SM2-TES: Functional Programming and Property-Based Testing

Jan Midtgaard

MMMI, SDU

SDU

Introduction (1/4)

As the title shows this course is two-fold.

We will study both

- functional programming and
- property-based testing

Lectures will be a mixture (a little of this, a little of that).

Introduction (1/4)

As the title shows this course is two-fold.

We will study both

- functional programming and
- property-based testing

Lectures will be a mixture (a little of this, a little of that).

Specifically we will

- use functional programming as a vehicle
- to learn property-based testing.

Introduction (2/4)

Programming to study testing?!?

Introduction (2/4)

Programming to study testing?!? Yes, exactly

Introduction (2/4)

Programming to study testing?!? Yes, exactly

Functional programming is an approach that

- emphasizes stateless programming
- uses first-class functions and recursion
- leads to programs closer in spirit to math

Introduction (2/4)

Programming to study testing?!? Yes, exactly

Functional programming is an approach that

- emphasizes stateless programming
- uses first-class functions and recursion
- leads to programs closer in spirit to math

Property-based testing

- is also know as *'QuickCheck'*
- is a powerful approach to automated testing
- grew out of functional programming

Introduction (3/4)

In this semester, you will learn

- OCaml – a functional programming language
- the principles and practice of QuickCheck

Introduction (3/4)

In this semester, you will learn

- OCaml – a functional programming language
- the principles and practice of QuickCheck

Why?

- QuickCheck is based on testing properties
- These are most easily expressed in a functional language with roots in mathematics and logic
- You can still QuickCheck software written in other languages
- Once we agree on the involved concepts you get to study other QuickCheck frameworks

Introduction (4/4)

We will focus on learning concepts rather than products

The QuickCheck concepts we will cover are
language independent

The functional programming concepts we will need are also universal:

- They are as relevant to
  F#, Scala, Haskell, Standard ML, Clean, . . .
- They may come in handy next time you program,
  e.g., callbacks in JavaScript.

Practicalities

The course will consist of

- lectures
- exercises
- a course project
- a project presentation
- a project report

We'll have an oral exam where you'll receive a combined grade for the report and presentation.

Measure of success: apply QuickCheck and the covered techniques to a project of your choice.

Outline

- Today we'll spend on preliminaries
  (getting OCaml working, etc.)
- Over the next months we'll gradually learn OCaml and QuickCheck through lectures and exercises
- We may have a guest lecture by Jesper Louis Andersen (uses QuickCheck in the "real world")
  (still to be determined)
- Start thinking about a project topic
  - test a library, an app, a webserver, a compiler, . . .
- Mobile Industrial Robots will give a guest lecture about an API they would like us to test

OCaml Basics

OCaml, botanically

- OCaml is a functional language (opposed to OO)
  - Functions are first class citizens (like in JavaScript, F#)
  - The core syntactic category is the expression (opposed to statements)
  - Assignments are possible, but rare
- OCaml is statically and strongly typed
  - No NullPointerExceptions, no ClassCastExceptions
  - actually no casts at all!
  - Since everything is an expression, everything has a type
- The interpreter infers types automatically
  - Variables are (mostly) declared without an explicit type

Riding the Camel

OCaml comes with a read-eval-print loop (like Python):

```
$ ocaml
        OCaml version 4.02.3

# print_endline "hello, world!";;
hello, world!
- : unit = ()
#
```

All interaction must end with two semicolons

Basic Types (1/4)

OCaml comes with a number of base types:

`int`, `char`, `bool`, `string`, `float`, ...

- Integers are 63 bits for a 64-bit OCaml (and 31 bits for a 32-bit OCaml): `-1`, `0`, `1`, `42`, `max_int`, ... all have type `int`
- `int`s come with the usual arsenal of operations:

  `+`, `-`, `*`, `/`, `mod`, `land`, `lor`, `lxor`, ...

Basic Types (1/4)

OCaml comes with a number of base types:

`int`, `char`, `bool`, `string`, `float`, ...

- Integers are 63 bits for a 64-bit OCaml (and 31 bits for a 32-bit OCaml): `-1`, `0`, `1`, `42`, `max_int`, ... all have type `int`
- `int`s come with the usual arsenal of operations:

  `+`, `-`, `*`, `/`, `mod`, `land`, `lor`, `lxor`, ...
- Both 64-bit and 32-bit integers are also available:
  - `-1L`, `0L`, `1L`, ... all have type `int64` and come with separate operations: `Int64.add`, `Int64.sub`, `Int64.div`, ...
  - `-1l`, `0l`, `1l`, ... all have type `int32` and also come with separate operations: `Int32.add`, ...

