Advanced Programming 2022

Testing Basics

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 15, 2022

Today's lecture

- Refresher of testing principles
  - Mostly common sense
  - Should be familiar, but maybe forgotten
  - Focus on topics particularly relevant for AP
- Practical testing with GHC/Tasty
  - Setting up test suites
  - Checking test coverage

Why test systematically?

- Relatively low-effort to get reasonable assurance of correctness
  - Sufficient for most, except safety/mission-critical applications
    - avionics, nuclear plant control, high-frequency trading, ...
  - Still much less labor-intensive than full formal verification
    - Although progress is being made there...
- Underappreciated, even by people who should know better.
  - Recent DIKU MSc graduate works in high-profile, strongly knowledge-based software company that shall remain unnamed.
  - New middle management: “We spend an awful lot of effort on testing our software; what’s the business case for that?”
  - Time to leave...
- Not the pure overhead it may seem like.
  - Can integrate in development process: write tests *before* code.
  - May help uncover incomplete or inconsistent specifications *early*.
  - Working out expected result in *particular* case will usually suggest a possible *general* algorithm to implement.

Unit testing

- Main testing activity in AP
- Systematic testing of each module's functionality, according to its specification.
- **Also:** integration testing
  - Do separately tested modules actually fit together as intended?
    - Or were "agreed-upon" specs interpreted in inconsistent ways by implementer and user of module?
  - Usually not a *major* concern in AP
    - At most a handful of modules, (usually) pre-validated specs.
    - But we may test your *individual* modules with known-good implementations of the others.
  - Relevant when task expects you to specify (part of) *internal* interfaces
- **Also:** acceptance testing
  - Does final program actually work in the "real world"?
  - AP task may come with a couple of larger examples
    - Not nearly enough to just test on those!

Test suite is a **deliverable**

- Essential companion to the code itself
  - Needs to (be able to) evolve with it
- General SE principles still apply
  - Organize logically
  - Avoid duplication of boilerplate code
  - Try to separate declarative and procedural parts
  - Readable layout, comments, ...
- Will also be assessed as part of assignment/exam.
  - Does suite find (or indicate absence of) bugs in your code?
  - Does suite find (without false positives!) bugs in *non*-your code?
  - Usually not weighted **quite** as much as code itself,
    - Unless testing is the explicit objective of a subtask.
- Two-person teams can use *developer–tester* methodology:
  - One person develops the code, the other the test cases.
  - In industry, often close to 1-1 parity developers/testers
  - *Complementary* to pair programming
  - Remember to switch roles frequently!

Black-box testing

- Should be the bulk (maybe all) of test suite.
- Tests derived from *specification*, not from the code itself.
  - Example: the OnlineTA tests
  - Should work with *any* correct implementation of spec, not only your own.
    - At the very least, must be able to compile against it
  - Can later completely re-implement module (e.g., for improved performance) while reusing test cases.
- Only tests the exported/accessible functionality.
  - Ultimately, the only one that matters
- Don't usually need to maintain complete mental firewall between code and tests.
  - Fine to let knowledge of the code *inform* what parts or aspects of specification need particularly thorough testing.
  - The longer/more complex the code, the more tests are usually needed.

White-box testing

- **Supplement** to black-box testing, sometimes appropriate.
- Tests specific to particular implementation, may not meaningfully apply to others.
- Two main uses:
  - Testing internal (non-exported) functionality that would be substantially difficult to test adequately through exported API.
    - Crucial to first formulate an *explicit* specification of that functionality, as a source of test cases.
  - Expecting specific behavior, where specification allows many.
    - Better than not testing at all...
    - May give false positives when implementation subsequently tweaked or redone.
    - Wherever possible, try to replace with tests that will work in general (some tips later).
- Should be clearly separated from general (black-box) tests, typically placed in separate file.

Writing tests (1)

- All tests should have a predetermined *expected result*.
  - Just showing the actual output for some inputs is **not** a test.
- Tests should be *automated*.
  - Should be easy to run a test suite and get a summary report (indicating, at the very least, whether all tests succeeded).
- In some cases, may be infeasible or impractical to test some functionality in automated way.
  - Canonical example: program *should* loop forever in some case.
  - Also (especially in Haskell): program *should* abort with a runtime error/exception
    - Dubious API/spec design; normally prefer an explicit error result.
  - In those cases, it's OK to explain why some functionality was only "tested" informally/non-automatedly.
- Tests should be *logically organized*.
  - Purpose of each test should be clear from its name and/or placement in hierarchically organized test suite.
  - Makes it easier to systematically verify that all parts and aspects of spec are reasonably covered.

