SM2-TES: Functional Programming and Property-Based Testing, Day 12

Jan Midtgaard

MMMI, SDU

Today

- □ Project presentations
- □ Final course evaluation
- □ Exam/project report
- □ Course summary

[Project presentations]

[Final course evaluation]

Exam and project report

Due to the corona situation

the oral exam in June will be online.

The form is:

- ☐ you make a group presentation of your project
- ☐ you receive and answer questions individually
- ☐ you get a combined grade for the written+oral part

The deadline for hand-in is unchanged: May 31

Course Summary

Functional programming, briefly

Functional programming emphasizes purity over side-effects (assignment and state, exceptions):

- functions (as first-class citizens)
- recursion
- lists (+ `fold`, `map`, `iter` for typical list processing)
- algebraic data types
- pattern matching

Functional programming, briefly

Functional programming emphasizes purity over side-effects (assignment and state, exceptions):

- functions (as first-class citizens)
- recursion
- lists (+ `fold`, `map`, `iter` for typical list processing)
- algebraic data types
- pattern matching

A function's type signature induces a code skeleton.

Functional programming, briefly

Functional programming emphasizes purity over side-effects (assignment and state, exceptions):

- functions (as first-class citizens)
- recursion
- lists (+ `fold`, `map`, `iter` for typical list processing)
- algebraic data types
- pattern matching

A function's type signature induces a code skeleton.

With tail-call optimization a recursive function compiles down to a loop.

Functional programming, briefly

Functional programming emphasizes purity over side-effects (assignment and state, exceptions):

- ☐ functions (as first-class citizens)
- ☐ recursion
- ☐ lists (+ `fold`, `map`, `iter` for typical list processing)
- ☐ algebraic data types
- ☐ pattern matching

A function's type signature induces a code skeleton.

With tail-call optimization a recursive function compiles down to a loop.

OCaml's type system even corresponds to a formal logic!

Property-based testing (PBT)

Property-based testing is also known as QuickCheck.

It phrases tests in terms of

- ☐ a generator (producing test input)
- ☐ a property (what must hold?)

Property-based testing (PBT)

Property-based testing is also known as QuickCheck.

It phrases tests in terms of

- ◻ a generator (producing test input)
- ◻ a property (what must hold?)

A shrinker cuts a counterexample down to a more comprehensible one.

Property-based testing (PBT)

Property-based testing is also known as QuickCheck.

It phrases tests in terms of

- □ a generator (producing test input)
- □ a property (what must hold?)

A shrinker cuts a counterexample down to a more comprehensible one.

QuickCheck offers builtin generators (`float`, `char`, . . .)

Generators compose nicely (`pair`, `triple`, `list`, . . .)

Property-based testing (PBT)

Property-based testing is also known as QuickCheck.

It phrases tests in terms of

- □ a generator (producing test input)
- □ a property (what must hold?)

A shrinker cuts a counterexample down to a more comprehensible one.

QuickCheck offers builtin generators (`float`, `char`, ...)

Generators compose nicely (`pair`, `triple`, `list`, ...)

Properties depend heavily on the domain, but there are common patterns (idempotency, round-trip, oracle, ...)

Model-based testing

Model-based testing with a state-machine framework compares the system-under-test to a model.

Model-based testing

Model-based testing with a state-machine framework compares the system-under-test to a model.

For systems with state this may be a viable option.

Random command sequences help bring the system-under-test into arbitrary states. . .

Model-based testing

Model-based testing with a state-machine framework compares the system-under-test to a model.

For systems with state this may be a viable option.

Random command sequences help bring the system-under-test into arbitrary states. . .

This approach tests the interaction of several commands.

State-machine preconditions further describe which commands are allowed when (a protocol).

Model-based testing

Model-based testing with a state-machine framework compares the system-under-test to a model.

For systems with state this may be a viable option.

Random command sequences help bring the system-under-test into arbitrary states...

This approach tests the interaction of several commands.

State-machine preconditions further describe which commands are allowed when (a protocol).

State-dependent command generation can be useful.

Write model-based tests by hand – or use a framework.

Applicability of property-based testing

We studied PBT of a range of setups:

- ☐ a simple deterministic API: testing simple properties
- ☐ a system/API with state: using a model-based test
- ☐ a concurrent system: using a parallelized model test

Applicability of property-based testing

We studied PBT of a range of setups:

- a simple deterministic API: testing simple properties
- a system/API with state: using a model-based test
- a concurrent system: using a parallelized model test

We don't (necessarily) need to PBT a system in language X from language X itself.

PBT can be used both black-box and white-box.

PBT can be used both for positive and negative testing.

For negative testing and security hardening, fuzz testing is a good choice.

Improving property-based tests

Generated tests are only as good as the generators and properties that produce them:

- generators should exercise boundary cases and not be confined to a subset of the input space
- properties should characterize intended behavior

Improving property-based tests

Generated tests are only as good as the generators and properties that produce them:

- generators should exercise boundary cases and not be confined to a subset of the input space
- properties should characterize intended behavior

Several tools can help us understand and improve PBTs:

- Statistics
- Coverage reports
- Fault injection
- Thinking like the devil's advocate
- Wrong properties (and counterexamples)
- Grammars for capturing all valid input in a spec.