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 pure, 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 pure, 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, Reason, 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 smaller mandatory one?)
- **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.

Course Expectations

I expect you to make an effort and participate every week.

We build up every week so don't skip lectures.

In return, I'll do my best to help you.

Course Expectations

I expect you to make an effort and participate every week.

We build up every week so don't skip lectures.

In return, I'll do my best to help you.

From past course evaluations:

- *"... The teaching itself from Jan was very good and he has always been able to help if needed."* (2019)
- *"... Overall i believe that the 5 ECTS is filled out great, more than enough content i think (in a good way). He takes time to work with you if you have problems, and does so, at the expense of his own free time. The course has been very interesting. "* (2019)
- *"The project consultations worked wonders"* (2020)
- *"Jan's method of online lectures worked well."* (2020)

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
- Guest lecture, April 27: TBA
- Start thinking about a project topic
  - test a library, an app, a webserver, . . .
  - Orbit `https://orbit.online/` will offer 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 4.07.1

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

Loop interaction must end with two semicolons

Riding the Camel with `utop`

`utop` is an enhanced (better) read-eval-print:

```
$ utop
                Welcome to utop version 2.2.0 (using OCaml version 4.07.1)!

 [lots of output omitted]

Type #utop_help for help about using utop.

-( 15:48:47 )-< command 0 >-------------------------------{ counter: 0 }-
utop # print_endline "hej, verden!";;
hej, verden!
- : unit = ()
-( 15:48:47 )-< command 1 >-------------------------------{ counter: 0 }-
utop #
```

`utop` supports

- arrows and Ctrl-a / Ctrl-e for navigation,
- tab for completion,
- and more. . .

Intermezzo: Installation

OCaml resources

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

    `http://caml.inria.fr/pub/docs/manual-ocaml/libref/`
- There is even a (separately available) compiler to JavaScript (`js_of_ocaml`)
- 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, I recommend Visual Studio Code with the **'OCaml and Reason IDE'** extension.

This setup is easy to install and gives you both syntax highlighting and type feedback (via `merlin`).

Merlin is a "language server" providing type feedback and context-sensitive completion for a range of IDEs and editors: `https://github.com/ocaml/merlin`

There are other IDE/editor options:
Atom, emacs w/tuareg-mode, VIM w/OMLet, . . .
Unless you are familiar with juggling the command line and environment variables I suggest you stick to VS Code. . .

Package managing and build tools

The package manager is called `opam`
(think `npm`, but for OCaml libraries)

Like `npm`, `opam` offers a heap of libraries for different purposes: `https://opam.ocaml.org/`

OCaml has a build tool called `ocamlbuild`, which will do for our purposes.

There's another, increasingly popular build tool called `dune` (originally called `jbuilder`), see `https://dune.build/`

Install away!

Install OCaml and setup VS Code as described in the installation guide.

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)

Booleans: **`true`** and **`false`** have type `bool`

- Negation is `not`, conjunction is `&&`, and disjunction is `||`
- The usual comparison operations also produce bools: `=`, `<>`, `<`, `<=`, `>`, `>=`, ...

Basic Types (2/4)

Booleans: **`true`** and **`false`** have type `bool`

- Negation is `not`, conjunction is `&&`, and disjunction is `||`
- The usual comparison operations also produce bools: `=`, `<>`, `<`, `<=`, `>`, `>=`, `...`

Floats: `3.14`, `-1.`, `max_float`, `nan`, `...`

- They have type `float` and come with their own operations `+.`, `-.`, `sqrt`, `floor`, `...`

Basic Types (2/4)

Booleans: **`true`** and **`false`** have type `bool`

- Negation is `not`, conjunction is `&&`, and disjunction is `||`
- The usual comparison operations also produce bools: `=`, `<>`, `<`, `<=`, `>`, `>=`, `...`

Floats: `3.14`, `-1.`, `max_float`, `nan`, `...`

- They have type `float` and come with their own operations `+.`, `-.`, `sqrt`, `floor`, `...`

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_ascii`, `String.lowercase_ascii`, ...
- 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
#
```

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
#
```

This example also illustrates type inference at work

Intermezzo: Grammars

OCaml, recap

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

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

$$value ::= \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots$$

(bools, ints, chars, strings,...)

