02913 Advanced Analysis Techniques

QuickCheck, Day 3

Jan Midtgaard

DTU Compute

DTU

Yesterday'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.

DTU

Outline

Even More OCaml

QuickChecking Algebraic Datatypes

Shrinking

Testing Exception-Throwing Code

DTU

Even More OCaml

OCaml recap

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

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

(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 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 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
```

DTU

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 -> ...
```

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

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 -> ...
```

DTU

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"
```

DTU

Polymorphic data types

- OCaml comes with a builtin option type

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

- Note: option is polymorphic – 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 -> ...
```

Polymorphic data types

- OCaml comes with a builtin option type

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

- Note: option is polymorphic – 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 – 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 – 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 close to what you've seen in 02141:

```
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))
```

DTU

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) ->
    ...
```

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 run the thing:

```
# 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
  | Add
  | Mult
```

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

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) ->
    ...
```

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
```

- 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
```

- The standard library also comes with functions `incr` and `decr` for incrementing or decrementing an integer reference, respectively

Records

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

```
type identifier = { pos : int * int;
                    id  : string }
```

- After which record values can be created:

```
let someid = { pos = (2,3);
               id  = "x" }
```

- Entries are looked up using the familiar dot syntax:

```
let error unknownid =
  print_endline (unknownid.id ^ "␣is␣not␣defined")

# error someid;;  (* signal an example error *)
x is not defined
```

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 is handy to locally evaluate sequentially
We can do so with **`begin`** ... **`end`**:

```
begin
  print_string ("Warning:␣" ^ someid.id ^ "␣from␣line␣");
  print_int (fst someid.pos);
  print_string "␣is␣unused";
  print_newline()
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 is handy to locally evaluate sequentially
We can do so with **`begin`** ... **`end`**:

```
begin
  print_string ("Warning: " ^ someid.id ^ " from line ");
  print_int (fst someid.pos);
  print_string " is unused";
  print_newline()
end
```

which works as expected:

```
Warning: x from line 2 is unused
- : 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 *)
  ...
  collect : ('a -> string) option; (* classify values *)
}
```

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

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 an optional printer `p` and a pure generator

Without `collect` we don't classify input (as expected)

But more importantly:

Without a printer QCheck cannot print counterexamples!

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)) ]
```

which we've parameterized by a size limit `n`

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:

```
law test interpret: 100 relevant cases (100 total)
success (ran 1 tests)
```

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 (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

Observing our generator

In addition to classifying a full generator we can also call the underlying pure generator and study the outcome

For this QCheck provides `Gen.generate` n and `Gen.generate1`:

```
# Gen.generate1 mygen;;
- : aexp = Lit 20255551980536894 34
# Gen.generate1 mygen;;
- : aexp =
Plus (Lit 28497251622135983 92,
 Plus
  (Plus (Lit 594528501120269545,
    Plus (Lit 516453523062010388, Lit (-948838965895273413))),
  Lit 42343865052874352 99))
```

You can (of course) do the same for the builtin, pure generators such as `small_int.gen`

Other primitives

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

- `Gen.return` d constructs a pure, constant generator that always returns d
- `Gen.oneofl` $[d_1; \ldots; d_n]$ constructs a pure generator that generates one of the values $d_1, \ldots, d_n$
- ...

For more details see

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

Classifying user-defined data types (1/3)

Again, we can inspect the output distribution of our programmed generator using a classifier.

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 as a classifier

Classifying user-defined data types (2/3)

We can now classify, e.g., 1000 runs:

```
let height_pred e = let h = height e in
                    if h=0 then "␣0" else
                      if h=1 then "␣1" else ">1"
let mygen =
  set_collect
    (fun (e0,e1) -> "height:␣" ^ (height_pred e0)
                               ^ ",␣" ^ (height_pred e1))
    (pair arb_tree arb_tree)
let test_interpret =
  Test.make ~title:"test␣interpret" ~count:1000
    mygen
    (fun (e0,e1) ->
       interpret (Plus(e0,e1))
                              = interpret (Plus(e1,e0)))
```

based on a simple classification of height 0, 1, or above.

Classifying user-defined data types (3/3)

The output reveals a reasonable distribution:

```
law test interpret: 1000 relevant cases (1000 total)
  height:  1, >1: 38 cases
  height:  1,  0: 37 cases
  height:  1,  1: 8 cases
  height:  0, >1: 208 cases
  height:  0,  1: 38 cases
  height:  0,  0: 143 cases
  height: >1,  1: 51 cases
  height: >1, >1: 277 cases
  height: >1,  0: 200 cases
success (ran 1 tests)
```

- In $\sim 28\%$ of the cases we generate a pair of trees with height ≥ 2 ($\sim 53\%$ for a single tree)
- In $\sim 38\%$ of the cases we generate a single leaf

Generators with weights

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

QCheck offers other operations to 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` $[(w_1, d_1); \ldots; (w_n, d_n)]$ constructs a pure generator that generates one of the values $d_1, \ldots, d_n$ with integer weights $w_1, \ldots, w_n$, respectively (like `Gen.oneofl`)

DTU

Generators with weights: an example

We can decrease the chance of generating leaves:

```
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)))]))
```

which has a visible effect:

```
law test interpret: 1000 relevant cases (1000 total)
  height:  1, >1: 47 cases
  height:  1,  0: 13 cases
  height:  1,  1: 6 cases
  height:  0, >1: 149 cases
  height:  0,  1: 23 cases
  height:  0,  0: 49 cases
  height: >1,  1: 40 cases
```

Shrinking

Small and big counterexamples (1/2)

Suppose we try to QuickCheck a false claim, e.g., a buggy version of `rev_twice_test` from yesterday:

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

QCheck has post-processed the counterexample behind the scenes to present something easily understandable:

```
law rev twice: 1 relevant cases (1 total)
  test `rev twice` failed on ≥ 1 cases: [0; -1]
                              (after 6635 shrink steps)
```

Small and big counterexamples (2/2)

Suppose we try to QuickCheck the claim,
but disable the post-processing:

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

QCheck will present something less easily understandable (size varies with randomization seed):

```
law rev twice: 1 relevant cases (1 total)
  test `rev twice` failed on ≥ 1 cases:
  [17333575682935835O; 131331135304079374O;
   164912947165978625 1; 116216843269607130 5;
   164709751794012231 8; 266491200647788571;
   336338342765483454 7; 343527340326083156 9]
```

Big counterexamples

Once we move to user-defined data types the need becomes even more clear. Consider the false claim $\forall e.\, \texttt{interpret}\ (e + e) = \texttt{interpret}\ e$:

```
Test.make ~name:"test wrong"
  arb_tree
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

Big counterexamples

Once we move to user-defined data types the need becomes even more clear. Consider the false claim $\forall e.\, \texttt{interpret}\,(e + e) = \texttt{interpret}\, e$:

```
Test.make ~name:"test wrong"
  arb_tree
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

QCheck will refute this, but sometimes with needlessly huge, machine-generated counterexamples:

```
law test wrong: 1 relevant cases (1 total)
  test `test wrong` failed on ≥ 1 cases:
  ((-4473550023748818486+(28519751513096714 78*207733
431167466622 3))*((7555965334978816 14*-3736529492849 5
22494)+(-2763006417892435450+5275168495138130 47)))
```

Shrinkers and iterators in QCheck

- A shrinker attempts to cut a counterexample down to something more comprehensible for humans
- In QCheck shrinkers are implemented as iterators:
  `'a Shrink.t = 'a -> 'a QCheck.Iter.t`
- Given a counterexample, QCheck will repeatedly invoke the iterator to find a simpler value, possibly still representing a counterexample
- Internally iterators are observed with `Iter.find`:

