SM2-TES: Functional Programming and Property-Based Testing, Day 2

Jan Midtgaard

MMMI, SDU

SDU

Last time's exercises

Outline

OCaml recap

More OCaml

QuickChecking with QCheck

Testing for properties

OCaml recap

OCaml recap

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

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

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)

Note: separate toplevel expressions in a file by `;;` to distinguish them from calls:

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)

Note: separate toplevel expressions in a file by `;;` to distinguish them from calls:

$$
\begin{aligned}
topdecls &::= exp\ tdrest \mid definition\ tdrest \\
definition &::= \texttt{let}\ id\ id \ldots id = exp \\
tdrest &::= \epsilon \qquad \text{(the 'empty production')} \\
&\mid \texttt{;;}\ exp\ tdrest \\
&\mid \texttt{;;}\ definition\ tdrest \qquad \text{(colons are optional here)}
\end{aligned}
$$

If not, OCaml thinks `f x [newline] f y` is one call with 3 args...

More OCaml

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>
```

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"
```

Here each pattern is just a constant (literal):

$$pattern ::= value$$

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 matching order now matters:

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

Pattern matching (4/5)

Beware that the pattern matching 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>
```

Underscores add another possible grammar production:

$$pattern ::= \_ \mid value$$

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
  | m -> m * fac (m - 1)
```

to which OCaml (again) responds:

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

Our extended pattern grammar now reads:

$$pattern ::= \_ \mid id \mid value$$

Tuples

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

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

which declares point3d as an alias for `int` triples

OCaml will infer tuple types (you don't need to declare them):

```
# 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 (here: on pair) 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>
```

(`simple.ml` from the screen cast also does this)

Tuple matching

One can also pattern match (here: on pair) 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>
```

(`simple.ml` from the screen cast also does this)

This match is equivalent to **`match`** `p` **`with`** `(x,y) ->`

Tuple matching

One can also pattern match (here: on pair) 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>
```

(`simple.ml` from the screen cast also does this)

This match is equivalent to **`match`** `p` **`with`** `(x,y) ->`

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)
```

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 type as `List<Integer>`

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 type 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]
```

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 type 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.

Since a list has two cases (empty, non-empty), a code skeleton is immediate from the type:

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

Lists, polymorphically

We can now write structurally recursive functions over lists.

Since a list has two cases (empty, non-empty), a code skeleton is immediate from the type:

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

Lists, polymorphically

We can now write structurally recursive functions over lists.

Since a list has two cases (empty, non-empty), a code skeleton is immediate from the type:

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

Lists, polymorphically

We can now write structurally recursive functions over lists.

Since a list has two cases (empty, non-empty), a code skeleton is immediate from the type:

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

OCaml infers a polymorphic type for this definition:

```
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 `Stdlib` (was: `Pervasives`) are visible at the top level.
- Many of the functions we have covered (and more) come from `Stdlib` – 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

Last time 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]
```

Naming tests and increasing test iterations

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

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

while option `~verbose:true` makes the test run's output 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];;
random seed: 25032179
generated error fail pass / total     time test name
[✓]  300     0    0  300 /  300     0.0s floor test
==================================================
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 (with little output).

The command-line argument `--verbose` (or `-v`) has the same effect as passing `~verbose:true` to `QCheck_runner.run_tests`

In addition it accepts `--seed` (or `-s`) for seeding the randomization (useful to recreate a problem)

A QCheck note on iteration count

In QCheck the `~count:n` parameter is bounded upwards by the option `~max_gen:m` which may surprise:

```
 # 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];;
random seed: 186572156
generated error  fail  pass / total     time test name
[✓]  1000     0      0  1000 / 10000     0.0s anon_test_1
==================================================================
success (ran 1 tests)
```

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

A QCheck note on iteration count

In QCheck the `~count:n` parameter is bounded upwards by the option `~max_gen:m` which may surprise:

```
 # 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];;
random seed: 186572156
generated error  fail  pass / total     time test name
[✓]  1000     0      0  1000 / 10000     0.0s anon_test_1
==================================================================
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:

```
generated error fail pass / total     time test name
[✓]  201    0    0  100 /  100     0.0s succ test
```

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:

```
generated error fail pass / total     time test name
[✓]  201    0    0  100 /  100     0.0s succ test
```

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:

```
[✓]  100    0    0  100 /  100     0.0s succ test'
```

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 to have enough that satisfies 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:

```
generated error fail pass / total     time test name
[✓]  200    0    0   95 /  100     0.0s succ test
```

When generation is expensive you may want to limit it.

Alternative: Write a custom generator whose output is guaranteed to satisfy the precondition (more later)

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, e.g.:

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

will thus error:

```
generated error fail pass / total     time test name
[✗]    45     1    0   44 /  100     0.0s div test

=== Error ==========================================

Test div test errored on (0 shrink steps):

0

exception Division_by_zero

============================================================
failure (0 tests failed, 1 tests errored, ran 1 tests)
```

Note how such errors are listed+counted separately

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_signed_int`
  - `...`
- There are many more (see the API):

  `http://c-cube.github.io/qcheck/0.16/`

Observing generators

It is simple to observe a generator with `Gen.generate1`:

```
# Gen.generate1 float.gen;;
- : float = 259083.184082186432
# Gen.generate1 float.gen;;
- : float = -0.000129108526967527053
```

There's also `Gen.generate` which accepts labeled argument `n`:

```
# Gen.generate ~n:5 small_int.gen;;
- : int list = [374; 7; 29; 0; 3]
```

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

where one then acts as oracle to the other.

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

where one then acts as oracle to the other.

This approach is also referred to as differential testing.

For example: a data structure with poor and improved $\mathcal{O}$-bounds on time (or space) complexity

Testing pairs (1/2)

Suppose we write a (terrible) 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:

```
generated error fail pass / total     time test name
[✓]  100     0    0  100 /  100     0.0s mymult,* agreement
```

(for negative numbers – with the sign-bit set – this is not immediately obvious)

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, ...

Summary

We can

- write general properties (in QCheck)
  - as Boolean-valued functions
  - with preconditions using `==>`
  - that test two implementations against each other
- formulate generators
  - based on builtin ones `int`, `pos_int`, `float`,
  - for tuples with `pair`, `triple`,...
  - for lists with `list`