Basic Types (2/4)

OCaml comes with

- Booleans: **`true`** and **`false`** have type `bool`
  - Negation is `not`, conjunction is `&&`, and disjunction is `||`
  - The usual comparison operations also produce bools: `=`, `<>`, `<`, `<=`, `>`, `>=`, ...
- Characters: `'a'`, `'X'`, `'\n'`, `'\\'`, `'\012'`, ...
  - all have type `char`
  - One can convert back and forth with `char_of_int` and `int_of_char`

Basic Types (3/4)

OCaml comes with strings:

- `""` and `"hello, world!"` have type `string`
- String concatenation is `^`: `"SM2" ^ "-TES"`
- One can inspect and manipulate strings: `String.length`, `String.uppercase`, `String.lowercase`, ...
- And convert to and from strings: `int_of_string`, `string_of_int`, `Int64.of_string`, `Int32.to_string`, `bool_of_string`, `string_of_bool`, ...

Basic Types (4/4)

- `()` has the type `unit`
- Notice how `print_endline` returned `unit`
- `unit` serves the purpose of `void` in C and Java
- and doubles as the "empty argument (list)": `print_newline()`
- Technically (or pedantically) it is not the "empty type" since one value has `unit` type, namely `()`
- `(* Comments are enclosed in parentheses and asterisks *)`

Conditionals

Conditionals in OCaml are expressions and hence have a type:

```
if (1=2) || true then 1+3 else 42
```

As a consequence the two branches have to return something of the same type:

```
# if not false then "hello" else ();;
Error:  This expression has type unit
but an expression was expected of
type string
#
```

Intermezzo: installation and grammars

OCaml, recap

- So far we've written some basic OCaml expressions following the below grammar:

$$
\begin{array}{lr}
exp ::= value & \text{(ints, bools, chars, strings,...)} \\
\quad \mid exp + exp \mid exp - exp \mid \ldots & \text{(binary operations)} \\
\quad \mid -exp & \text{(unary minus)} \\
\quad \mid (exp) & \text{(parenthesized exps)} \\
\quad \mid id\ exp & \text{(function calls)} \\
\quad \mid \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp & \text{(conditionals)}
\end{array}
$$

Top-level let bindings

- In ML one can bind the value of an expression to a name:

  ```
  let id = exp
  ```

  For example:

  ```
  let x = 3;;
  let y = 4;;
  x + y;;
  ```

- Important note: this is not an assignment!
- An assignment has state, i.e., a little piece of memory that can (and will) change under your feet. . .

Nested let bindings

- One can also locally bind a value to a name locally within an expression:

```
let id = exp in exp
```

- Confusingly this is also expressed with the **`let`** keyword(!)

  For example:

```
let x = 3 in x * x * x
```

  gives 27 but afterwards `x` is no longer visible:

```
# x+x;;
Error:  Unbound value x
```

OCaml syntax, recap

A grammar can formally distinguish top-level **`lets`** from the nested, expression-level **`lets`**:

$$
\begin{aligned}
topdecl ::= {} & exp \\
\mid {} & \texttt{let}\ id = exp && \text{(top-level let)} \\
\\
exp ::= {} & id \\
\mid {} & value \\
\mid {} & exp + exp \mid exp - exp \mid \ldots \mid -exp \\
\mid {} & (exp) \\
\mid {} & id\ exp \\
\mid {} & \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp \\
\mid {} & \texttt{let}\ id = exp\ \texttt{in}\ exp && \text{(expr-level let)}
\end{aligned}
$$

Functions (are fun) (1/2)

Functions are written with the **`fun`** keyword:

```
fun id ... id -> exp
```

For example: `fun x -> x * x`
has the type: `int -> int`

We can bind the function value to a name:

```
let square = fun x -> x * x
```

and call it:

```
# square 4;;
- : int = 16
#
```

Functions (are fun) (2/2)

It is so common to bind a function value to a name that there is a short hand notation:

```
let funname id ... id = exp
```

For example: `let square x = x * x`

One can also locally define functions with similar short hand notation:

```
let funname id ... id = exp in exp
```

For example:

```
let quadruple n =
  let double m = m + m in
  double (double n)
```

Install OCaml and dive in

- The community homepage is `http://ocaml.org/`
- The standard OCaml distribution comes with, e.g,
  - a bytecode interpreter (`ocamlc`), a native code compiler (`ocamlopt`), a build tool (`ocamlbuild`) and a standard library:

    `http://caml.inria.fr/pub/docs/manual-ocaml/libref/`

- There are even (separately available) compilers to JavaScript (`js_of_ocaml` and `BuckleScript`)
- We'll use 'Introduction to Objective Caml' by Jason Hickey, available at:

  `http://courses.cms.caltech.edu/cs134/cs134b/book.pdf`

  We will only need the first 12 chapters (∼130 pages)

Editing OCaml code

IDE-wise, for

- Atom the OCaml support is pretty good.
- emacs I recommend tuareg-mode
- VIM: OMLet
- IntelliJ: intellij-ocaml?
- _: please share your findings

The basic modes provide syntax highlighting. More info:
`http://pl.cs.jhu.edu/pl/ocaml/index.shtml`

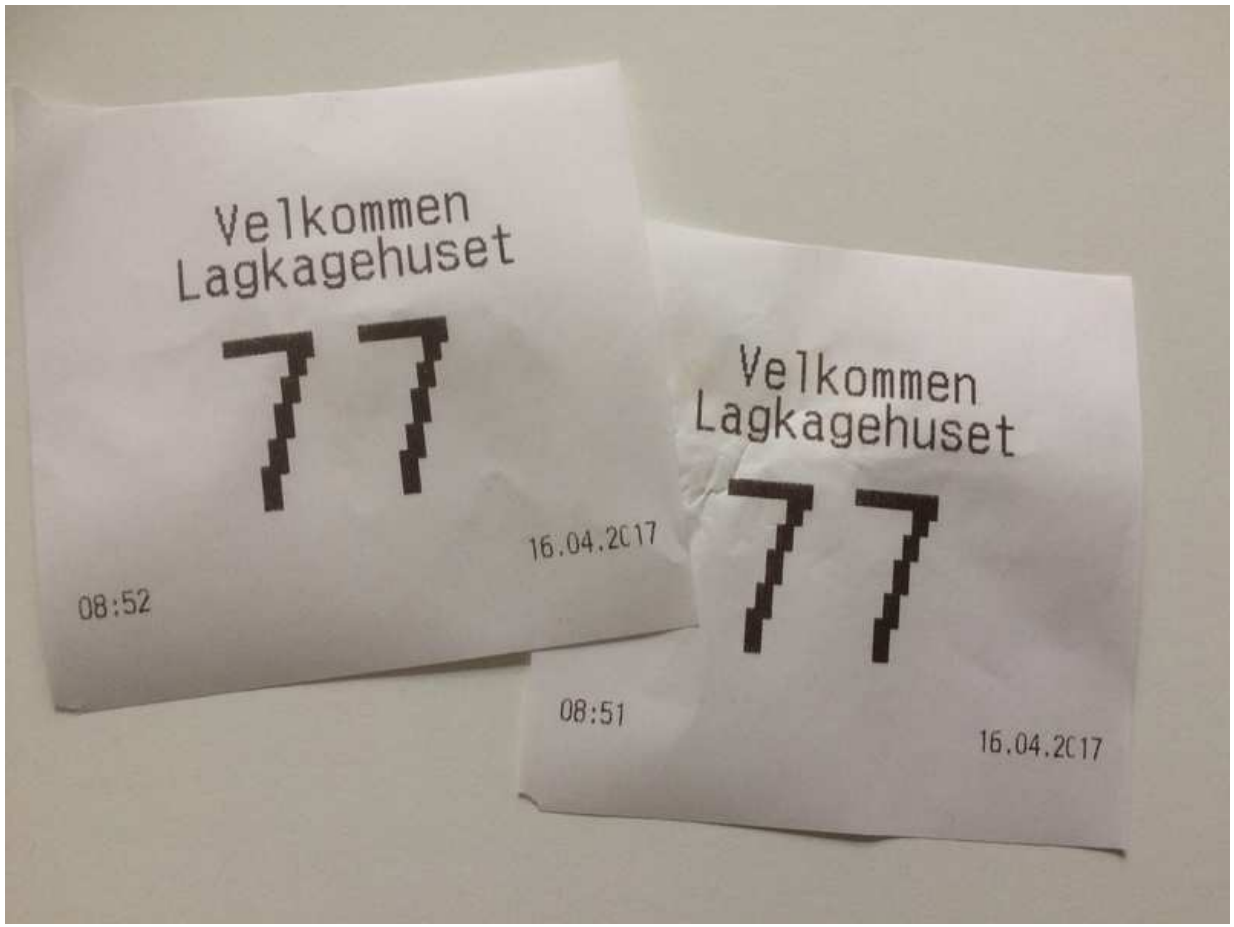
Merlin adds type feedback and context-sensitive completion for Atom/emacs/vim/...
`https://github.com/ocaml/merlin`

Property-Based Testing

Q: What do you know about testing?

