SM2-TES: Functional Programming and Property-Based Testing, Day 3

Jan Midtgaard

MMMI, SDU

SDU

Last lecture's exercises

Lessons learned

To summarize some of our hard-learned lessons:

- Many errors can lurk in a first specification
- You understand code better by exploring the spec.

  – mostly if it fails
- Sometimes the error is in the code,

  – and sometimes the error is in the spec.

Outline

Even More OCaml

QuickChecking Algebraic Datatypes

Observing Generators

Testing Exception-Throwing Code

Even More OCaml

OCaml recap

You've now written (and tested) multiple OCaml functions following the below grammar:

$$
\begin{aligned}
topdecls &::= (exp \mid definition)\ (\ \texttt{;;}\ exp \mid\ \texttt{;;}\ definition)^* \\
definition &::= \texttt{let}\ id\ pat \ldots pat = exp \\
exp &::= id \\
&\mid value \\
&\mid exp + exp \mid exp - exp \mid \ldots \mid -exp \\
&\mid \texttt{fun}\ pat \ldots pat\ \texttt{->}\ exp \\
&\mid exp\ exp \ldots exp \\
&\mid \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp \\
&\mid (exp\texttt{,}\ \ldots\ \texttt{,}exp) \\
&\mid \texttt{let}\ id\ pat \ldots pat = exp\ \texttt{in}\ exp \\
&\mid \texttt{match}\ exp\ \texttt{with}\ \mid pat\ \texttt{->}\ exp \mid \ldots \mid pat\ \texttt{->}\ exp \\
&\mid [exp\texttt{;}\ \ldots\ \texttt{;}exp]
\end{aligned}
$$

(leaving out the labeled and optional arguments)

Data types (1/3)

- One can also form new types by creating disjoint unions (aka algebraic data types)
- They represent a variant (like an enum in C or Java) with a limited (closed) number of choices.
- For example:

```
type mybool =
    | Mytrue
    | Myfalse
```

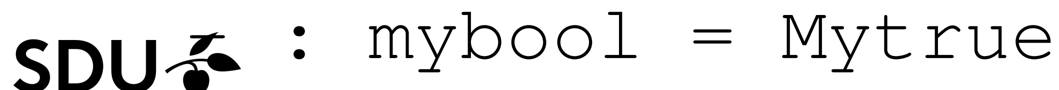
Data types (1/3)

- One can also form new types by creating disjoint unions (aka algebraic data types)
- They represent a variant (like an enum in C or Java) with a limited (closed) number of choices.
- For example:

```
type mybool =
    | Mytrue
    | Myfalse
```

- Note: the constructors have to be upper case
- Values are created with the constructors:

```
# Mytrue;;
- : mybool = Mytrue
```



Data types (2/3)

- We can process the data types using pattern matching
- There is typically one case per constructor

- For example:

```
let mybool_to_bool mb = match mb with
   | Mytrue  -> ...
   | Myfalse -> ...
```

SDU

Data types (2/3)

- We can process the data types using pattern matching
- There is typically one case per constructor
- For example:

```
let mybool_to_bool mb = match mb with
   | Mytrue  -> true
   | Myfalse -> false
```

- In this way we let the types guide us

— They suggest a skeleton for the code

Data types (3/3)

Data types can also carry data:

```
type card =
    | Clubs of int
    | Spades of int
    | Hearts of int
    | Diamonds of int
```

which we also extract by pattern matching:

```
let card_to_string c = match c with
  | Clubs i    -> ...
  | Spades i   -> ...
  | Hearts i   -> ...
  | Diamonds i -> ...
```

Data types (3/3)

Data types can also carry data:

```
type card =
  | Clubs of int
  | Spades of int
  | Hearts of int
  | Diamonds of int
```

which we also extract by pattern matching:

```
let card_to_string c = match c with
  | Clubs i    -> (string_of_int i) ^ " of clubs"
  | Spades i   -> (string_of_int i) ^ " of spades"
  | Hearts i   -> (string_of_int i) ^ " of hearts"
  | Diamonds i -> (string_of_int i) ^ " of diamonds"
```

Polymorphic data types

- OCaml comes with a builtin option type

```
type 'a option =
  | None
  | Some of 'a
```

- Note: option is polymorphic (Java-speak: generic)
  — it can carry any kind of data
- This is handy for signalling "data is there / isn't there"
- For example:

```
let first_elem l = match l with
  | []    -> ...
  | e::es -> ...
```