```
# let i = Iter.of_list [0;1;2;3;4;5];;
val i : int QCheck.Iter.t = <fun>
# Iter.find (fun i -> true) i;;
- : int option = Some 0
# Iter.find (fun i -> i>=3) i;;
- : int option = Some 3
```

Builtin QCheck shrinkers

`set_shrink` s g constructs a shrinking generator out of a shrinker s and a generator g

We typically build new shrinkers as a combination of iterators and by composing builtin shrinkers

QCheck comes with a number of builtin shrinkers:

- `Shrink.nil` performs no shrinking
- `Shrink.int` for reducing integers
- `Shrink.string` for reducing strings
- `Shrink.list` for reducing lists
- `Shrink.pair` for reducing pairs
- ...

Builtin QCheck iterators

QCheck also comes with a number of builtin iterators:

- `Iter.empty` is an empty iterator
- `Iter.return` v is a one-element iterator
- `Iter.of_list` $[v_1; \ldots; v_n]$ creates an iterator out of a list of values
- `Iter.pair` i_1 i_2 creates a pair iterator out of two iterators
- `Iter.map` f g transforms an `'a` iterator g into a `'b` iterator using `f : 'a -> 'b`
- `Iter.append` i_1 i_2 combines two iterators sequentially
- ...

A shrinking example

We can hand-write a shrinker for our ASTs:

```
let rec tshrink e = match e with
  | Lit i ->
  | Plus (ae0, ae1)  ->



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

A shrinking example

We can hand-write a shrinker for our ASTs:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i -> Lit i) (Shrink.int i)
  | Plus (ae0, ae1)  ->



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

A shrinking example

We can hand-write a shrinker for our ASTs:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i -> Lit i) (Shrink.int i)
  | Plus (ae0, ae1)  ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Plus (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Plus (ae0,ae1')) (tshrink ae1)))
  | Times (ae0, ae1) ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Times (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Times (ae0,ae1')) (tshrink ae1)))
```

A shrinking example

We can hand-write a shrinker for our ASTs:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i -> Lit i) (Shrink.int i)
  | Plus (ae0, ae1)  ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Plus (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Plus (ae0,ae1')) (tshrink ae1)))
  | Times (ae0, ae1) ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Times (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Times (ae0,ae1')) (tshrink ae1)))
```

and use it to cut down a counterexample:

```
  Test.make ~name:"test␣wrong"
    (set_shrink tshrink arb_tree)
    (fun e -> interpret (Plus(e,e)) = interpret e)
```

A shrinking example

We can hand-write a shrinker for our ASTs:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i -> Lit i) (Shrink.int i)
  | Plus (ae0, ae1)  ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Plus (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Plus (ae0,ae1')) (tshrink ae1)))
  | Times (ae0, ae1) ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Times (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Times (ae0,ae1')) (tshrink ae1)))
```

and use it to cut down a counterexample:

```
Test.make ~name:"test␣wrong"
  (set_shrink tshrink arb_tree)
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

with a nice effect:

```
law test wrong: 1 relevant cases (1 total)
  test `test wrong` failed on ≥ 1 cases: -1 (after 66 shrink st
```

Testing Exception-Throwing Code

Testing exception-throwing functions

Sometimes your code (or 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

```
law fac mod:  0 relevant cases (1 total)
  test `fac mod` raised exception `Division_by_zero` on `0`
```

Testing exception-throwing functions

Sometimes your code (or 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:

```
law fac mod:  3 relevant cases (4 total)
  test `fac mod` raised exception `Stack overflow` on `262049`
                                        (after 138 shrink steps)
```

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

Negative tests

This idea 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)
```

DTU

Summary

Today we have covered

- OCaml:
  - algebraic datatypes
  - other features: references, records, exceptions
- Quickchecking:
  - handwriting generators (pure vs. full)
  - shrinking and iterators
  - testing exception-throwing code

DTU