Get going with equations
for simple lists, queues and even arrays

Formal Specifications FOR DUMMIES®

Demystifies everything from algebraic specifications to abstraction functions!

A Reference for the Rest of Us!®

FREE eTips at dummies.com®

Koen Claessen
John Hughes
Nick Smallbone

Specifications are good to have

But they're such a pain to write

Enter QuickSpec!

What's QuickSpec?

QuickSpec takes a collection of functions and finds nice equations about them!

Caveat

Pure functions only

How does QuickSpec work?

Step 1: run a lot of tests on the functions

Step 2: work out what they do :)
(less facetious answer later)

A function

```
sort([]) →
   [];
sort([X|Xs]) →
   lists:merge([X], sort(Xs)).
```

Let's find out what it does!

What QuickSpec thinks

- First write some boilerplate in sort_sig.erl
- Then:

```
1> laws:laws(sort_sig, sort).
... QuickSpec thinks for a bit, and then ...
   1. sort([]) == []
   2. sort(reverse(Xs)) == sort(Xs)
   3. sort(sort(Xs)) == sort(Xs)
   4. sort(Ys ++ Xs) == sort(Xs ++ Ys)
   5. sort([X]) == [X]
```

What about usort?

- usort is like sort but removes duplicates

```
30> laws:laws(sort_sig, usort).
... QuickSpec thinks for a bit, and then ...
   1. usort([]) == []
   2. usort(reverse(Xs)) == usort(Xs)
   3. usort(usort(Xs)) == usort(Xs)
   4. usort(Ys ++ Xs) == usort(Xs ++ Ys)
   5. usort(Xs ++ Xs) == usort(Xs)
   6. usort([X]) == [X]
```

QuickSpec can help you understand functions

The boilerplate

- Tell QuickSpec what kind of equations to look for: which functions to use...

```
fun_types() ->
    [{lists, usort, [list], list},
     {erlang, '++', [list, list], list},
     {lists, reverse, [list], list},
     ...].
```

(which we can abbreviate like this:)

```
fun_types() ->
    [{lists, usort, [list], list}|
     stdsigs:list_funs(int, list)].
```

The boilerplate

- ...and variables:

```
var_types() ->
    [{[xs, ys, zs], list},
     {[x, y, z], int}].
```

The boilerplate

- Tell QuickSpec how to generate random test data to test the equations:

```
list() ->
    list(int()).
```

QuickSpec does the rest!

What can QuickSpec actually find?

- QuickSpec looks for *equations* built from the functions and variables you specify
- QuickSpec can only find equations that are not too deeply nested

`Xs`	depth 1
`Xs ++ []`	depth 2
`sort(Xs) ++ sort(Ys)`	depth 3
`Xs ++ sort(Ys ++ Zs)`	depth 4

Too deep

- But: if an equation is not too deep, it will always be found
- Are all the equations true? Maybe not, but they're well-tested