SDU

Polymorphic data types

- OCaml comes with a builtin option type

```
type 'a option =
  | None
  | Some of 'a
```

- Note: option is polymorphic (Java-speak: generic)
  — it can carry any kind of data
- This is handy for signalling "data is there / isn't there"
- For example:

```
let first_elem l = match l with
  | []    -> None    (*no first element!*)
  | e::es -> ...
```

Polymorphic data types

- OCaml comes with a builtin option type

```
type 'a option =
   | None
   | Some of 'a
```

- Note: option is polymorphic (Java-speak: generic)
  — it can carry any kind of data
- This is handy for signalling "data is there / isn't there"
- For example:

```
let first_elem l = match l with
   | []     -> None
   | e::es -> Some e (*first elem present*)
```

Polymorphic data types

- OCaml comes with a builtin option type

```
type 'a option =
   | None
   | Some of 'a
```

- Note: option is polymorphic (Java-speak: generic)
  — it can carry any kind of data
- This is handy for signalling “data is there / isn't there”
- For example:

```
let first_elem l = match l with
   | []    -> None
   | e::es -> Some e (*first elem present*)
```

for which OCaml infers a polymorphic type:

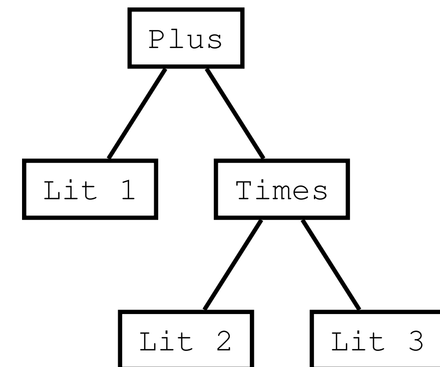
```
val first_elem :  'a list -> 'a option = <fun>
```

Example: Aritmetic expressions (1/4)

Here's a data type representing arithmetic expressions:

```
type aexp =
    | Lit of int
    | Plus of aexp * aexp
    | Times of aexp * aexp
```

Let's build a little tree data structure representing `1+2*3`:




```
# let mytree = Plus (Lit 1, Times (Lit 2, Lit 3));;
val mytree : aexp = Plus (Lit 1, Times (Lit 2, Lit 3))
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> ...
  | Plus (ae0, ae1) ->
    ...


  | Times (ae0, ae1) ->
    ...
```

SDU

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    ...


  | Times (ae0, ae1) ->
    ...
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    let v0 = interpret ae0 in
    ...

  | Times (ae0, ae1) ->
    ...
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    ...
  | Times (ae0, ae1) ->
    ...
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    v0 + v1
  | Times (ae0, ae1) ->
    ...
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    v0 + v1
  | Times (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    v0 * v1
```

Example: Aritmetic expressions (2/4)

Inductive datatypes and recursive functions make a powerful cocktail:

```
let rec interpret ae = match ae with
  | Lit i -> i
  | Plus (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    v0 + v1
  | Times (ae0, ae1) ->
    let v0 = interpret ae0 in
    let v1 = interpret ae1 in
    v0 * v1
```

Now let's run the thing:

```
# interpret mytree;;
- : int = 7
```

Example: Aritmetic expressions (3/4)

By a similar traversal we can serialize the tree into a string:

```
let rec exp_to_string ae = match ae with
  | Lit i -> ...
  | Plus (ae0, ae1) ->
    ...


  | Times (ae0, ae1) ->
    ...
```

Example: Aritmetic expressions (3/4)

By a similar traversal we can serialize the tree into a string:

```
let rec exp_to_string ae = match ae with
  | Lit i -> string_of_int i
  | Plus (ae0, ae1) ->
    let s0 = exp_to_string ae0 in
    let s1 = exp_to_string ae1 in
    "(" ^ s0 ^ "+" ^ s1 ^ ")"
  | Times (ae0, ae1) ->
    let s0 = exp_to_string ae0 in
    let s1 = exp_to_string ae1 in
    "(" ^ s0 ^ "*" ^ s1 ^ ")"
```

Example: Aritmetic expressions (3/4)

By a similar traversal we can serialize the tree into a string:

```
let rec exp_to_string ae = match ae with
    | Lit i -> string_of_int i
    | Plus (ae0, ae1) ->
      let s0 = exp_to_string ae0 in
      let s1 = exp_to_string ae1 in
      "(" ^ s0 ^ "+" ^ s1 ^ ")"
    | Times (ae0, ae1) ->
      let s0 = exp_to_string ae0 in
      let s1 = exp_to_string ae1 in
      "(" ^ s0 ^ "*" ^ s1 ^ ")"
```

And again call it:

```
# exp_to_string mytree;;
- :  string = "(1+(2*3))"
```

Example: Aritmetic expressions (4/4)

Now suppose we have a little target language for a simple stack machine:

```
type inst =
  | Push of int (* push an int to the stack *)
  | Add         (* pop two ints and push their sum *)
  | Mult        (* pop two ints and push their product *)
```

We can now write a compiler from arithmetic expressions into this type by a similar traversal...

SDU

Example: Aritmetic expressions (4/4)

Again there are three cases to consider:

```
let rec compile ae = match ae with
  | Lit i ->
    ...
  | Plus (ae0, ae1) ->
    ...


  | Times (ae0, ae1) ->
    ...
```

SDU

Example: Aritmetic expressions (4/4)

Again there are three cases to consider:

```
let rec compile ae = match ae with
  | Lit i ->
    [Push i]
  | Plus (ae0, ae1) ->
    ...


  | Times (ae0, ae1) ->
    ...
```

SDU

Example: Aritmetic expressions (4/4)

Again there are three cases to consider:

```
let rec compile ae = match ae with
  | Lit i ->
    [Push i]
  | Plus (ae0, ae1) ->
    let is0 = compile ae0 in
    let is1 = compile ae1 in
    is0 @ is1 @ [Add]
  | Times (ae0, ae1) ->
    ...
```

SDU

Example: Aritmetic expressions (4/4)

Again there are three cases to consider:

```
let rec compile ae = match ae with
  | Lit i ->
    [Push i]
  | Plus (ae0, ae1) ->
    let is0 = compile ae0 in
    let is1 = compile ae1 in
    is0 @ is1 @ [Add]
  | Times (ae0, ae1) ->
    let is0 = compile ae0 in
    let is1 = compile ae1 in
    is0 @ is1 @ [Mult]
```

Example: Aritmetic expressions (4/4)

Again there are three cases to consider:

```
let rec compile ae = match ae with
  | Lit i ->
    [Push i]
  | Plus (ae0, ae1) ->
    let is0 = compile ae0 in
    let is1 = compile ae1 in
    is0 @ is1 @ [Add]
  | Times (ae0, ae1) ->
    let is0 = compile ae0 in
    let is1 = compile ae1 in
    is0 @ is1 @ [Mult]
```

and we can now compile our syntax trees:

```
# compile mytree;;
- : inst list = [Push 1; Push 2; Push 3; Mult; Add]
```

References and side effects

- Assignment *is* available in OCaml
- You allocate a mutable cell in memory with **`ref`**:

```
let mycell = ref 0
```

OCaml responds: `val mycell : int ref`

References and side effects

- Assignment *is* available in OCaml
- You allocate a mutable cell in memory with **`ref`**:

```
let mycell = ref 0
```

  OCaml responds: `val mycell : int ref`
- One can assign a new value to the cell with `:=`

```
mycell := 42
```

References and side effects

- Assignment *is* available in OCaml
- You allocate a mutable cell in memory with **ref**:

```
let mycell = ref 0
```

  OCaml responds: `val mycell : int ref`
- One can assign a new value to the cell with `:=`

```
mycell := 42
```

- And read off the cell content by dereferencing the address:

```
# !mycell;;
- : int = 42
```

SDU

References and side effects

- Assignment *is* available in OCaml
- You allocate a mutable cell in memory with **`ref`**:

```
let mycell = ref 0
```

  OCaml responds: `val mycell : int ref`

- One can assign a new value to the cell with `:=`

```
mycell := 42
```

- And read off the cell content by dereferencing the address:

```
# !mycell;;
- : int = 42
```

- `incr` and `decr` from the standard library increment or decrement an integer reference, respectively

Records

- Records are a viable alternative to tuples as they name each entry (field) with a label
- Records need to be declared up front:

```
type person = { name : string;
                age  : int }
```

- After which record values can be created:

```
let someguy = { name = "Jan";
                age  = 77  }
```

- Entries are looked up using the familiar dot syntax:

```
let person_to_string p =
  p.name ^ ",␣" ^ (string_of_int p.age) ^ "␣years"