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 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`

Function types are written with arrow: `t -> ... -> t`
For example the above function has 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)
```

Exercise: Solve exercise 5

5. Implement the following three functions:

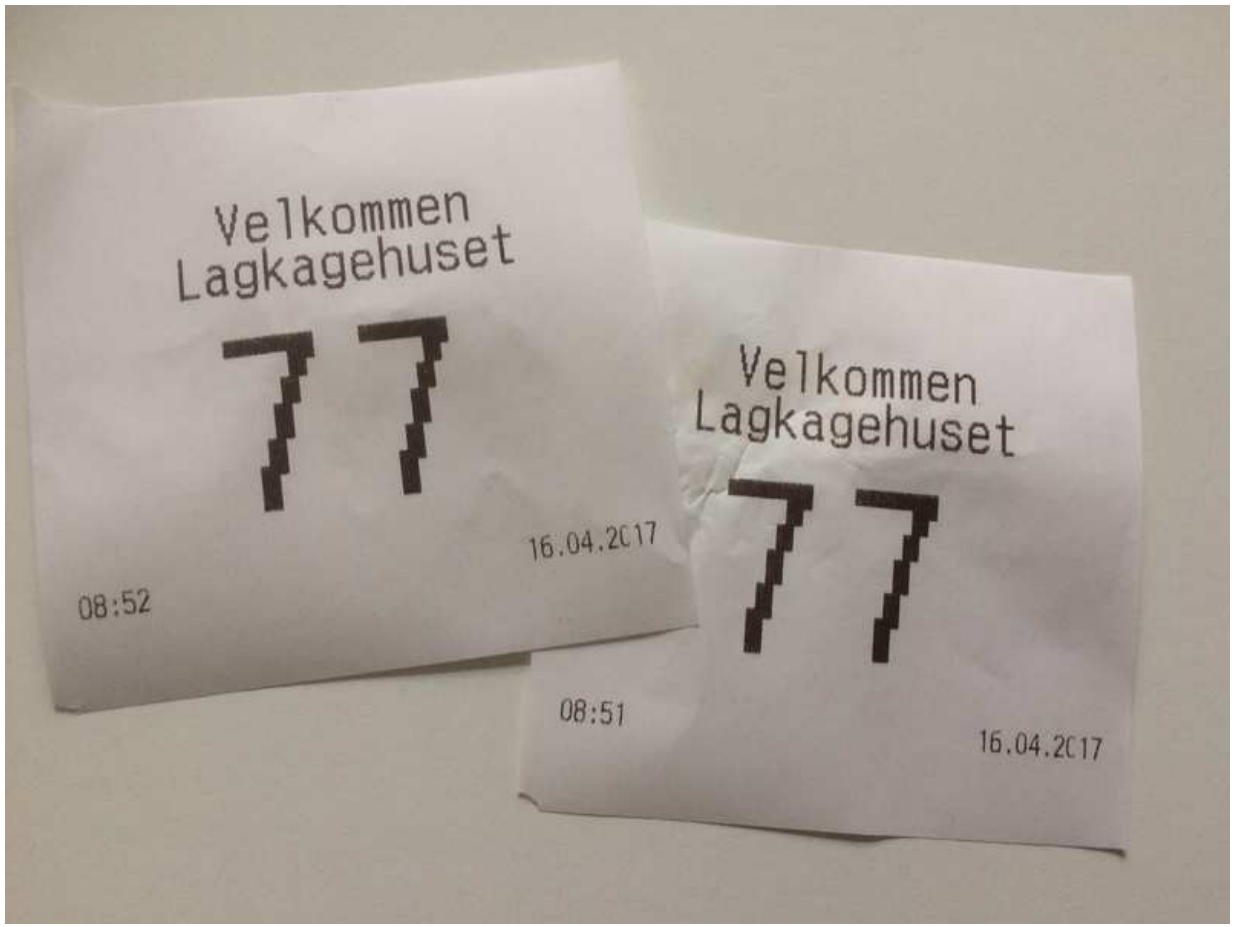
```
  cube : int -> int

    the function should return the cube of its argument,
    so that cube 2 returns 8, cube 3 returns 27, ...

