SM2-TES: Functional Programming and Property-Based Testing, Day 8

Jan Midtgaard

MMMI, SDU

SDU

Outline

Talk: Growing and Shrinking Polygons . . .

Coverage

Program Generation

Talk: Growing and Shrinking Polygons . . .

Ilya Sergey, ICFP 2016:
Growing and Shrinking Polygons for
Random Testing of Computational Geometry

Coverage

Coverage, generally

It can be useful to compute coverage of a test suite.

It generally works by instrumenting the system's code to record the visited parts.

After having run tests, a coverage report then which parts (lines, branches) were visited and how much.

Coverage tools typically summarize the results in percent.

100% coverage is desirable but hard to achieve in practice (due to impossible code paths etc.)

Rationale: unvisited code is potentially untested.

There could be bugs hiding here.

Coverage in OCaml (1/3)

It is relatively easy to compute coverage of OCaml code with `bisect_ppx`:

Suppose we have the following code in a file:

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
;;
Printf.printf "%i\n" (fac 0)
```

Now:

Coverage in OCaml (1/3)

It is relatively easy to compute coverage of OCaml code with `bisect_ppx`:

Suppose we have the following code in a file:

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
;;
Printf.printf "%i\n" (fac 0)
```

Now:

`ocamlbuild -use-ocamlfind -package bisect_ppx fac.native`
(compile program with coverage instrumentation)

Coverage in OCaml (1/3)

It is relatively easy to compute coverage of OCaml code with `bisect_ppx`:

Suppose we have the following code in a file:

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
;;
Printf.printf "%i\n" (fac 0)
```

Now:

`ocamlbuild -use-ocamlfind -package bisect_ppx fac.native`
(compile program with coverage instrumentation)

`BISECT_COVERAGE=YES ./fac.native`
(run and record coverage to a file, e.g., `bisect676950869.coverage`)

Coverage in OCaml (1/3)

It is relatively easy to compute coverage of OCaml code with `bisect_ppx`:

Suppose we have the following code in a file:

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
;;
Printf.printf "%i\n" (fac 0)
```

Now:

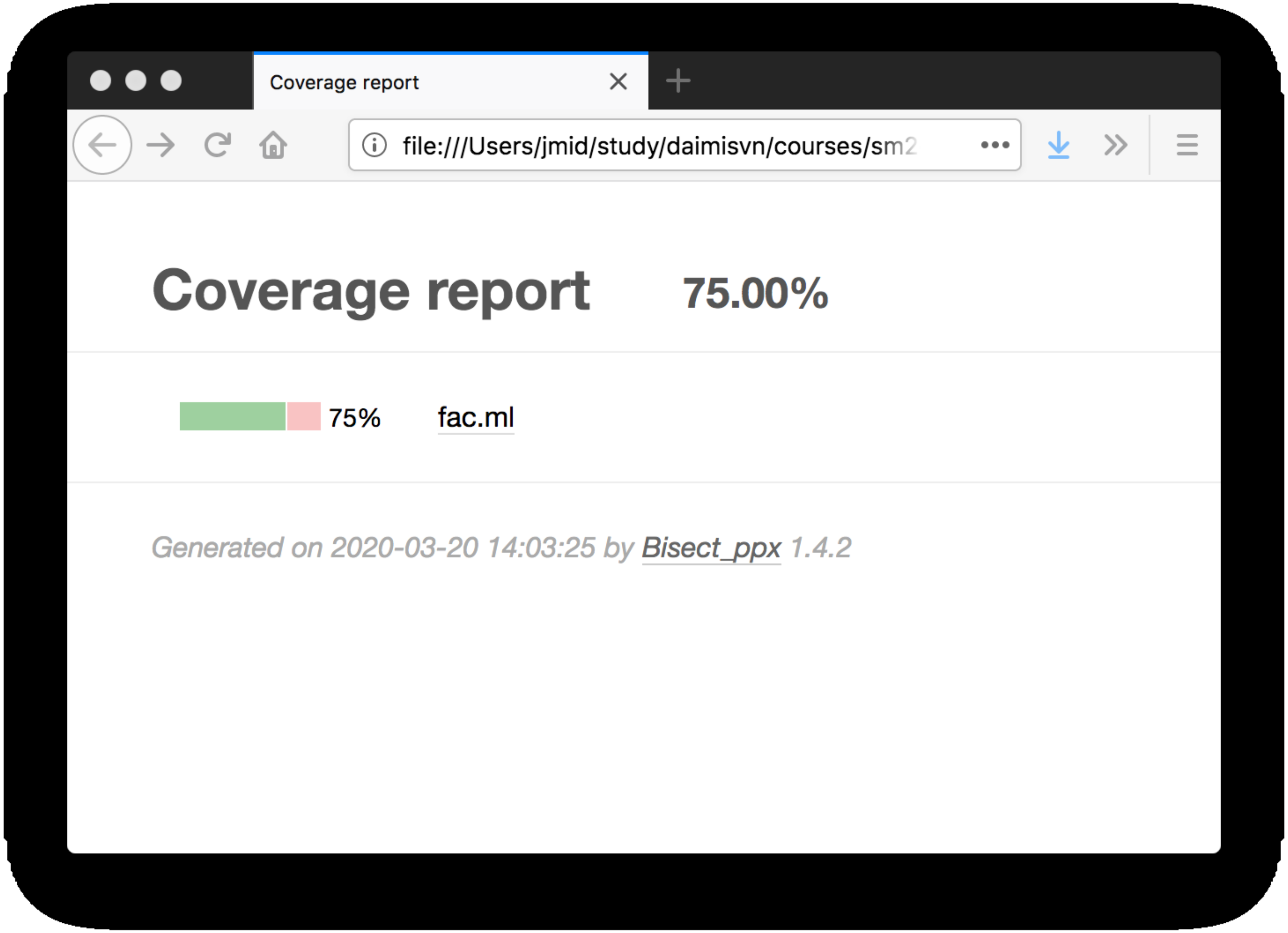
`ocamlbuild -use-ocamlfind -package bisect_ppx fac.native`
(compile program with coverage instrumentation)

