog`ram to be executed, we will produce a resulting stack.

- If `prog` is empty, we return the current `stack`.
- To `Push` an integer, we cons it in the front of the list and execute the remainder of the program.
- To perform an arithmetic operation, we expect two integers at the top of the stack, operate, push the result, and execute the remainder of the program.
- Finally, if such a thing is not possible (i.e. we tried to perform a binary operation with less than 2 elements on the stack), we ignore the instruction and proceed to the rest.

Now we can compile expressions to their corresponding stack-instruction sequences.

```
Fixpoint compile (e : exp) : list sinstr :=
  match e with
  | ANum n ⇒ [SPush n]
  | APlus e1 e2 ⇒ compile e1 ++ compile e2 ++ [SPlus]
  | AMinus e1 e2 ⇒ compile e1 ++ compile e2 ++ [SMinus]
  | AMult e1 e2 ⇒ compile e1 ++ compile e2 ++ [SMult]
  end.
```

In the compilation above we have made a rookie compiler-writer mistake. (Can you spot it without running QuickChick?)

The property we would expect to hold is that executing the compiled instruction sequence of a given expression `e`, would result in a single element stack with `eval e` as its only element.

```
Definition compiles_correctly (e : exp) := (execute [] (compile e)) =
[eval e]?.

(*! QuickChick compiles_correctly. *)
```

```
===>
    QuickChecking compiles_correctly
    AMinus (ANum 0) (ANum 1)
    *** Failed after 3 tests and 2 shrinks. (0 discards)
```

The problem is that subtraction is not associative and we have compiled the two operands in the wrong order! We can now log that mutant in our development as shown in the `Stack` module.

```
    Fixpoint compile (e : exp) : list sinstr :=
      match e with
      | ANum n ⇒ [SPush n]
    (*! *)
    | APlus  e1 e2 ⇒ compile e2 ++ compile e1 ++ [SPlus]
      | AMinus e1 e2 ⇒ compile e2 ++ compile e1 ++ [SMinus]
      | AMult  e1 e2 ⇒ compile e2 ++ compile e1 ++ [SMult]
    (*!! Wrong associativity *)
    (*!
      | APlus  e1 e2 => compile e1 ++ compile e2 ++ SPlus
      | AMinus e1 e2 => compile e1 ++ compile e2 ++ SMinus
      | AMult  e1 e2 => compile e1 ++ compile e2 ++ SMult
    *)
      end.
```

We can now run a different command to test `compiles_correctly`, using `-s stack` to only check the `stack` section.

```
    quickChick -color -top Stack -s stack
```

```
    Testing base...
    make -f Makefile.coq
    make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
    COQDEP VFILES
    COQC Exp.v
    COQC Stack.v
    COQC QuickChickTop.v
```

```
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
Checking Stack.compiles_correctly...
+++ Passed 10000 tests (0 discards)

Testing mutant 0 (./Stack.v: line 33): Wrong associativity
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
AMinus (ANum 0) (ANum 1)
Checking Stack.compiles_correctly...
*** Failed after 5 tests and 6 shrinks. (0 discards)

Testing mutant 1 (./Exp.v: line 20): Plus-copy-paste-error
make -f Makefile.coq
make[1]: Entering directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
COQDEP VFILES
COQC Exp.v
COQC Stack.v
COQC QuickChickTop.v
make[1]: Leaving directory '/home/lemonidas/sfdev/qc/_qc_stack-compiler.tmp'
APlus (ANum 0) (ANum 1)
Checking Stack.compiles_correctly...
*** Failed after 5 tests and 2 shrinks. (0 discards)
All tests produced the expected results
```

We can see that the main property succeeds, while the two mutants (the one in `arithmetic_expressions` that was included because of the extension and the one in `stack`) both fail.

QuickChick Command-Line Tool Flags

For more information on the tool's flags, look at the reference manual in QuickChickInterface.