02913 Advanced Analysis Techniques

QuickCheck, Day 2

Jan Midtgaard

DTU Compute

Yesterday’s exercises

Outline

OCaml recap

More OCaml

QuickChecking with QCheck

Testing for properties

Classification

OCaml recap

OCaml recap

- By now you've installed OCaml and written/sent your first expression to the toplevel
- Yesterday we wrote some basic OCaml expressions following the below grammar:

$$
\begin{aligned}
topdecl ::= {} & exp \\
\mid {} & \texttt{let}\ id\ id \ldots id = exp \\
exp ::= {} & id \\
\mid {} & value \\
\mid {} & exp + exp \mid exp - exp \mid \ldots \mid - exp \\
\mid {} & \texttt{fun}\ id \ldots id\ \texttt{->}\ exp \\
\mid {} & exp\ exp \ldots exp \\
\mid {} & \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp \\
\mid {} & \texttt{(}exp\texttt{)} \\
\mid {} & \texttt{let}\ id\ id \ldots id = exp\ \texttt{in}\ exp \\
\mid {} & \texttt{match}\ exp\ \texttt{with} \mid pat\ \texttt{->}\ exp \mid \ldots \mid pat\ \texttt{->}\ exp
\end{aligned}
$$

DTU

Read-eval-print loop vs `.ml` files

In the screen casts you saw

- □ an example of writing a QCheck QuickCheck test directly in the REPL loop

  (we finish these *topdecl*s with `;;`)

- □ the same example written to an `.ml` file and compiled with `ocamlbuild` (here `;;` is not required)

  However: toplevel expressions in a file should be separated by `;;` to distinguish them from calls:

$$definition ::= \texttt{\textbf{let}}\ id\ id \ldots id = exp$$

$$topdecls ::= (exp \mid definition)\ (\ \texttt{;;}\ exp \mid\ [\texttt{;;}]\ definition)^{*}$$

DTU

More OCaml

Tuples

Tuples are one way to combine types to build new ones:

```
type point3d = int * int * int
```

which declares point3d as a short hand for `int` triples

OCaml will infer tuple types (they don't need to be declared):

```
# let mypair = (1,2);;
val mypair : int * int = (1, 2)
```

One can project data from pairs with `fst` and `snd`:

```
# snd mypair;;
- : int = 2
```

Tuple matching

One can also pattern match on tuple types using **`let`**:

```
let distance_from_origo p =
  let (x,y) = p in
  let sqr_dist = (x * x) + (y * y) in
  sqrt (float_of_int sqr_dist)
```

for which OCaml infers the type:

```
val distance_from_origo : int * int -> float = <fun>
```

Alternatively one can pattern match directly in the function header:

```
let distance_from_origo' (x,y) =
  let sqr_dist = (x * x) + (y * y) in
  sqrt (float_of_int sqr_dist)
```

The `simple.ml` file from the screen cast also does this

Lists

Lists are created inductively from the empty list `[]` and the cons operator `::`

```
# let mylist = 1::2::3::[];;
val mylist : int list = [1; 2; 3]
```

In Java we would (probably) write this as
`List<Integer>`

As a short hand one can also write list literals with square brackets and semicolon as element separator:

```
# let mylist' = [0;1;2;3];;
val mylist' : int list = [0; 1; 2; 3]
```

One can concatenate lists with `@`:

```
# mylist@mylist;;
- : int list = [1; 2; 3; 1; 2; 3]
```

Lists, polymorphically

We can now write structurally recursive functions over lists:

```
let rec length l = match l with
  | [] -> 0
  | elem::elems -> 1 + length elems
```

For which OCaml will infer the polymorphic type:

```
val length : 'a list -> int = <fun>
```

The corresponding generic Java method would accept a `List<X>` and return a Java `int`

Labeled arguments

OCaml supports labeled (named) arguments

The syntax for the receiver (the formal parameters) is:

```
let id ~label:pattern ... ~label:pattern = exp
```

Example: `let mymod ~num:n ~modulus:m = n mod m`

Labeled arguments

OCaml supports labeled (named) arguments

The syntax for the receiver (the formal parameters) is:

```
let id ~label:pattern ... ~label:pattern = exp
```

Example: `let mymod ~num:n ~modulus:m = n mod m`

A short-hand is available for arguments that don't need pattern matching (labels and patterns agree):

```
let id ~label ... ~label = exp
```