`BISECT_COVERAGE=YES ./fac.native`
(run and record coverage to a file, e.g., `bisect676950869.coverage`)

`bisect-ppx-report html bisect676950869.coverage`
(make coverage report in HTML from `bisect676950869.coverage`
without a file name `bisect-ppx-report` searches in the current directory)

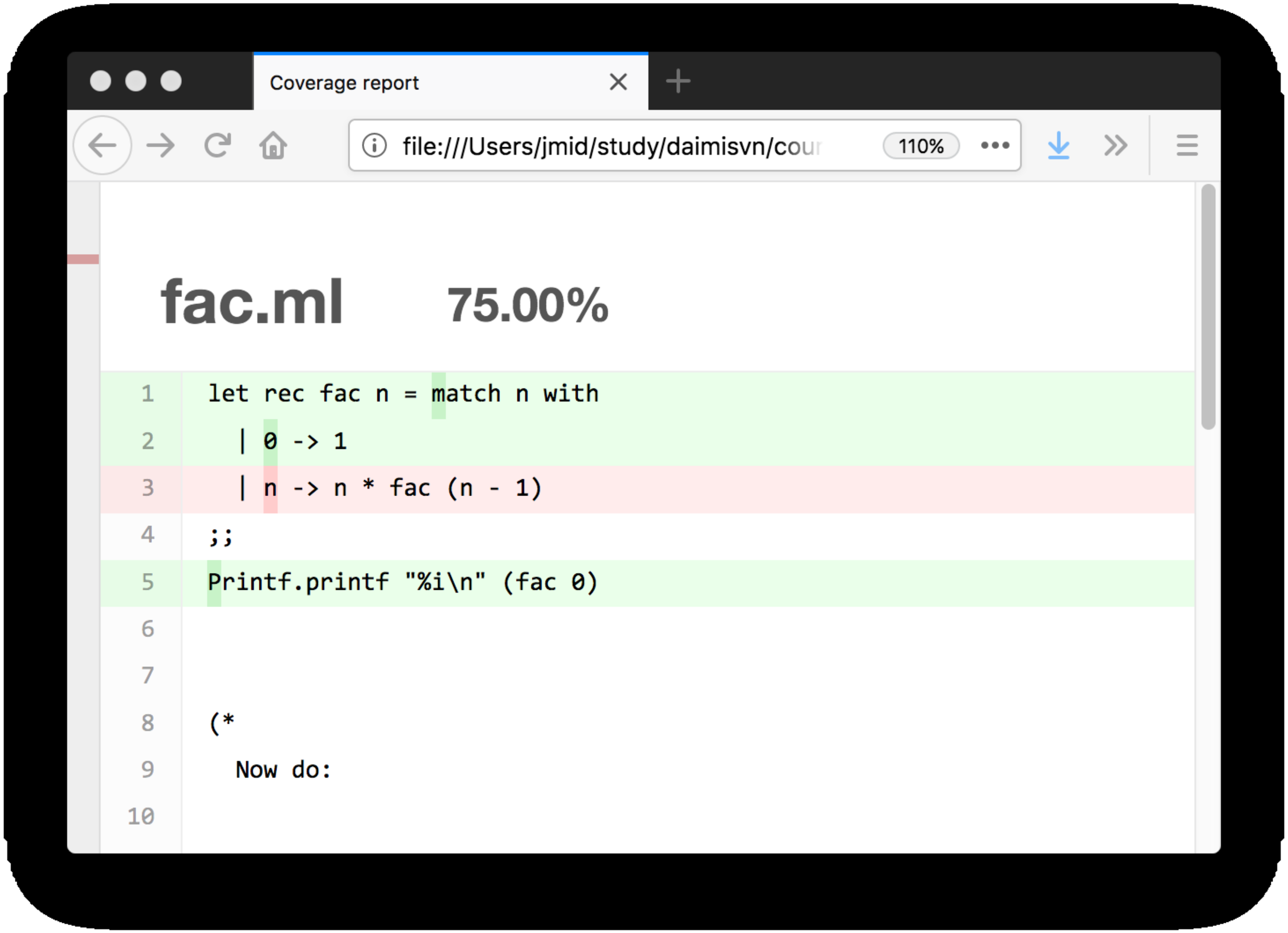
Coverage in OCaml (2/3)