  is_even : int -> bool

    is_even returns a Boolean indicating whether the argument
    is divisible by 2, e.g., is_even 2 returns true, is_even 41
    returns false.

  quadroot : float -> float

    rather than the square root, quadroot should return the
    fourth root of its argument, i.e., a number which raised
    to the fourth power gives the argument. For example:
    quadroot 16. returns 2.,
    quadroot 4.0 returns 1.41421...
```

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

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

1997 - USS Yorktown division-by-zero error

“A system failure on the USS Yorktown last September temporarily paralyzed the cruiser, leaving it stalled in port for the remainder of a weekend. [. . .] The source of the problem on the Yorktown was that bad data was fed into an application running on one of the 16 computers on the LAN. The data contained a zero where it shouldn’t have, and when the software attempted to divide by zero, a buffer overrun occurred - crashing the entire network and causing the ship to lose control of its propulsion system. ”

From `https://www.wired.com/1998/07/sunk-by-windows-nt/`

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`

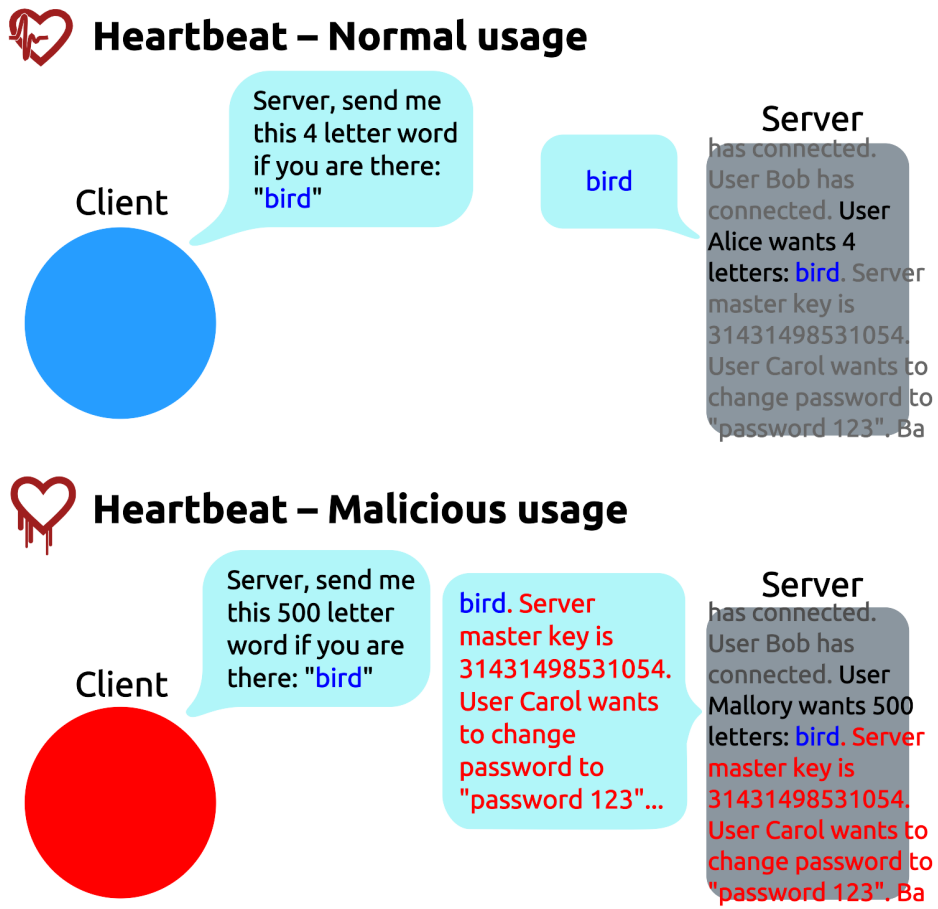
Details:

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

Q: How might testing have caught this error?

Example bugs

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?

Example bugs

2019 - Boeing screen blanking

“Boeing’s 737 Next Generation airliners have been struck by a peculiar software flaw that blanks the airliners’ cockpit screens if pilots dare attempt a westwards landing at specific airports. [. . .] Seven runways, of which five are in the US, and two in South America - in Colombia and Guyana respectively - trigger the bug. Instrument approach procedures guide pilots to safe landings in all weather conditions regardless of visibility. [. . .] "All six display units (DUs) blanked with a selected instrument approach to a runway with a 270-degree true heading, and all six DUs stayed blank until a different runway was selected," noted the FAA’s airworthiness directive, summarising three incidents that occurred on scheduled 737 flights to Barrow, Alaska, in 2019. ”