Looking at reverse instead

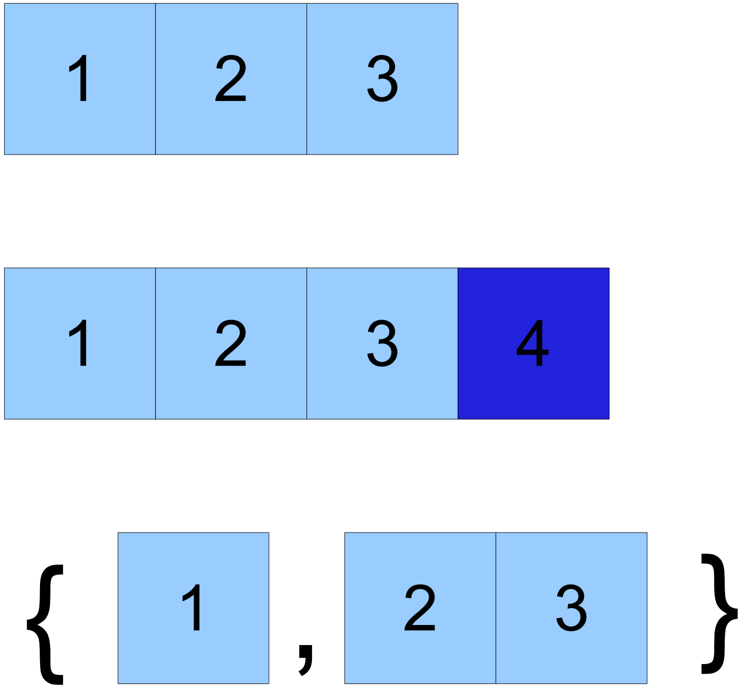
- Since reverse is in the signature, we can just as easily look at laws for that:

```
1> laws:laws(sort_sig, reverse).
... QuickSpec thinks for a bit, and then ...
   1. reverse([]) == []
   2. reverse(reverse(Xs)) == Xs
   3. sort(reverse(Xs)) == sort(Xs)
   4. reverse([X]) == [X]
   5. reverse(Xs) ++ reverse(Ys) == reverse(Ys ++ Xs)
   6. reverse(Xs) ++ ([X]) == reverse([X|Xs])
```