Opening `_coverage/index.html`:



There are detailed reports for each source file

Coverage in OCaml (3/3)



Line color coding:

Visited points – Unvisited points – Line contains both

Coverage and QuickCheck (1/3)

So, (how) can we utilize coverage information within property-based testing?

Consider again the arithmetic expressions:

```
type aexp =
  | X
  | Lit of int
  | Plus of aexp * aexp
  | Times of aexp * aexp [@@deriving show]

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

Coverage and QuickCheck (2/3)

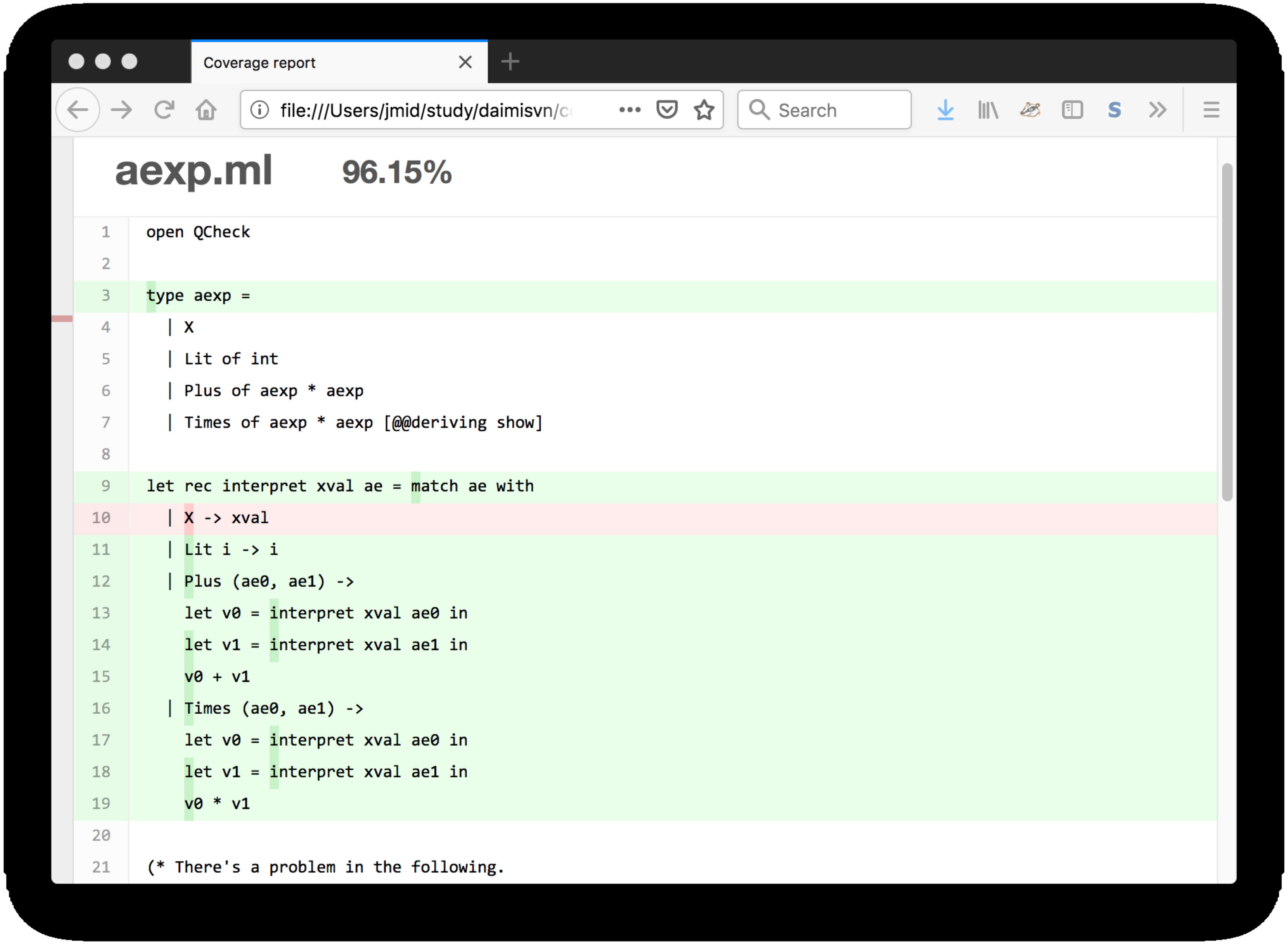
```
let leafgen = Gen.map (fun i -> Lit i) Gen.int
let mygen =
  Gen.sized (Gen.fix (fun recgen n -> match n with
    | 0 -> leafgen
    | n ->
      Gen.oneof
       [leafgen;
        Gen.map2 (fun l r -> Plus(l,r))
                    (recgen(n/2)) (recgen(n/2));
        Gen.map2 (fun l r -> Times(l,r))
                    (recgen(n/2)) (recgen(n/2))]))
let arb_tree = make ~print:show_aexp mygen