    # person_to_string someguy;;
    - : string = "Jan,␣77␣years"
```

Exceptions

- Exceptions are declared with the **exception** keyword:

```
exception Darn_list_is_empty
```

- They are created with the constructor and thrown with `raise`:

```
let first_elem' l = match l with
   | []     -> raise Darn_list_is_empty
   | e::es -> e
```

- Finally they are handled with the **try**/**with** construct:

```
try first_elem' []
with Darn_list_is_empty -> 0
```

Sequential evaluation

By now you've all written OCaml files structured as

```
e1;;
e2;;
e3;;
```

In the presence of side effects (e.g., printing) it's handy to locally evaluate expressions sequentially.

We can do so with **begin** . . . **end**:

```
begin
  print_string ("OMG␣OMG,␣this␣guy␣" ^ someguy.name);
  print_string ",␣he␣is␣like␣";
  print_int someguy.age;
  print_endline "␣years␣old!!!"
end
```

Sequential evaluation

By now you've all written OCaml files structured as

```
e1;;
e2;;
e3;;
```

In the presence of side effects (e.g., printing) it's handy to locally evaluate expressions sequentially.

We can do so with **begin** . . . **end**:

```
begin
  print_string ("OMG␣OMG,␣this␣guy␣" ^ someguy.name);
  print_string ",␣he␣is␣like␣";
  print_int someguy.age;
  print_endline "␣years␣old!!!"
end
```

which works as expected:

```
OMG OMG, this guy Jan, he is like 77 years old!!!
- : unit = ()
```

QuickChecking Algebraic Datatypes

Composing generators

QCheck has several operations for composing existing generators:

- we have already seen `pair` g_1 g_2 for constructing a pair generator out of two generators g_1 and g_2
- similarly there is `triple` g_1 g_2 g_3 for constructing a triple generator
- `oneof` $[g_1; \ldots; g_n]$ constructs a generator that chooses between generators $g_1, \ldots, g_n$
- `map` f g converts a generator g of type `'a` to a generator of type `'b` using a map f`: 'a -> 'b`

These builtin operations merge a number of concepts under the surface of `'a arbitrary`.

Full generators vs. bare generators (1/2)

To write our own generators for user-defined data types we need to separate the concepts again.

- In QCheck the type `'a arbitrary` covers both generation and printing

  It is defined as a record type:

```
type 'a arbitrary = {
  gen      : 'a Gen.t;                  (* "pure" generator *)
  print    : ('a -> string) option;     (* print values *)
  ...
  (* additional entries omitted *)
}
```

- In contrast, the type `'a Gen.t` denotes the bare generators of type `'a` (only generation)

SDU

Full generators vs. bare generators (2/2)

The operation

```
make ~print:p : 'a Gen.t -> 'a arbitrary
```

lets us build a full QCheck generator out of

- □ a pure generator and
- □ an optional printer `p`
- □ . . . (other optional parameters omitted for now)

Important note:

Without a printer QCheck cannot print counterexamples!

SDU

Generating user-defined data types (1/2)

QCheck has separate operations for composing pure generators (of type `'a Gen.t`):

- `Gen.pair` g_1 g_2 constructs a pure pair generator out of two pure generators g_1 and g_2
- `Gen.oneof` $[g_1; \ldots; g_n]$ constructs a pure generator that chooses between pure generators $g_1, \ldots, g_n$
- `Gen.map` f g converts a pure generator g of type `'a` to a pure generator of type `'b` using a map f`: 'a -> 'b`
- `Gen.map2` f g_1 g_2 converts two pure generators (g_1 of type `'a` and g_2 of type `'b`) to a pure generator of type `'c` using a map f`: 'a -> 'b -> 'c`
- and similarly for `Gen.map3`, ...

Generating user-defined data types (2/2)

For example this is a pure generator of AST leaves:

```
let leafgen = Gen.map (fun i -> Lit i) Gen.int
```

and we can use it to write a pure generator of ASTs:

```
let rec mygen n = match n with
  | 0 -> leafgen
  | n ->
    Gen.oneof
      [leafgen;
       Gen.map2 (fun l r -> Plus(l,r))
                (mygen (n/2)) (mygen (n/2));
       Gen.map2 (fun l r -> Times(l,r))
                (mygen (n/2)) (mygen (n/2)) ]
```

It's parameterized by a size limit `n` to bound the recursive generation (`n` ≈ "gas left in the tank").

Testing user-defined data types

With the pure generator in hand we can now test properties, e.g., of our `interpret` function:

```
let arb_tree = make ~print:exp_to_string (mygen 8)
let test_interpret =
  Test.make ~name:"test␣interpret"
    (pair arb_tree arb_tree)
    (fun (e0,e1) ->
       interpret (Plus(e0,e1))
                                  = interpret (Plus(e1,e0)))

;; (* remember this from the grammar *)
QCheck_runner.run_tests_main [test_interpret]
```

Testing user-defined data types

With the pure generator in hand we can now test properties, e.g., of our `interpret` function:

```
let arb_tree = make ~print:exp_to_string (mygen 8)
let test_interpret =
  Test.make ~name:"test␣interpret"
    (pair arb_tree arb_tree)
    (fun (e0,e1) ->
       interpret (Plus(e0,e1))
                           = interpret (Plus(e1,e0)))

;; (* remember this from the grammar *)
QCheck_runner.run_tests_main [test_interpret]
```

which we can run and verify:

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

Generating sized terms (1/2)

Size-bounded generators are so common that QCheck has a dedicated type for them:

```
type 'a Gen.sized = int -> Gen.t
```

There are special operations over these:

- `Gen.sized_size` i g converts an integer generator i and a size-bounded generator g to a pure generator (i is used to first generate a random size)
- `Gen.sized` g converts size-bounded generator g to a pure generator (it uses `small_int` to first generate a random size)
- `Gen.fix` f converts a `'a Gen.sized` transformer f into a recursive, size-bounded generator

Generating sized terms (2/2)

With these we can now write our example generator as:

```
let mygen =
  Gen.sized (Gen.fix
    (fun recgen n -> match n with
      | 0 -> leafgen
      | n ->
        Gen.oneof
          [leafgen;
           Gen.map2 (fun l r -> Plus(l,r))
                    (recgen (n/2)) (recgen (n/2));
           Gen.map2 (fun l r -> Times(l,r))
                    (recgen (n/2)) (recgen (n/2))]))
```

Note how this is no longer explicitly recursive:

The inner function just takes a sized generator `recgen` and returns a new one

Other primitives

In addition to `Gen.oneof`, `Gen.map`, `Gen.map2`, ... the `Gen` module contains a number of other useful operations:

- `Gen.return` v constructs a pure, constant generator that always returns v.
  Example: `Gen.return 0`
- `Gen.oneofl` $[v_1; \ldots; v_n]$ constructs a pure generator that generates one of the values $v_1, \ldots, v_n$.
  Example: `Gen.oneofl [min_int; max_int]`
- ...

For more details see

`http://c-cube.github.io/qcheck/dev/QCheck.Gen.html`

Generators with weights

The QuickCheck tester decides/programs the distribution of generated values.

We can also adjust the distribution with weights:

- `Gen.frequency` $[(w_1, g_1); \ldots; (w_n, g_n)]$ constructs a pure generator that chooses between pure generators $g_1, \ldots, g_n$ with integer weights $w_1, \ldots, w_n$, respectively (like `Gen.oneof`)
- `Gen.frequencyl` $[(w_1, v_1); \ldots; (w_n, v_n)]$ constructs a pure generator that generates one of the values $v_1, \ldots, v_n$ with integer weights $w_1, \ldots, w_n$, respectively (like `Gen.oneofl`)

  Ex: `Gen.frequencyl [(3,'a'); (1,'b')]` has 75% vs. 25% chance of generating `'a'` vs. `'b'`

SDU

Observing Generators

SDU

Observing Generators – Why?

QCheck lets us check a property across many inputs

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

Perhaps our (weighted) generator has an unfortunate skew?

In QCheck there are (so far) three ways to inspect the distribution of generators:

- Sampling (generate and look at some output)
- Statistics (by coersion into `int`)
- Collect (by coersion into `string`)

Each approach may be useful in different situations.

Observing our generator

The simplest observation is simply to call the underlying pure generator and study the outcome with `Gen.generate ~n:`i or `Gen.generate1`:

```
# Gen.generate ~n:10 Gen.small_int;;
- : int list = [44; 46; 3; 7; 12; 86; 5; 5; 3; 78]
# Gen.generate1 mygen;;
- : aexp = Lit 202555198053689434
# Gen.generate1 mygen;;
- : aexp =
Plus (Lit 2849725162213598392,
 Plus
  (Plus (Lit 594528501120269545,
    Plus (Lit 516453523062010388, Lit (-948838965895273413))),
  Lit 4234386505287435299))
```

You can observe any pure generator or full generator (by projecting its pure generator `g.gen`) in this way.

Statistics of generators (1/4)

When your data can be coerced to an `int` it is natural to compute some statistics of a generator's distribution.

Statistics support in QCheck is quite recent.

A statistical measure of some type `'a` is defined as a pair:

```
type 'a stat = string * ('a -> int)
```

- where the first component is a label and
- the second is a function for coercing the data into `int`

To support more than one statistical measure QCheck's operations accept a list of these: `'a stat list`

SDU

Statistics of generators (2/4)

You can enhance an existing generator to compute statistics in two ways:

- `set_stats` sl g returns a full generator like g, enhanced with the statistics list sl
- `make ~stats:`sl g returns a full generator when given the optional statistics list sl and a pure generator g

For example let's compute statistics using `bitwidth`:

```
let int_gen =
  set_stats [("bitwidth",fun i -> bitwidth i)] int in
Test.make ~count:10000 ~name:"true" int_gen (fun _ -> true)
```

This is a constant true test over `int`s

Statistics of generators (2/4)

You can enhance an existing generator to compute statistics in two ways:

- `set_stats` sl g returns a full generator like g, enhanced with the statistics list sl
- `make ~stats:`sl g returns a full generator when given the optional statistics list sl and a pure generator g

For example let's compute statistics using `bitwidth`:

```
let int_gen =
  set_stats [("bitwidth",bitwidth)] int in
Test.make ~count:10000 ~name:"true" int_gen (fun _ -> true)
```

This is a constant true test over `int`s

(We can abbreviate it a bit — this is called eta-reduction)

Statistics of generators (3/4)

The result is an additional output section:

```
+++ Stat ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Stat for test true:

stats bitwidth:
  num: 10000, avg:  62.01, stddev:  1.41, median 63, min 51, max 63
               51:                                                              2
               52:                                                              2
               53:                                                              6
               54:                                                             10
               55:                                                             26
               56:                                                             50
               57:                                                             65
               58: #                                                          137
               59: ###                                                        282
               60: ######                                                     623
               61: ##############                                            1304
               62: ###########################                               2488
               63: #######################################################   5005
```

Statistics of generators (3/4)

The result is an additional output section:

```
+++ Stat +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Stat for test true:

stats bitwidth:
  num: 10000, avg:  62.01, stddev:  1.41, median 63, min 51, max 63
             51:                                                                  2
             52:                                                                  2
             53:                                                                  6
             54:                                                                 10
             55:                                                                 26
             56:                                                                 50
             57:                                                                 65
             58: #                                                              137
             59: ###                                                            282
             60: ######                                                         623
             61: ##############                                                1304
             62: ###########################                                   2488
             63: #######################################################       5005
```

- 5005/10000 `int`s have the sign-bit set (no. 63)
- 2488/10000 `int`s have bit 62 as the highest one

SDU

Statistics of generators (3/4)

The result is an additional output section:

```
+++ Stat ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Stat for test true:

stats bitwidth:
  num: 10000, avg:  62.01, stddev:  1.41, median 63, min 51, max 63
               51:                                                        2
               52:                                                        2
               53:                                                        6
               54:                                                       10
               55:                                                       26
               56:                                                       50
               57:                                                       65
               58: #                                                    137
               59: ###                                                  282
               60: ######                                               623
               61: ##############                                      1304
               62: ###########################                         2488
               63: #######################################################  5005
```

- 5005/10000 `int`s have the sign-bit set (no. 63)
- 2488/10000 `int`s have bit 62 as the highest one

Q: A uniform distribution?

Statistics of generators (3/4)

The result is an additional output section:

```
+++ Stat +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Stat for test true:

stats bitwidth:
  num: 10000, avg:  62.01, stddev:  1.41, median 63, min 51, max 63
              51:                                                      2
              52:                                                      2
              53:                                                      6
              54:                                                     10
              55:                                                     26
              56:                                                     50
              57:                                                     65
              58: #                                                  137
              59: ###                                                282
              60: ######                                             623
              61: ##############                                    1304
              62: ###########################                       2488
              63: #######################################################  5005
```

- 5005/10000 `int`s have the sign-bit set (no. 63)
- 2488/10000 `int`s have bit 62 as the highest one

Q: A uniform distribution?

Q: What's the chance of generating a small integer?

Statistics of generators (4/4)

QCheck automatically merges consecutive integers when there are too many to output. For example