Q: Why is testing important?

Example bugs



Q: How might testing have caught this error?

Example bugs

The binary search impl. in `java.util.Arrays` (and most other implementations) had a line:

```
int mid = (low + high) / 2;
```

- which overflows for sufficiently large integer values of `low` and `high`
- thus throwing an `ArrayIndexOutOfBoundsException`

Details:

https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.h

Q: How might testing have caught this error?

Example bugs

1988 - Buffer overflow in Berkeley Unix finger daemon.

"The first internet worm (the so-called Morris Worm) infects between 2,000 and 6,000 computers in less than a day by taking advantage of a buffer overflow. The specific code is a function in the standard input/output library routine called `gets()` designed to get a line of text over the network. Unfortunately, `gets()` has no provision to limit its input, and an overly large input allows the worm to take over any machine to which it can connect.

Programmers respond by attempting to stamp out the `gets()` function in working code, but they refuse to remove it from the C programming language's standard input/output library, where it remains to this day."

From

`https://www.wired.com/2005/11/historys-worst-software-bugs/`

Q: How might testing have caught this error?

Example bugs

1988–1996 - Kerberos Random Number Generator.

“The authors of the Kerberos security system neglect to properly "seed" the program’s random number generator with a truly random seed. As a result, for eight years it is possible to trivially break into any computer that relies on Kerberos for authentication. It is unknown if this bug was ever actually exploited.”

From

`https://www.wired.com/2005/11/historys-worst-software-bugs/`

Q: How might testing have caught this error?

Example bugs

January 15, 1990 - A&T Network Outage.

“A bug in a new release of the software that controls AT&T’s #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines - a message that the neighbors send out when they recover from a crash.

One day a switch in New York crashes and reboots, causing its neighboring switches to crash, then their neighbors’ neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.”

From

`https://www.wired.com/2005/11/historys-worst-software-bugs/`

Q: How might testing have caught this error?

Example bugs

1995/1996 - The Ping of Death.

“A lack of sanity checks and error handling in the IP fragmentation reassembly code makes it possible to crash a wide variety of operating systems by sending a malformed "ping" packet from anywhere on the internet. Most obviously affected are computers running Windows, which lock up and display the so-called "blue screen of death" when they receive these packets. But the attack also affects many Macintosh and Unix systems as well.”