QuickSpec can discover properties for you

Another example: queues

The queue API

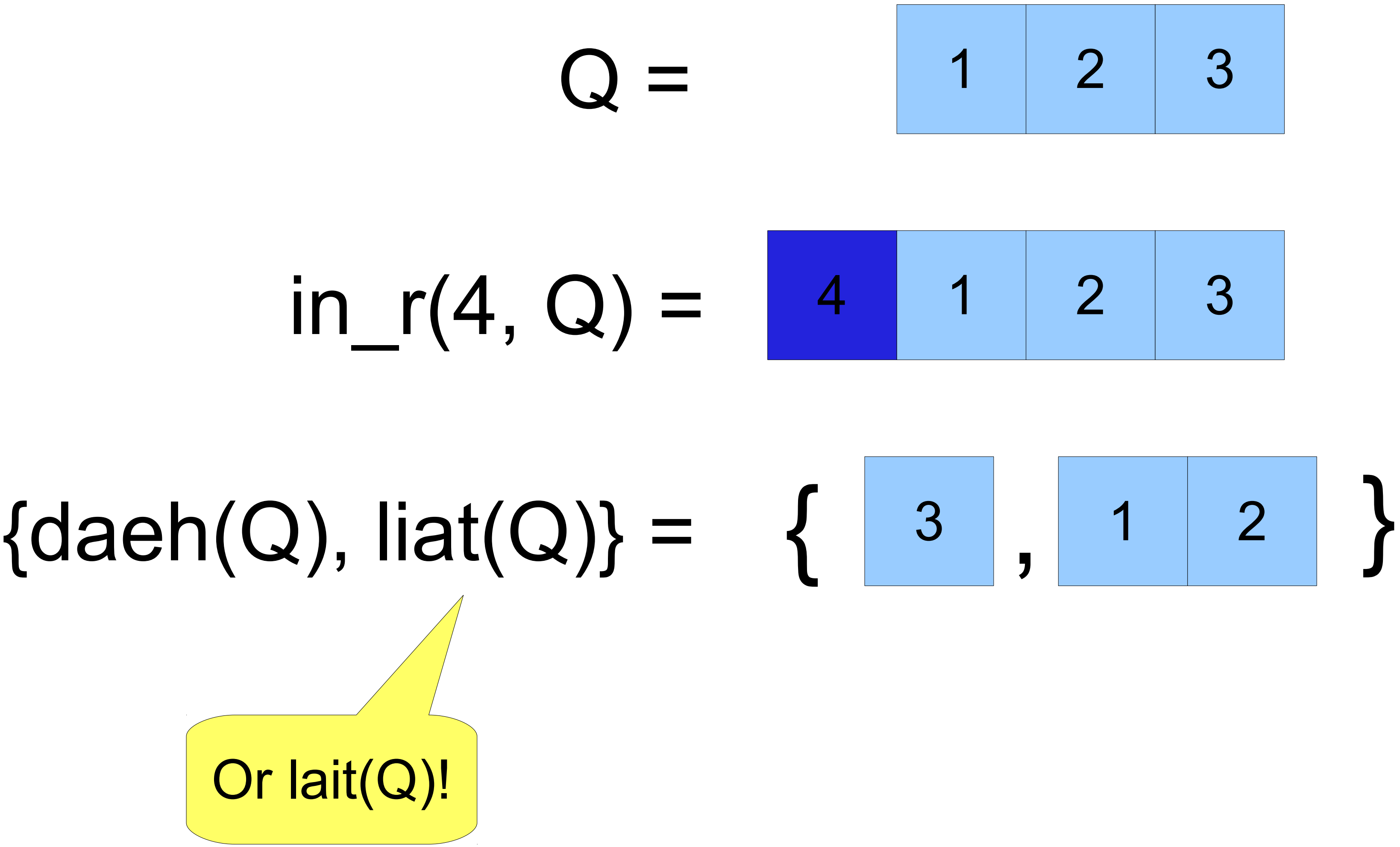


Q = 1 2 3

in(4, Q) = 1 2 3 4

{head(Q), tail(Q)} = { 1 , 2 3 }

The queue API, in reverse



Q = 1 2 3

in_r(4, Q) = 4 1 2 3

{daeh(Q), liat(Q)} = { 3 , 1 2 }

Or lait(Q)!

DEMO

queuesig.erl

The mystery of the missing law

QuickSpec prints

```
head(in_r(X,Q)) == X
tail(in_r(Y,Q)) == tail(in_r(X,Q))
```

But what about

```
tail(in_r(X,Q)) == Q?
```

Maybe QuickSpec didn't find it? No!

What can QuickSpec actually find?

- QuickSpec looks for *equations* built from the functions and variables you specify
- QuickSpec can only find equations that are not too deeply nested

```
Xs                              depth 1
Xs ++ []                        depth 2
sort(Xs) ++ sort(Ys)            depth 3
Xs ++ sort(Ys ++ Zs)            depth 4
```

Too deep

- But: **if an equation is not too deep, it will always be found**
- Are all the equations true? Maybe not, but they're well-tested

Missing equations

- If an equation doesn't get printed, either
  - It has a too deeply-nested expression
  - It uses some function that's not in your signature
  - It follows from the equations that *were* printed
  - **It's false!**
- The missing equation must be false...really?!

Why is our equation false?

- Test it with QuickCheck:

```
prop_in_r_tail() ->
    ?FORALL(X, int(),
    ?FORALL(Q, laws:symbolic(queuesig, queue),
    queue:tail(queue:in_r(X, eval(Q))) == eval(Q))).

1> eqc:quickcheck(queuesig:prop_in_r_tail()).
.....Failed! After 6 tests.
0
{call,queue,in,[0,{call,queue,new,[]}]}
false
2> Q = queue:in(0, queue:new()).
{[0],[]}
3> queue:tail(queue:in_r(0, Q)).
{[],[0]}
```

QuickSpec provides a generator

Observation functions

`tail(in_r(X, Q)) /= Q`, because the two queues might have different representations!

But the queues should have the same contents.

Let's compare the contents of the queues instead of the representations.

```
observe(X, queue) →
  queue:to_list(X).
```

The mystery of the missing law (again)

QuickSpec prints

```
is_empty(in(X,Q2)) == is_empty(in(X,Q))
is_empty(in(Y,Q)) == is_empty(in(X,Q))
```

Why not simply

```
is_empty(in(X,Q)) == false?
```

Missing equations (again)

- If an equation doesn't get printed, either
  - It has a too deeply-nested expression
  - **It uses some function that's not in your signature**
  - It's false!
- Our signature doesn't have false in it!
- Answer: add all the boolean operations to the signature

Nice laws

- `daeh(reverse(Q)) == head(Q)`

  `tail(reverse(Q)) == reverse(lait(Q))`

  - Symmetry laws: whatever head does to the front of a queue, daeh does to the back of the queue
- `in_r(X,in(Y,Q)) == in(Y,in_r(X,Q))`
  - Adding to different ends of the queue doesn't interfere
- `in_r(head(Q),tail(Q)) == in(daeh(Q),lait(Q))`
  - Both equal to Q, but not if Q is empty!