let test_interpret =
  Test.make ~name:"test␣interpret"
    (triple small_int arb_tree arb_tree)
    (fun (xval,e0,e1) -> interpret xval (Plus(e0,e1))
                         = interpret xval (Plus(e1,e0)))
;;
QCheck_runner.run_tests ~verbose:true [test_interpret]
```


Can you spot any problems?

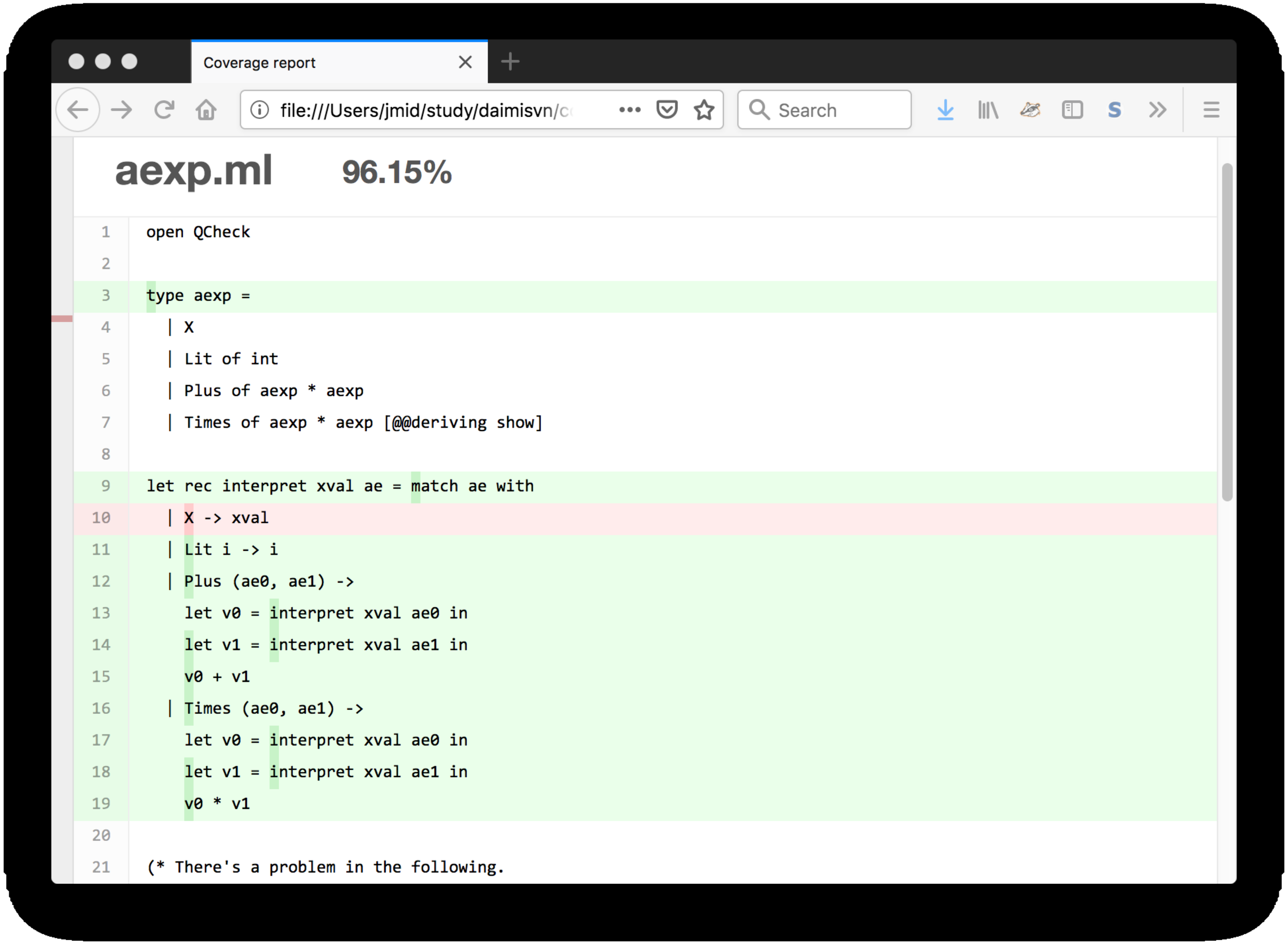
Coverage and QuickCheck (3/3)

The coverage report is informative:



Coverage and QuickCheck (3/3)

The coverage report is informative:



So coverage can give feedback, e.g., about generators!

Program Generation

How can we test a compiler?

(or generally: a language processor)

How can we generate random programs? (to do so)

Take 1: Program generation as random strings

Generate arbitrary strings:

```
Gen.generate1 (Gen.string ~gen:Gen.printable)   ?
```

Take 1: Program generation as random strings

Generate arbitrary strings:

```
Gen.generate1 (Gen.string ~gen:Gen.printable)  ?
```

```
"␣W"
```

Take 1: Program generation as random strings

Generate arbitrary strings:

```
Gen.generate1 (Gen.string ~gen:Gen.printable)   ?
```

```
"␣W"
```

```
"Jh/M{8/:m%_}d,w'Tc^$"
```

Take 1: Program generation as random strings

Generate arbitrary strings:

```
Gen.generate1 (Gen.string ~gen:Gen.printable)   ?

" W"

"Jh/M{8/:m%_}d,w'Tc^$"

"y?V&TIW%D$R\\@i5dRh>2EvF\nv<N:0%CGv>\nvJ[KJ1hR_
   :,M|RBP!aj>ymY7|3\"=N1]3r)]gmf[u:01v8Ln;.&`
   c@q9R;u3Mzczhhn;27\"zU.)x|[pIm.=e|DSdXsd1[;3B
   }2o@(s_|LV0irR'CH-su>8J49h-l-MARkQ[4+O(lyQu\"
   nZ)!K*5Yh#r!;rs+O9,I*oW6BY9VWZ\nan(VnI!N=PKv,
   wJS\\CU298\nzeA3<*Ag<@#Qu_]!T3A6X'^P7s/Q[RP.K
   }\\#5e+Q`"
```

Take 1: Program generation as random strings

Generate arbitrary strings:

```
Gen.generate1 (Gen.string ~gen:Gen.printable)  ?

"␣W"

"Jh/M{8/:m%_}d,w'Tc^$"

"y?V&TIW%D$R\\@i5dRh>2EvF\nv<N:0%CGv>\nvJ[KJ1hR_
   :,M|RBP!aj>ymY7|3\"=N1]3r)]gmf[u:01v8Ln;.&`
   c@q9R;u3Mzczhhn;27\"zU.)x|[pIm.=e|DSdXsd1[;3B
   }2o@(s_|LV0irR'CH-su>8J49h-l-MARkQ[4+O(lyQu\"
   nZ)!K*5Yh#r!;rs+O9,I*oW6BY9VWZ\nan(VnI!N=PKv,
   wJS\\CU298\nzeA3<*Ag<@#Qu_]!T3A6X'^P7s/Q[RP.K
   }\\#5e+Q`"
```

Perhaps as a stress test, but few of these will make it through the lexer and the parser. . .

Take 2: Grammar-based generation (1/2)

A grammar specifies the structure of programs:

$$e \ ::= \ \mathtt{x} \ | \ i \ | \ e + e \ | \ e * e$$

We have already seen such a generator:

```
open Gen
let leafgen = oneof [return X; map (fun i -> Lit i) int]
let mygen = sized (fix (fun rgen n -> match n with
  | 0 -> leafgen
  | n ->
    oneof
      [leafgen;
       map2 (fun l r -> Plus(l,r)) (rgen (n/2)) (rgen (n/2));
       map2 (fun l r -> Times(l,r)) (rgen (n/2)) (rgen (n/2))]
  ))
```

This generator is structured like the grammar.

When we run out of fuel, we generate a terminal.

SDU Otherwise we choose one of the four productions.

Take 2: Grammar-based generation (2/2)

This approach seems to work better:

```
0

(((x+40)*x)+x)

(((x*x)+(x*1))+((46*-1)*x))

(((16*((x+x)+(x*x)))*((x+(-7*x))+((-6*-5)+(x
    *-8))))*x)

(((4*(((x*x)+(0*-48))*((93+19)+(x*-70))))+x)*(x
    +((((x+x)+x)+0)+(((x+-73)+(x*x))+((x+-32)+(x
    +2))))))
```

There's only one variable x though...

SDU

Take 2.5: Multiple variables (1/2)

To go beyond just `x` we first extend the type:

```
type aexp =
  | Var of string          (* a string models var.name *)
  | Lit of int
  | Plus of aexp * aexp
  | Times of aexp * aexp
```

Take 2.5: Multiple variables (1/2)

To go beyond just `x` we first extend the type:

```
type aexp =
  | Var of string          (* a string models var.name *)
  | Lit of int
  | Plus of aexp * aexp
  | Times of aexp * aexp
```

Based on this we can refine our generator:

```
open Gen

let vargen =
  string_size ~gen:(char_range 'a' 'z') (int_range 1 10)

let leafgen =
  oneof [map (fun v -> Var v) vargen;
         map (fun i -> Lit i) small_signed_int]
```

Variables have to be at least one character long. . .

Take 2.5: Multiple variables (2/2)

This seems to work reasonably:

```
(vyiir+wzmv)

(((-38+7)*(rraumlnjb*57))+((8*tjduu)+(5+a)))

((4*((mucflus+a)*(pqirp*)))+(-3+((yuyznrp+)*(-17+-1))))

((-6*((cgpsh*pwxbwi)*(2+4)))+(((-6*-6)*(-8+0))*((yprvdqd
    *-11)*dmt)))

(((-3+ipexzth)*(((((0+63)+(m*-2))+((wjwkn+dqwkyrtiw)+brmq
    ))+72)+99))+(((((ekj*-4)*7)*v)+(biibuqahb*o))+(((((
    buvulnr+yyld)+7)+(ylviji+(wigmjldr+xot)))*(((ewof+-8)*
    onjnrn)+((icl+jlgtxn)+(-7*tglkbek))))*-1)))
```

These expressions however refer to arbitrary variables and rarely the same ones…

Take 3: Environment-passing generator (1/2)

We can pass an environment of declared variables:

```
let leafgen env = match env with
  | [] -> map (fun i -> Lit i) small_signed_int
  | _  -> oneof [map (fun v -> Var v) (oneofl env);
                 map (fun i -> Lit i) small_signed_int]
```

If there are variables in the environment, we choose among them or generate a literal.

Take 3: Environment-passing generator (1/2)

We can pass an environment of declared variables:

```
let leafgen env = match env with
  | [] -> map (fun i -> Lit i) small_signed_int
  | _  -> oneof [map (fun v -> Var v) (oneofl env);
                 map (fun i -> Lit i) small_signed_int]
```

If there are variables in the environment, we choose among them or generate a literal.

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

The environment `env` is a parameter to our generator.

Take 3: Environment-passing generator (2/2)

To try it out we generate a variable list and pass that as our environment:

```
let proggen = Gen.small_list vargen >>= fun env -> mygen env
```

This seems to work reasonably well:

```
xilox

(-33*tyxzel)

(5*((ibkrjeaq*ibkrjeaq)+ibkrjeaq))

((ezzh+(((-5*ezzh)*((unart*rcnofq)*(-6+-4)))*yxxypzz))
    +(-3*(((vqhxpb+(-39+vqhxpb))+((-9+unart)+eqprbs))
    +(((-2*3)+unart)+((-8*rcnofq)+(-12+eqprbs))))))
```

Take 3: Environment-passing generator (2/2)

To try it out we generate a variable list and pass that as our environment:

```
let proggen = Gen.small_list vargen >>= fun env -> mygen env
```

This seems to work reasonably well:

```
xilox

(-33*tyxzel)

(5*((ibkrjeaq*ibkrjeaq)+ibkrjeaq))