Writing tests (2)

- Test cases should be *simple*.
  - Each case should ideally check one part or aspect of functionality
  - Beware of complex tests checking lots of things at once:
    - if test succeeds, what can we really conclude with good confidence?
    - if test fails, still a lot of work to do to isolate the problem
- Really useful test suites should be *diagnostic*.
  - Pattern of successful and failed tests should ideally indicate likely problem area
  - Generally overkill for AP (as a student!)
- Test cases should be easily *human-checkable*
  - Beware of “promoting” *actual* outputs to *expected* after a cursory look-over.
    - Sometimes it’s evident that even the look-over step was skipped.
  - If expected output is cumbersome to work out manually, test case is probably too complex.

How much to test

- "Enough to verify all specified functionality, but not much more."
  - Diminishing returns for both
    - uncovering additional bugs
    - increasing confidence in correctness
- Usually a good idea to include at least some (failing) test cases for functionality that you haven't implemented at all.
  - Shows that you have at least thought about what the correct results should be.
  - But don't over-do it; concentrate on actual supposed-to-be-working code.
- Don't add test cases that are intuitively unlikely to fail, given earlier similar successes.
  - **Ex:** one test for each arithmetic operator (+ branch/error cases) probably enough.
  - But *do* test every single operator
    - copy+paste+tweak errors are a common source of bugs.

Test coverage

- Rule of thumb: test cases should (between them) exercise every single fragment of your code at least once.
  - Except for (clearly marked) “impossible” cases.
  - If you can’t see how to reach some (non-trivial) code by concrete tests, it probably shouldn’t be there.
  - May not always be sufficient, but usually comes close.
- Use `stack test --coverage`!
  - Computes coverage percentages: not very useful on its own
  - But also generates a beautifully colored code listing!
  - Don’t have to include in report, but definitely use as guide/safety net for your testing.
- **Tip:** mine assignment/exam text for relevant test cases!
  - Examples usually highlight tricky points that may not be immediately evident from the textual specification.
    - Behavior that directly contradicts an explicit example, but isn’t reflected as a failing test case, may be judged particularly harshly.
  - Don’t have to make *every* example into a test case, but do make a conscious decision about each one.

Dealing with non-functional specifications (1)

- Frequent source of confusion in connection with AP.
- Often, but far from always, specifications are *functional*:
  - *Mathematical* sense, not *programming-paradigm* one.
  - For any possible input, there is *exactly one* correct output.
  - Fairly straightforward to write suitable test cases.
- Sometimes, a specification may be deliberately vague: "...; otherwise, the function should return a suitable, `Left`-tagged error message."
  - Black-box test case should *not* check for the exact message your code happens to give, just that it is of the form `Left "..."`.
  - Maybe message will later be deemed imprecise or confusing; should be possible to change it without breaking test suite.

Non-functional specifications (2)

- Sometimes the specification may be explicitly *relational*
  - **Ex:** "Define `invSquare:: Double -> Double`, so that `invSquare` x (where $x \geq 0$) returns a y such that $y^2 = x$."
    - (Let's ignore round-off issues for simplicity.)
  - Bad test: `invSquare 4.0 == 2.0`.
  - Better: `let y = invSquare 4.0 in y == 2.0 || y == -2.0`
  - Even better: `let y = invSquare 4.0 in y*y == 4.0`.
- Side issue: what if we call `invSquare (-4.0)`?
  - Uses function in situation where its behavior is *not* specified.
  - Not really possible to (black-box) *test* anything here, because we have no idea of what to expect.
    - Formally correct behavior would include returning NaN, 0.0, -2.0, or 666.0; or calling `error`; or deleting all files.
    - Hopefully, developer didn't pick last option...
  - Matter of code *robustness*, and highly context-dependent.
    - Is it more important that the code keeps running at all costs, or that it avoids producing even slightly incorrect final results?