Example: `let mymod ~num ~modulus = num mod modulus`

Labeled arguments

OCaml supports labeled (named) arguments

The syntax for the receiver (the formal parameters) is:

```
let id ~label:pattern ... ~label:pattern = exp
```

Example: `let mymod ~num:n ~modulus:m = n mod m`

A short-hand is available for arguments that don't need pattern matching (labels and patterns agree):

```
let id ~label ... ~label = exp
```

Example: `let mymod ~num ~modulus = num mod modulus`

Functions are also invoked with labels
`id ~label ... ~label` in no particular order:

```
# mymod ~modulus:4 ~num:10;;
- : int = 2
```

Optional arguments

In addition OCaml supports optional arguments:

arguments which may or may not be supplied.

```
let id ?(label = exp) ... ?(label = exp) = exp
```

When absent the receiver assumes a *default value*

For example:

```
let distance ?(src = (0,0)) (tx,ty) =
  let (sx,sy) = src in
  let xdiff = tx - sx in
  let ydiff = ty - sy in
  let sqr_dist = (xdiff*xdiff) + (ydiff*ydiff) in
  sqrt (float_of_int sqr_dist)
```

which we can invoke as a labeled argument:

```
# distance ~src:(1,1) (4,5);;
- : float = 5.
```

The standard library

- OCaml includes a decent standard library:

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

- All bindings in the module Pervasives are available in the top-level.
- Many of the functions we have covered (and more) come from Pervasives - so have a look :-)
- Note: There are at least 3 other competing "standard libraries". We'll stick to the one from the standard distribution

QuickChecking with QCheck

From one to many tests

Yesterday we saw how to write one test:

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

Most often we want to check more than one thing

We can do so by writing individual tests:

```
let floor_test =
  Test.make float (fun f -> floor f <= f)
let ceil_test =
  Test.make float (fun f -> f <= ceil f)
```

and running them all:

```
let _ = QCheck_runner.run_tests
          [floor_test;
           ceil_test]
```

DTU

Naming tests and increasing test iterations

Both `Test.make` and `QCheck_runner.run_tests` support a range of labeled, optional arguments. In particular:

- `~name:str` sets the title of a test to the string `str`
- `~count:n` sets the number of test runs to `n`
- ...

while option `~verbose:true` makes the test run a bit more informative. For example:

```
# let floor_test = Test.make ~name:"floor␣test" ~count:300
                             float (fun f -> floor f <= f) in
  QCheck_runner.run_tests ~verbose:true [floor_test];;
law floor test: 300 relevant cases (300 total)
success (ran 1 tests)
- : int = 0
#
```

Running QCheck from the command line

QCheck provides `QCheck_runner.run_tests_main` as an alternative way to drive a test suite:

```
let floor_test =
  Test.make float (fun f -> floor f <= f)
let ceil_test =
  Test.make float (fun f -> f <= ceil f)
;; (* important to distinguish the last call
      from additional arguments to Test.make *)
QCheck_runner.run_tests_main [floor_test; ceil_test]
```

By default this runs non-verbose, but the command-line argument `--verbose` has the same effect as passing `~verbose:true` to `QCheck_runner.run_tests`

In addition it accepts `--seed` for the randomization

A QCheck note on iteration count

In QCheck the `~count:n` parameter is bounded upwards by the option `~max_gen:m` which may be a bit surprising:

```
# let floor_test = Test.make ~count:10000 ~max_gen:1000
                               float (fun f -> floor f <= f) in
  QCheck_runner.run_tests ~verbose:true [floor_test];;
law <test>: 1000 relevant cases (1000 total)
success (ran 1 tests)
```

If specified, it is a good idea to supply a `~max_gen` option greater than the `~count` option.

The default value for the optional parameter `~max_gen` is the value of `~count` + 200

The default value for `~count` is 100

Testing properties with preconditions (1/3)

In QCheck with ==> we can also express properties involving a precondition:

```
let is_even i = (i mod 2 = 0)
let is_odd i = (i mod 2 = 1)
let succ_test =
  Test.make ~name:"succ␣test"
    pos_int (fun i -> (is_even i) ==> (is_odd (succ i)))
```

Not all generated input will satisfy the precondition:

```
law succ test: 100 relevant cases (206 total)
```

Alternatively we can express the implication via the well-known encoding $[p \implies q] \iff [\neg p \vee q]$ but doing so loses track of failed preconditions:

```
law succ test': 100 relevant cases (100 total)
```

Q: does this lead to fewer or more tests of `succ`?

Testing properties with preconditions (2/3)

In using `==>` we need to generate more input for enough to satisfy the precondition.