From

`https://www.theregister.co.uk/2020/01/08/boeing_737_ng_cockpit_screen_blank_bug`

SDU Q: How might testing have caught this error?

Testing (1/3)

As you know (and as some of these stories illustrate), it is custom to test a system's boundary cases (corner cases): `0`, `max_int`, `""`, `null`, `...`

It is also common, that an operation

- assumes certain things about the input – a pre-condition (e.g., a non-negative integer representing a length, a non-null reference, ...) – such assumptions may be expressed with `assert`.
- lets a caller distinguish a successful execution from a failing one – a post-condition (e.g., non-null signals successful file opening or memory allocation, ...)

A good testsuite has to take these into account

Testing (2/3)

As the previous bug stories illustrate

- some errors are associated with normal usage:
  an expected input yields an unexpected output or behavior, aka. positive testing
- some errors are associated with “misuse”:
  an unexpected input yields an unexpected output or behavior, aka. negative testing

A testsuite should (attempt to) validate both of these.

Testing (3/3)

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: Volvo, Google’s LevelDB, Riak DB, Galois, Ericsson, Motorola, Spotify, Uber, Stripe, . . .

In the course we will study some of these cases.

The Essence of QuickCheck

With QuickCheck one expresses a family of test cases at a higher level of abstraction.

Tests are described by

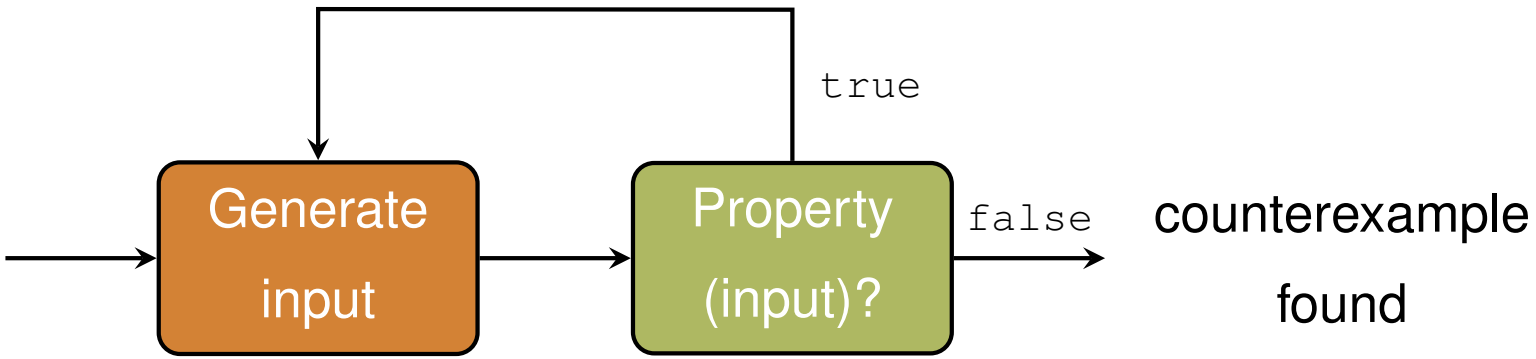
- a generator (delivering random input)
- a property (Boolean-valued function)

The Essence of QuickCheck

With QuickCheck one expresses a family of test cases at a higher level of abstraction.

Tests are described by

- a generator (delivering random input)
- a property (Boolean-valued function)

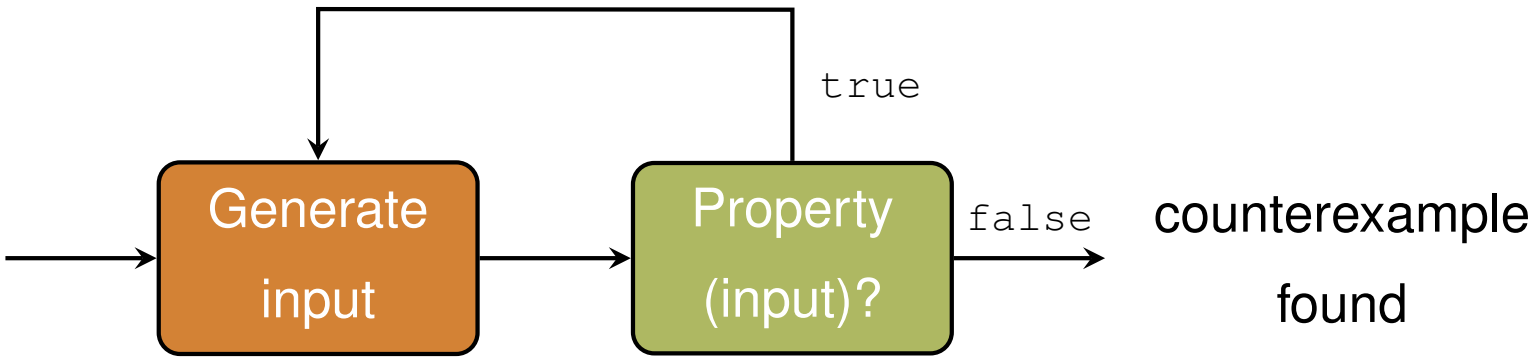


The Essence of QuickCheck