Testing relational specifications

Two general approaches:

1. Formulate test cases to explicitly separate all correct from all incorrect outputs:
   - Bad: `output == Left "Attempted division by zero"`
   - OK: `Data.Either.isLeft output` (or `isLeft output == True`)
2. (Not always feasible): *normalize* all equivalent outputs to some *canonical* representative and expect (only) that:

   ```
   norm output == Left msg where
     msg = "<message about division by 0>"
     norm (Right x) = Right x  -- don't touch proper results
     norm (Left e) | length e <= 3 = Left e -- uninformative?
     norm (Left _) = Left msg  -- assume it was sensible
   ```

   Then makes sense to present the actual (non-normalized) and expected outputs if test fails.

Bigger example: polynomial roots

```haskell
type IntPoly = [Integer] -- coefficients, highest power first
myPoly :: IntPoly; myPoly = [1,0,-4]  -- myPoly(x) = x^2 - 4

evalPoly :: IntPoly -> Integer -> Integer -- evaluate polynomial
rootsPoly :: IntPoly -> [Integer]
  -- find all _integer_ roots, in unspecified order
```

- How to test? Sample test cases (clearly need more):
  - `evalPoly myPoly 5 == 21`: unproblematic
  - `rootsPoly myPoly == [2,-2]`: bad!
  - Normalize: `Data.List.sort (rootsPoly myPoly) == [-2,2]`
- **Aside:** can actually test *soundness* of `rootsPoly` generically:

```haskell
sound myPoly && soundp myPoly2 && ... where
 sound p = all (\r -> evalPoly p r == 0) (rootsPoly p)
```

- And even *completeness* (partially):

```haskell
 complete p = all (\r -> evalPoly p r /= 0 ||
                        r `elem` rootsPoly p) [-1000..1000]
```

- Maybe can even generate test cases automatically? Later lecture.

(Potentially) buggy specifications

- Sometimes the specification is *accidentally* incomplete:
  - **Ex:** the description of a function doesn't mention one of its parameters at all.
    - Possible, but very unlikely, intended interpretation: the parameter should simply be ignored.
  - Other (more subtle) cases of where behavior appears underspecified, without the specification explicitly noting so.
  - Ask for clarification!
    - Maybe the specification is OK, but you are just misreading it.
    - Forestalls problems in integration phase of project.

- Sometimes the specification may even be *inconsistent*.
  - Typically one part of document subtly contradicts another, distant part.
  - Probably something got changed, but rest wasn't consistently updated.
  - Definitely ask for clarification!

Use a testing framework!

- In principle, automated test suite could be just
  `tests = (test1 && test2 && ...&& testn)`
  - With a little wrapper to be runnable with `stack test`
  - Maybe OK if all tests succeed, but otherwise?
- Use a *testing framework* to show which tests (or at least how many) were run and which succeeded.
  - A very rudimentary skeleton was included in Assignment 1.
  - Not really usable for larger test suites.
- Use an existing, well engineered framework such as `Tasty`.
  - Hierarchical test organization makes it easy to represent what is tested and how thoroughly.
    - (Possible to run subsets of tests selectively)
  - Robust in the face of individual test failures.
    - Including crashes/exceptions, divergence (timeout).
  - Readable presentation of test results
    - In particular, shows expected+actual outputs for failing test cases.
    - Custom formatting for relational tests also possible
  - (Support for complex test setup/teardown)

Using Tasty+HUnit for assignments/exam

- The skeleton test suite distributed with the assignment code is usually a good starting point.
  - But preferably add some tree structure!
- Distinguish clearly (e.g. by grouping or naming) between *positive* and *negative* tests.
  - The latter check that a variety of relevant error conditions are correctly detected and reported.
- Particular point of care: make sure to write test cases as "*actual* `@?=` *expected*" (or "*expected* `@=?` *actual*");
  - If you accidentally swap them, output is going to be very confusing!
- See Hoogle (`Test.Tasty`, `Test.Tasty.HUnit`) for fuller documentation, especially if you want to do something fancier.
  - Probably not needed for AP, but might be relevant if you use Tasty for more substantial projects.

What next?

- If time, brief presentation of Assignment 2 highlights.
- This afternoon: exercise labs (mostly same rooms as last Thursday, but only 3 rooms in late batch)
  - Work on monad exercises and/or Assignment 2
  - Last chance for in-person help with Assignment 1
- Next week: parsing
  - Monads again!