For this reason the default `max_gen` is 300 for the default `count` of 100 (a factor 3).

Setting `max_gen` to, e.g., 200 will limit the number of tests further:

```
law succ test: 97 relevant cases (200 total)
```

When generation is expensive you may want to limit it

DTU

Testing properties with preconditions (3/3)

Be careful that `==>` evaluates its arguments eagerly

As a consequence side-effects on the right-hand-side of `==>` are not guarded by the left-hand-side

For example:

```
Test.make ~name:"div␣test"
  small_int
  (fun i -> (i <> 0) ==> (42 / i >= 0))
```

will thus (surprisingly) fail:

```
test `div test` raised exception `Division_by_zero` on `0`
```

Note: this is not a listed as a failed property but as an internal failure

Testing for properties

Properties and generators

- We've seen how to write properties as Boolean valued functions and
- implication properties using QCheck's builtin `==>`
- We've also seen some builtin generators
  - `float`
  - `pos_int`, `small_int`
  - ...
- There are many more (see the API):

  `http://c-cube.github.io/qcheck/0.5/`

DTU

Which properties?

So far, we've seen examples of testing immediate properties of functions (`floor`, `succ`, ...)

Admittedly, these properties are not always easy to come up with :-/

Sometimes we are interested in testing agreement between two implementations:

- an initial version vs.
- a revised/optimized version

For example: a data structure with poor and improved O-bounds on time/space complexity

Testing pairs (1/2)

Suppose we write a recursive version of multiplication by repeated shifting:

```
let rec mymult n m = match n,m with
  | 0,_ -> 0
  | _,0 -> 0
  | _,_ ->
    let tmp = mymult (n lsr 1) m in
    if n land 1 = 0
    then (tmp lsl 1)
    else (tmp lsl 1) + m
```

Hopefully this version agrees with the builtin $*$:

$$\forall n, m.\ \texttt{mymult}\ n\ m = n * m$$

To test it, we need to generate pairs of integers

Testing pairs (2/2)

We can do so using

```
pair : 'a arbitrary -> 'b arbitrary -> ('a * 'b) arbitrary
```

which forms a pair generator out of a pair of generators (read `'a arbitrary` as "generator of `'a`s")

With `pair` in hand the test is straightforward:

```
Test.make ~name:"mymult,*␣agreement"
  (pair int int) (fun (n,m) -> mymult n m = n * m)
```

...and the two operations seems to agree:

```
law mymult,* agreement: 100 relevant cases (100 total)
```

Testing lists: type parameters (1/3)

`List.rev` has type `'a list -> 'a list` (for any `'a`). Suppose we want to test 3 properties of it:

$$\forall x.\ \texttt{List.rev}\ [x] = [x]$$

$$\forall xs.\ \texttt{List.rev}(\texttt{List.rev}\ xs) = xs$$

$$\forall xs, ys.\ \texttt{List.rev}(xs @ ys) = (\texttt{List.rev}\ ys) @ (\texttt{List.rev}\ xs)$$

We have to test it for a concrete type parameter, e.g., `int`.

The first property is now straightforward to write:

```
let rev_sgl_test =
  Test.make ~name:"rev␣single"
    int (fun x -> List.rev [x] = [x])
```

Testing lists: generators (2/3)

We need to generate arbitrary lists to test the second property $\forall xs.\ \texttt{List.rev}(\texttt{List.rev}\ xs) = xs$.

We can write one using a builtin generator:

```
list : 'a arbitrary -> 'a list arbitrary
```

where the parameter generates the elements

The second property can now be tested as follows:

```
let rev_twice_test =
  Test.make ~name:"rev␣twice"
    (list int)
    (fun xs -> List.rev (List.rev xs) = xs)
```

Testing lists: generating pairs/tuples (3/3)

To test the third property

$$\forall xs, ys.\ \texttt{List.rev}(xs@ys) = (\texttt{List.rev}\ ys)@(\texttt{List.rev}\ xs)$$

we need to generate arbitrary pairs of lists.

Again we do so using `pair`:

```
let rev_concat_test =
  Test.make ~name:"rev␣concat"
    (pair (list int) (list int))
    (fun (xs,ys) ->
       List.rev (xs @ ys)
         = (List.rev ys) @ (List.rev xs))
```

Similarly `triple` can form triple generators, ...

Classification

Classification – Why?

QCheck lets us check a property across many inputs

How can we be sure that these input are non-trivial, e.g., that they are not limited to a narrow corner of the input space?