((ezzh+(((-5*ezzh)*((unart*rcnofq)*(-6+-4)))*yxxypzz))
    +(-3*(((vqhxpb+(-39+vqhxpb))+((-9+unart)+eqprbs))
    +(((-2*3)+unart)+((-8*rcnofq)+(-12+eqprbs))))))
```

What if we want to support a bigger language?

Take 4: More non-terminals (1/4)

Let's now scale this approach to a bigger language with several non-terminals:

$$
\begin{aligned}
e &::= x \mid i \mid e + e \mid e * e \\
b &::= \texttt{false} \mid \texttt{true} \mid e < e \mid e <= e \mid e == e \\
s &::= x = e \mid \{\, slist \,\} \mid \texttt{if (}b\texttt{)}\ s \mid \texttt{while (}b\texttt{)}\ s \\
slist &::= \epsilon \mid s\ slist
\end{aligned}
$$

Take 4: More non-terminals (1/4)

Let's now scale this approach to a bigger language with several non-terminals:

$$
\begin{aligned}
e &::= x \mid i \mid e + e \mid e \star e \\
b &::= \texttt{false} \mid \texttt{true} \mid e < e \mid e <= e \mid e == e \\
s &::= x = e \mid \{ \mathit{slist} \} \mid \texttt{if}\ (b)\ s \mid \texttt{while}\ (b)\ s \\
\mathit{slist} &::= \epsilon \mid s\ \mathit{slist}
\end{aligned}
$$

The non-terminals map straightforwardly to type decls:

```
type relexp =
  | False
  | True
  | Lt of aexp * aexp
  | Le of aexp * aexp
  | Equal of aexp * aexp
```

```
type stmt =
  | Assign of string * aexp
  | Block of stmt list
  | If of relexp * stmt
  | While of relexp * stmt
```

(`aexp` looks like before)

Take 4: More non-terminals (2/4)

The generator of Boolean (relational) expressions is straightforward:

```
let relexpgen env n = match n with
    | 0 -> oneofl [False; True]
    | _ ->
      oneof
        [oneofl [False; True];
         map2 (fun l r -> Lt(l,r))    (aexp env (n/2)) (aexp env (n/2));
         map2 (fun l r -> Le(l,r))    (aexp env (n/2)) (aexp env (n/2));
         map2 (fun l r -> Equal(l,r)) (aexp env (n/2)) (aexp env (n/2))]
```

Take 4: More non-terminals (2/4)

The generator of Boolean (relational) expressions is straightforward:

```
let relexpgen env n = match n with
    | 0 -> oneofl [False; True]
    | _ ->
      oneof
        [oneofl [False; True];
         map2 (fun l r -> Lt(l,r))    (aexp env (n/2)) (aexp env (n/2));
         map2 (fun l r -> Le(l,r))    (aexp env (n/2)) (aexp env (n/2));
         map2 (fun l r -> Equal(l,r)) (aexp env (n/2)) (aexp env (n/2))]
```

For statements we first write an assignment generator:

```
let assign_gen env n = match env with
  | [] -> map2 (fun x ae -> Assign (x,ae)) vargen (aexp env n)
  | _  ->
    oneof [
      map2 (fun x ae -> Assign (x,ae)) (oneofl env) (aexp env n);
      map2 (fun x ae -> Assign (x,ae)) vargen (aexp env n) ]
```

This language has no variable declarations. When the environment is empty, we assign to a random variable.

Take 4: More non-terminals (3/4)

The statement generator consists of two mutually recursive functions `stmtgen` and `stmtlistgen`:

```
let rec stmtgen env = fix (fun rgen n -> match n with
  | 0 -> assign_gen env n
  | _ ->
    oneof
      [assign_gen env n;
       map  (fun ss   -> Block ss) (stmtlistgen env (n-1));
       map2 (fun re s -> If (re,s)) (relexpgen env (n/2)) (rgen (n/2));
       map2 (fun re s -> While (re,s)) (relexpgen env (n/2)) (rgen (n/2));
      ])

and stmtlistgen env n = match n with
  | 0 -> return []
  | _ ->
    stmtgen env (n/2) >>= fun s ->
      let env' = (match s with
        | Assign (x,_) -> if List.mem x env then env else x::env
        | _            -> env) in
      stmtlistgen env' (n/2) >>= fun ss -> return (s::ss)
```

The `Block`-generator uses `stmtlistgen`. If the last statement was an assignment, we may extend the `env`.

Take 4: More non-terminals (4/4)

This seems to work reasonably:

```
jlrjkxpep = ((((((2*-4)*(3+-6))*((2*-5)*(59+5)))+-2)+-4)
    +(((((0*0)+(81*-8))*((82*5)+(4+79)))*5)+0))
```

```
while (-62 <= (-4*-9))
   while (96 < 5)
   { ogdvjyva = 6
}
```

```
if ((((0*(2+-4))*4)+((2+(-5+-9))+((2*-5)*(-34*8)))) <=
    (43*(((50*14)+-8)+((0*-6)+(0+-8))))) while ((((0+0)
    *62)+(-7*(-9*-2))) == 4)
   if (((9*-95)*(-1*-9)) == -63) rbugny = (8*((-5+7)*4))
```

Next, let us try to test something with this generator...

A target for our programs: `bc`

We can use the generator to test `bc`, a command-line calculator that comes with Linux, Mac, and Unix historically:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software
   Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
1+3
4
x = 1
while (x < 10) { x = x + 1 }
print(x)
10
quit
```

Try typing `man bc` in the terminal to read `bc`'s manual.