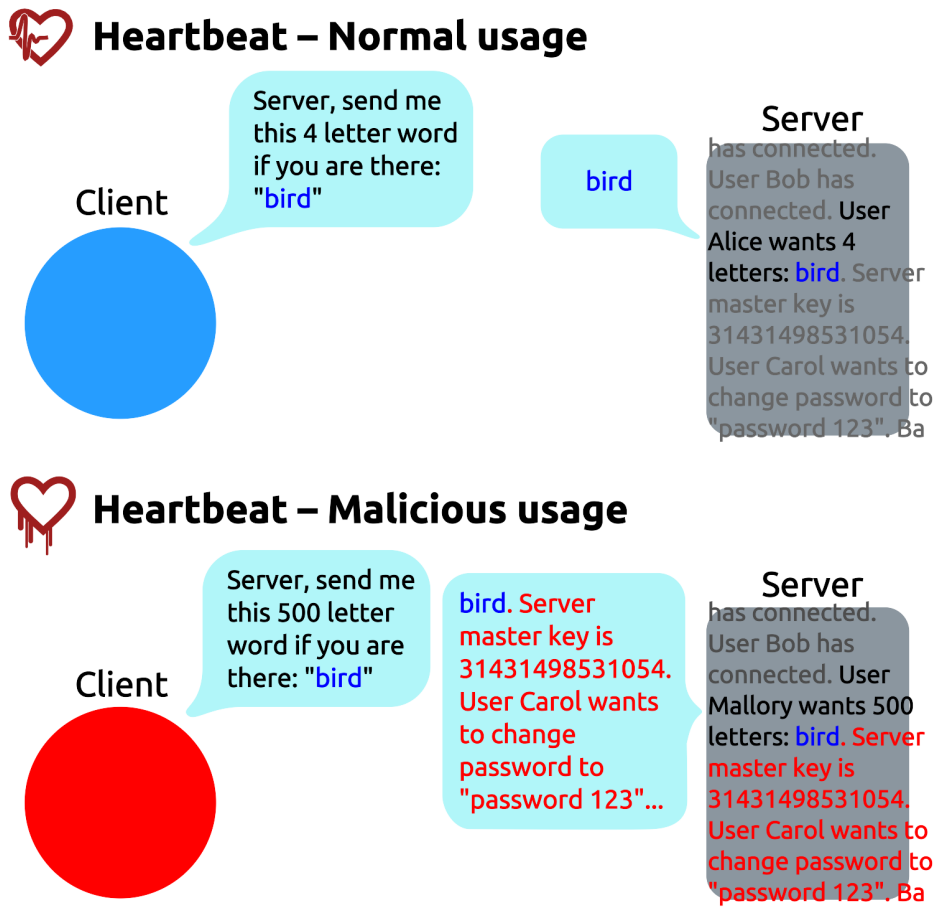
From

`https://www.wired.com/2005/11/historys-worst-software-bugs/`

Q: How might testing have caught this error?

Example bugs

2014 - Heartbleed was a security bug in the OpenSSL cryptography library



From `https://en.wikipedia.org/wiki/Heartbleed`

Q: How might testing have caught this error?

Testing

Testing (either by hand or by a hand-written test suite)

- requires discipline and
- involves repetitive tasks

Claim: Computers are much better

- at discipline and
- repetitive tasks

than humans

So let the computers aid us!

QuickCheck (1/2)

QuickCheck combines two key ideas:

- random testing (random input) and
- specifications as oracles (property-based)

For this reason it is also called

(randomized) property-based testing

It was conceived by Koen Claessen and John Hughes around 1999 (published in 2000).

Initially as a Haskell library, since then ported to >30 other languages:

`https://en.wikipedia.org/wiki/QuickCheck`

QuickCheck (2/2)

The QuickCheck approach has since grown out of academia and into industry:

John Hughes and friends formed ‘Quviq AB’ which

- produces an Erlang QuickCheck library and
- sells QuickCheck consultancy (`http://quviq.com/`)

Lots of success stories:

- Academia: algorithms, compilers, elections, . . .
- Industry: AUTOSAR/Volvo, Google’s LevelDB, Riak DB, Galois, Ericsson, Motorola, . . .

In the course we will study some of these cases.

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., for all f. `floor` f `<=` f
- a generator of arbitrary elements

  `0.17907055696997961, -237.29915004459596,`
  `111438.64440199361, ...`

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., for all f. `floor` f `<=` f
- a generator of arbitrary elements

  `0.17907055696997961, -237.29915004459596,`
  `111438.644401993617, ...`

and then have the property checked on, e.g., 100 arbitrary inputs

```
floor 0.17907055696997961 <= 0.17907055696997961,
floor -237.29915004459596 <= -237.29915004459596,
floor 111438.644401993617 <= 111438.644401993617,...
```

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., for all f. `floor` f `<=` f
- a generator of arbitrary elements

  `0.17907055696997 9616, -237.299150044595962, 111438.644401993617, ...`

and then have the property checked on, e.g., 100 arbitrary inputs

Bonus: you get to program, not really write tests :-)

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., for all f. `floor` f `<=` f
- a generator of arbitrary elements

  `0.17907055696997 9616, -237.29915004459 5962, 111438.644401993617, ...`

and then have the property checked on, e.g., 100 arbitrary inputs

Bonus: you get to program, not really write tests :-)

Risk: you may make programming errors in the generators and properties :-(

Garbage in, garbage out (also for QuickCheck)

> On two occasions I have been asked, – *"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?"* In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> – Charles Babbage, 1864

This also applies to QuickCheck tests:

generators and properties

Q: Can you think of a poor example of each?

QuickCheck in OCaml

- A number of libraries and frameworks are available for QuickCheck in OCaml

  (some more polished than others. . .)

- We will use the `QCheck` library

  `https://github.com/c-cube/qcheck/`

  Note: The API changed with the 0.5 release

- The library is also available for installation through OPAM, OCaml's package manager

QuickCheck with QCheck

A QuickCheck test in QCheck needs 2 arguments:

- a generator (of random elements)
- a property (or specification / law)

For example:

```
let mytest =
  Test.make float (fun f -> floor f <= f);;
```

where input is supplied by the builtin float generator `float` to test the floor function for the property "result of floor is less-or-equal than its argument".

We can now run it:

```
# QCheck_runner.run_tests [mytest];;
success (ran 1 tests)
```

For the rest of today

We need to get you up and running in OCaml and QCheck:

So: install OCaml, QCheck, and an editor following the instructions

Once installed:

- try the selected exercises
- read the listed chapters from Hickey's book