Classifiers lets us inspect the generated inputs

In QCheck we can classify elements by string coercion:

Concretely this is implemented as a transformer of generators:

```
set_collect : ('a -> string) -> 'a arbitrary -> 'a arbitrary
```

where the first parameter is the classifier and the second parameter is the generator we want to instrument

Classifying generated numbers (1/2)

Suppose we want to observe the inputs to `mymult`:

```
let sign n = if n=0 then "zero" else
             if n>0 then "pos" else "neg" in
let pair_gen =
  set_collect
    (fun (n,m) -> sign n ^ ",␣" ^ sign m)
    (pair int int) in
Test.make ~name:"mymult,*␣agreement"
  pair_gen (fun (n,m) -> mymult n m = n * m)
```

which gives us

```
law mymult,* agreement: 100 relevant cases (100 total)
  neg, neg: 23 cases
  pos, pos: 24 cases
  neg, pos: 27 cases
  pos, neg: 26 cases
```

Classifying generated numbers (2/2)

Suppose we want to observe the distribution more carefully we can write a digit-counting classifier:

```
let digits n =
  let n = if n<0 then (-n) else n in
  string_of_float (ceil (log10 (float_of_int n))) in
let pair_gen =
  set_collect
    (fun (n,m) -> digits n ^ ",␣" ^ digits m)
    (pair int int) in
Test.make ~name:"mymult,*␣agreement"
  pair_gen (fun (n,m) -> mymult n m = n * m)
```

which suggests that the builtin generator prefers big ints:

```
law mymult,* agreement: 100 relevant cases (100 total)
  17., 19.: 2 cases
  19., 18.: 16 cases
  18., 19.: 15 cases
  19., 19.: 60 cases
  18., 18.: 7 cases
```

Classifying generated numbers (2/2)

Suppose we want to observe the distribution more carefully we can write a digit-counting classifier:

```
let digits n =
  let n = if n<0 then (-n) else n in
  string_of_float (ceil (log10 (float_of_int n))) in
let pair_gen =
  set_collect
    (fun (n,m) -> digits n ^ ",␣" ^ digits m)
    (pair int int) in
Test.make ~name:"mymult,*␣agreement"
  pair_gen (fun (n,m) -> mymult n m = n * m)
```

which suggests that the builtin generator prefers big ints:

```
law mymult,* agreement: 100 relevant cases (100 total)
  17., 19.: 2 cases
  19., 18.: 16 cases
  18., 19.: 15 cases
  19., 19.: 60 cases
  18., 18.: 7 cases
```

Q: does this appear to be a uniform distribution?

Classifying lists

We can classify generated lists, e.g., on their length:

```
let list_gen =
  set_collect
    (fun xs -> "len:␣" ^ string_of_int (List.length xs))
    (list (int_range 0 100)) in
  Test.make ~name:"rev␣twice"
    list_gen (fun xs -> List.rev (List.rev xs) = xs)
```

which gives rise to an output like:

```
law rev twice: 100 relevant cases (100 total)
  len: 91: 2 cases
  len: 34: 1 cases
  len: 3: 7 cases
  len: 817: 1 cases
  len: 32: 1 cases
  len: 76: 1 cases
  len: 6: 3 cases
  len: 61: 2 cases
  len: 27: 1 cases
  len: 17: 1 cases
  len: 707: 2 cases
```

Classifying lists

We can classify generated lists, e.g., on their length:

```
let list_gen =
  set_collect
    (fun xs -> "len:␣" ^ string_of_int (List.length xs))
    (list (int_range 0 100)) in
  Test.make ~name:"rev␣twice"
    list_gen (fun xs -> List.rev (List.rev xs) = xs)
```

which gives rise to an output like:

```
law rev twice: 100 relevant cases (100 total)
  len: 91: 2 cases
  len: 34: 1 cases
  len: 3: 7 cases
  len: 817: 1 cases
  len: 32: 1 cases
  len: 76: 1 cases
  len: 6: 3 cases
  len: 61: 2 cases
  len: 27: 1 cases
  len: 17: 1 cases
  len: 707: 2 cases
```

Take away:

a mixed distribution

Summary

We can

- write general properties (in QCheck)
  - as Boolean-valued functions
  - with preconditions using `==>`
- formulate generators
  - based on builtin ones `int`, `pos_int`, `float`,
  - for tuples with `pair`, `triple`,...
  - for lists with `list`
- observe our tests with classifiers