A first property...

We write each generated program to a file `tmp.bc` and then pass its content to `bc`:

```
Test.make ~count:100 ~name:"bc␣test"
  arb_stmt
  (fun stmt ->
     let outch = open_out "tmp.bc" in
     Printf.fprintf outch "%s" (StmtLang.stmt_to_string stmt);
     close_out outch;
     0 = Sys.command "bc␣-q␣<␣tmp.bc␣>␣output.txt␣2>&1␣"
      && 0 <> Sys.command "grep␣-q␣error␣output.txt")
```

Even on a parse error `bc` returns error code `0` (success).

As a workaround we write `bc`'s output to `output.txt` and check if it contains `"error"`...

A first property. . .

We write each generated program to a file `tmp.bc` and then pass its content to `bc`:

```
Test.make ~count:100 ~name:"bc␣test"
  arb_stmt
  (fun stmt ->
     let outch = open_out "tmp.bc" in
     Printf.fprintf outch "%s" (StmtLang.stmt_to_string stmt);
     close_out outch;
     0 = Sys.command "bc␣-q␣<␣tmp.bc␣>␣output.txt␣2>&1␣"
      && 0 <> Sys.command "grep␣-q␣error␣output.txt")
```

Even on a parse error `bc` returns error code `0` (success).

As a workaround we write `bc`'s output to `output.txt` and check if it contains `"error"`. . .

Q: Which errors in `bc` can escape this property?

A first property...

We write each generated program to a file `tmp.bc` and then pass its content to `bc`:

```
Test.make ~count:100 ~name:"bc␣test"
  arb_stmt
  (fun stmt ->
     let outch = open_out "tmp.bc" in
     Printf.fprintf outch "%s" (StmtLang.stmt_to_string stmt);
     close_out outch;
     0 = Sys.command "bc␣-q␣<␣tmp.bc␣>␣output.txt␣2>&1␣"
      && 0 <> Sys.command "grep␣-q␣error␣output.txt")
```

Even on a parse error `bc` returns error code `0` (success).

As a workaround we write `bc`'s output to `output.txt` and check if it contains `"error"`...

Q: Which errors in `bc` can escape this property?

Q: What happens if we generate an infinite loop?

A refined property with `timeout`

This refined property runs `bc` with a `timeout` of 2 seconds:

```
Test.make ~count:100 ~name:"bc␣test"
  arb_stmt
  (fun stmt ->
     let outch = open_out "tmp.bc" in
     Printf.fprintf outch "%s" (StmtLang.stmt_to_string stmt);
     close_out outch;
     let retcode =
       Sys.command
         "timeout␣2␣bc␣-q␣<␣tmp.bc␣>␣output.txt␣2>&1␣" in
     (retcode = 0 || retcode = 124)
      && 0 <> Sys.command "grep␣-q␣error␣output.txt")
```

If `bc` hasn't completed within 2 seconds,

`timeout` interrupts and returns `124`.

We consider that a successful test case. . .

Running our `bc` tests

This seems to work:

```
generated error fail pass / total      time test name
[✓]  100     0     0  100 /  100    37.8s bc test
=========================================================================
success (ran 1 tests)
```

It takes a bit of time though, because of the timeouts...

Running our bc tests

This seems to work:

```
generated error fail pass / total      time test name
[✓]   100     0     0  100 /   100     37.8s bc test
==========================================================================
success (ran 1 tests)
```

It takes a bit of time though, because of the timeouts. . .

It is still not perfect though:

```
generated error fail pass / total      time test name
[✗]    58     0     1   57 /   100     23.1s bc test

--- Failure -----------------------------------------------------------

Test bc test failed (3 shrink steps):

{ rosvgl = if
}
```

Can you see the problem?

More types beyond integers

This is starting to look like a real imperative language.

For now, we only have one type of integers.

This means type checking will not be a problem, e.g., passing or assigning an `int` where a `string` was expected

Next time we will talk about how to generate type-correct programs. . .

Summary and conclusion

Today we have

- seen a completely different example use of QuickCheck within Computational Geometry
- talked about code coverage and how it may be useful — also within property-based testing
- taken the first steps within program generation
  - following the language grammar
  - passing an environment of variables
  - running the generated programs with a timeout