02913 Advanced Analysis Techniques

QuickCheck, the 2017 edition

Jan Midtgaard

DTU Compute

DTU

Introduction (1/2)

In the next 3 weeks, you will learn

- the principles and practice of QuickCheck
- the OCaml programming language

DTU

Introduction (1/2)

In the next 3 weeks, you will learn

- the principles and practice of QuickCheck
- the OCaml programming language

Why OCaml?

- QuickCheck is based on testing properties
- These are most easily expressed in a functional language such as OCaml, which has 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

DTU

Introduction (2/2)

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#, Haskell, Standard ML, . . .
- they may come in handy next time you program, e.g., callbacks in JavaScript.

DTU

Practicalities

The course will consist of

- lectures (mornings)
- exercises (afternoons)
- a course project
- a project report
- a project presentation (end of week 3)

You'll receive a combined grade for the report+presentation.

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

DTU

Outline

- Today we'll spend on preliminaries (getting OCaml working, etc)
- Over the next days we'll gradually learn QuickCheck (and OCaml) through lectures and exercises
- Guest lecture (week 2): Jesper Louis Andersen (does QuickCheck in the "real world")
- Start thinking about a project topic
  - test an app,
  - test a webserver,
  - test a compiler,
  - ...

DTU

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

DTU

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`, ...

DTU

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 `^`: `"029" ^ "13"`
- 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`, ...

DTU

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 *)
```

DTU

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
#
```

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
#
```

DTU

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)
```

Recursive functions

Recursive functions are explicitly marked as such with the **`rec`** keyword:

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

For example:

```
let rec fac n =
  if n = 0
  then 1
  else n * fac (n - 1)
```

To which OCaml responds:

```
val fac : int -> int = <fun>
```

DTU

Mutually recursive functions

Recursive functions that call each other should be declared simultaneously with **`and`**.

For example:

```
let rec is_even n =
  if n = 0
  then true
  else is_odd (n - 1)
and is_odd n =
  if n = 0
  then false
  else is_even (n - 1)
```

to which OCaml responds:

```
val is_even : int -> bool = <fun>
val is_odd : int -> bool = <fun>
```

Pattern matching (1/5)

One typically destructs data using pattern matching:

```
match exp with
  | pattern -> exp
  | pattern -> exp
  | ...
```

For example:

```
let bool_to_string b = match b with
  | true  -> "true"
  | false -> "false"
```

Pattern matching (2/5)

OCaml's pattern matching compiler helps ensure that you don't miss a case:

```
let is_valid_bool s = match s with
  | "true" -> true
  | "false" -> true
```

Pattern matching (2/5)

OCaml's pattern matching compiler helps ensure that you don't miss a case:

```
let is_valid_bool s = match s with
  | "true" -> true
  | "false" -> true
Warning 8:  this pattern-matching is not
exhaustive.
Here is an example of a value that is not
matched:
""
val is_valid_bool :  string -> bool = <fun>
```

Pattern matching (3/5)

The wildcard pattern can be used to catch “default” behavior:

```
let is_valid_bool s = match s with
  | "true"  -> true
  | "false" -> true
  | _       -> false
```

which will satisfy OCaml:

```
val is_valid_bool : string -> bool = <fun>
```

The exhaustiveness check is your friend.

Don’t use wildcards everywhere just to “make it shut up”.

Pattern matching (4/5)

Beware that the pattern order now matters:

```
let is_valid_bool s = match s with
  | _         -> false
  | "true"  -> true
  | "false" -> true
```

Pattern matching (4/5)

Beware that the pattern order now matters:

```
let is_valid_bool s = match s with
  | _         -> false
  | "true"  -> true
  | "false" -> true
```

```
Warning 11:  this match case is unused.
Warning 11:  this match case is unused.
val is_valid_bool :  string -> bool = <fun>
```

Pattern matching (5/5)

Patterns can also bind variables which will be visible within the pattern's right-hand-side:

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
```

to which OCaml (again) responds:

```
val fac : int -> int = <fun>
```

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 is even a (separately available) compiler to JavaScript (`js_of_ocaml`)
- I recommend '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)

DTU

Editing OCaml code

IDE-wise, for

- emacs I recommend tuareg-mode
- Eclipse people recommend: OCaIDE
  `http://www.algo-prog.info/ocaide/`
  `http://pl.cs.jhu.edu/pl/ocaml/ocaide.shtml`
- IntelliJ: intellij-ocaml?
- VIM: OMLet
- _: please share your findings

These modes provide syntax highlighting.

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

QuickCheck

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 most other languages

DTU

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.
- see `http://quviq.com/`

It has since been used, e.g., to test compatibility of AUTOSAR components for Volvo.

Both Hughes and our guest lecturer Jesper Louis Andersen uses QuickCheck to test distributive systems

DTU

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., $\forall n.\ (\texttt{fac}\ n)\ \texttt{mod}\ n = 0$
- a generator of arbitrary elements

  `0, 42, -2, 7234, -1000000, ...`

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., $\forall n.\ (\texttt{fac}\ n)\ \texttt{mod}\ n = 0$
- a generator of arbitrary elements

  `0, 42, -2, 7234, -1000000, ...`

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

QuickCheck, briefly

QuickCheck builds on the idea of expressing

a family of tests by

- a property of interest, e.g., $\forall n.\ (\texttt{fac}\ n)\ \texttt{mod}\ n = 0$
- a generator of arbitrary elements

  `0, 42, -2, 7234, -1000000, ...`

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., $\forall n.\ (\texttt{fac}\ n)\ \texttt{mod}\ n = 0$
- a generator of arbitrary elements

  `0, 42, -2, 7234, -1000000, ...`

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 a programming error in the property :-(

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. . .)

- This year we'll use the `QCheck` library

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

  Beware: The API changed with the 0.5 release

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

DTU

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 suggested chapters from Hickey's book

DTU