```
let list_gen =
  set_stats [("list␣length",List.length)] (list int) in
Test.make ~count:1000 list_gen (fun _ -> true)
```

yields:

```
num: 1000, avg:  447.01, stddev:  1463.62, median 9, min 0, max 9854
         0..492: #######################################################          845
       493..985: #####                                                             92
      986..1478:                                                                    1
     1479..1971:                                                                    6
     1972..2464:                                                                    5
     2465..2957:                                                                    2
     2958..3450:                                                                    4
     3451..3943:                                                                    5
     3944..4436:                                                                    4
     4437..4929:                                                                    3
     4930..5422:                                                                    4
     5423..5915:                                                                    2
     5916..6408:                                                                    4
     6409..6901:                                                                    4
     6902..7394:                                                                    2
     7395..7887:                                                                    2
     7888..8380:                                                                    2
     8381..8873:                                                                    2
     8874..9366:                                                                    6
     9367..9859:                                                                    5
    9860..10352:                                                                    0
```

Statistics of generators (4/4)

QCheck automatically merges consecutive integers when there are too many to output. For example

```
let list_gen =
  set_stats [("list␣length",List.length)] (list int) in
Test.make ~count:1000 list_gen (fun _ -> true)
```

yields:

```
num: 1000, avg:  447.01, stddev:  1463.62, median 9, min 0, max 9854
          0..492: #######################################################         845
        493..985: #####                                                            92
       986..1478:                                                                   1
      1479..1971:                                                                   6
      1972..2464:                                                                   5
      2465..2957:                                                                   2
      2958..3450:                                                                   4
      3451..3943:                                                                   5
      3944..4436:                                                                   4
      4437..4929:                                                                   3
      4930..5422:                                                                   4
      5423..5915:                                                                   2
      5916..6408:                                                                   4
      6409..6901:                                                                   4
      6902..7394:                                                                   2
      7395..7887:                                                                   2
      7888..8380:                                                                   2
      8381..8873:                                                                   2
      8874..9366:                                                                   6
      9367..9859:                                                                   5
     9860..10352:                                                                   0
```

Q: In your own words how is the distribution of list length?

SDU

Statistics of user-defined data types (1/3)

We can now inspect the output distribution of our programmed generator using statistics.

We first implement a straightforward height function:

```
let rec height ae = match ae with
  | Lit i -> 0
  | Plus (ae0, ae1) ->
    let h0 = height ae0 in
    let h1 = height ae1 in
    1 + (max h0 h1)
  | Times (ae0, ae1) ->
    let h0 = height ae0 in
    let h1 = height ae1 in
    1 + (max h0 h1)
```

which we can then use to classify the generated trees

Statistics of user-defined data types (2/3)

We can now calculate statistics, e.g., for 1000 trees:

```
let mygen = make ~stats:[("tree␣height", height)] mygen in
Test.make ~count:1000 mygen (fun _ -> true)
```

which reveals a reasonable distribution:

```
stats tree height:
  num: 1000, avg: 2.89, stddev: 3.45, median 2, min 0, max 14
               0: #######################################################        359
               1: ##################                                             122
               2: ##############                                                  92
               3: ##################                                             123
               4: ##########                                                      71
               5: ###                                                             25
               6: ######                                                          42
               7: #######                                                         49
               8: #                                                               12
               9: ###                                                             22
              10: ########                                                        55
              11:                                                                  0
              12:                                                                  3
              13: ##                                                              15
              14: #                                                               10
```

- In $\sim 36\%$ of the cases we generate a single leaf
- In $\sim 52\% \ (= 100 - 35.9 - 12.2)$ of the cases we generate a tree with height ≥ 2

Generators with weights: an example (1/2)

Suppose we are unhappy with this distribution.

We can, e.g., change it by adjusting the weights:

```
let mygen' =
  Gen.sized (Gen.fix
    (fun recgen n -> match n with
      | 0 -> leafgen
      | n ->
        Gen.frequency
          [(1,leafgen);
           (2,Gen.map2 (fun l r -> Plus(l,r))
                        (recgen (n/2)) (recgen (n/2)));
           (2,Gen.map2 (fun l r -> Times(l,r))
                        (recgen (n/2)) (recgen (n/2)))]))
```

When there's still gas left, this generates `Plus` or `Times` nodes with a 80% ($= 100 * \frac{2+2}{2+2+1}$) chance.

This is an increment from the original $\frac{2}{3}$ chance.

Generators with weights: an example (2/2)

The change has a visible effect:

```
stats tree height':
  num: 1000, avg: 3.99, stddev: 3.47, median 3, min 0, max 14
              0: #######################################################        226
              1: ##################                                              74
              2: ####################                                            83
              3: #####################################                          155
              4: ########################                                      102
              5: #######                                                         31
              6: ##################                                              74
              7: ####################                                            84
              8: #####                                                           21
              9: #########                                                       41
             10: ####################                                            84
             11: #                                                                5
             12: #                                                                5
             13: ##                                                              12
             14:                                                                  3
```

- Now in $\sim 23\%$ of the cases we generate a single leaf
- and in $\sim 70\%$ $(= 100 - 22.6 - 7.4)$ of the cases we generate a tree with height ≥ 2

SDU

Third option: `to_string` collection

Finally QCheck offers to collect data based on their string coercion into separate buckets.

This can be useful to observe
non-integer properties of a distribution

In QCheck data is grouped using "`to_string` functions" of type `'a -> string`.

Again there are two ways to collect:

- `set_collect` ts g returns a full generator like g that collects with the function ts
- `make ~collect:`ts g returns a full generator that collects with ts and generates data using the pure generator g.

SDU

Classifying generated pairs

Suppose we want to observe the input pairs 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 small_signed_int small_signed_int) in
Test.make ~name:"mymult,*␣agreement"
  pair_gen (fun (n,m) -> mymult n m = n * m)
```

which gives us

```
+++ Collect ++++++++++++++++++++++++++++++++++++++++++++++++++++

Collect results for test mymult,* agreement:

neg, neg: 20 cases
pos, pos: 27 cases
zero, neg: 1 cases
zero, pos: 2 cases
neg, zero: 2 cases
neg, pos: 24 cases
pos, neg: 21 cases
pos, zero: 3 cases
```

Classifying generated pairs

Suppose we want to observe the input pairs 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 small_signed_int small_signed_int) in
Test.make ~name:"mymult,* agreement"
  pair_gen (fun (n,m) -> mymult n m = n * m)
```

which gives us

```
+++ Collect ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Collect results for test mymult,* agreement:

neg, neg: 20 cases
pos, pos: 27 cases
zero, neg: 1 cases
zero, pos: 2 cases
neg, zero: 2 cases
neg, pos: 24 cases
pos, neg: 21 cases
pos, zero: 3 cases
```

No 0-pairs were generated

Increasing `count` *would increase their chance*

Testing Exception-Throwing Code

Testing exception-throwing functions

Sometimes your code (or worse: your specification!) throws an exception.

A lightweight approach is to simply catch plausible exceptions in the spec. For example:

```
Test.make ~name:"fac␣mod"
  (small_int_corners ())
  (fun n -> try (fac n) mod n = 0
            with Division_by_zero -> (n=0))
```

The added handler changes this behaviour

```
=== Error ======================================

Test fac mod errored on (0 shrink steps):

0

exception Division_by_zero
```

Testing exception-throwing functions

Sometimes your code (or worse: your specification!) throws an exception.

A lightweight approach is to simply catch plausible exceptions in the spec. For example:

```
Test.make ~name:"fac␣mod"
  (small_int_corners ())
  (fun n -> try (fac n) mod n = 0
            with Division_by_zero -> (n=0))
```

The added handler changes this behaviour . . . to this:

```
=== Error ======================================================

Test fac mod errored on (150 shrink steps):

262037

exception Stack overflow
```

. . . which we could then decide to add to the handler.

Negative tests

So far, we've mostly written positive tests:

– things that are supposed to work

Exception catching can also be used for negative tests

– things that are supposed to fail

For example:

```
Test.make ~name:"test␣exc" ~count:1000
  small_int
  (fun i -> try
              let _ = i/0 in
              false
            with Division_by_zero -> true)
```

has the expected effect:

```
law test exc: 1000 relevant cases (1000 total)
```

SDU

Summary

Today we have covered

- OCaml:
  - algebraic datatypes
  - other features: references, records, exceptions
- Quickchecking:
  - handwriting generators (pure vs. full)
  - 3 ways to study a generator (sampling, statistics, collecting)
  - testing exception-throwing code

SDU