With QuickCheck one expresses a family of test cases at a higher level of abstraction.

Tests are described by

- a generator (delivering random input)
- a property (Boolean-valued function)



Each run is driven by a random seed. Given the seed for a problematic run we can recreate the problem.

Generators and Properties: An Example

Suppose we want to test the builtin `floor` function.

What property should `floor` have?

Generators and Properties: An Example

Suppose we want to test the builtin `floor` function.

What property should `floor` have?

How about `floor` $f \leq f$ for any float f?

Generators and Properties: An Example

Suppose we want to test the builtin `floor` function.

What property should `floor` have?

How about `floor` $f \leq f$ for any float f?

There's builtin generators for base types such as `float`.

A complete test:

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

Generators and Properties: An Example

Suppose we want to test the builtin `floor` function.

What property should `floor` have?

How about `floor` $f \leq f$ for any float f?

There's builtin generators for base types such as `float`.

A complete test:

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

Underneath the hood the property is tested on 100 arbitrary inputs:

```
floor 0.17907055696979616 <= 0.17907055696979616,
floor -237.299150044595962 <= -237.299150044595962,
floor 111438.644401993617 <= 111438.644401993617,...
```

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 strengths and risks

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

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

Q: To test boundary cases, what must the QuickCheck tester do?

Q: In terms of pre- and post conditions, what should the QuickCheck tester do?

Q: In terms of positive and negative testing, what should the QuickCheck tester do?

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 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: finish installing OCaml, QCheck, and VS Code if you haven't done so

Once installed:

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