Testing against a model

Add to_list and some list functions to the signature:

```
[X|to_list(Q)] == to_list(in_r(X,Q))

to_list(Q) ++ ([X]) == to_list(in(X,Q))
hd(to_list(Q)) == head(Q)
tl(to_list(Q)) == to_list(tail(Q))

to_list(reverse(Q)) == reverse(to_list(Q))
```

Tells you what happens when the queue is represented by a list...

Testing against a model

Add to_list and some list functions to the signature:

```
[X|        Q ] ~=         in_r(X,Q)

        Q  ++ ([X]) ~=          in(X,Q)
hd(        Q ) ~= head(Q)
tl(        Q ) ~= tail(Q)

       reverse(Q)  ~= reverse(      Q )
```

A complete specification!

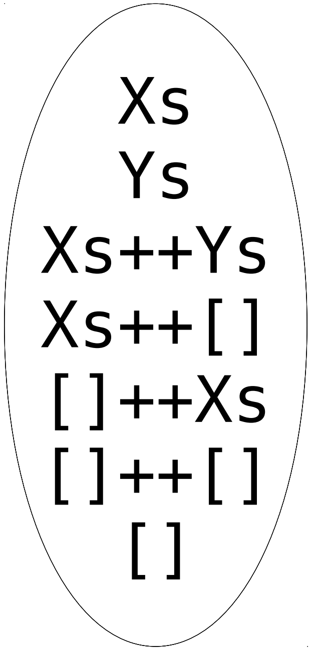
So, in summary...

- QuickSpec isn't magic
  - You need to think if you want the best specification
- Still, the results can be quite illuminating
- Good for
  - Getting properties for free
  - Understanding other people's code
  - Finding bugs

How QuickSpec works

How does QuickSpec work?

Generate all expressions up to a given depth:

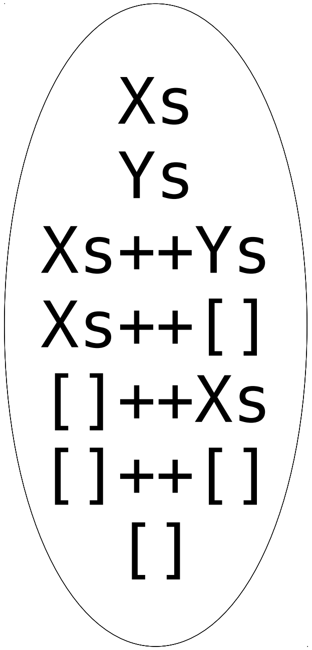


Xs
Ys
Xs++Ys
Xs++[]
[]++Xs
[]++[]
[]

(But picture thousands of expressions instead of 7)

How does QuickSpec work?

Then test which expressions are equal.

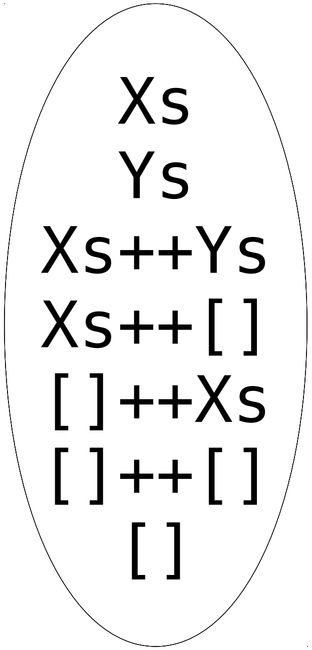


Xs
Ys
Xs++Ys
Xs++[ ]
[ ]++Xs
[ ]++[ ]
[ ]

To begin with, QuickSpec assumes that everything is equal.

How does QuickSpec work?

Try a random test case.

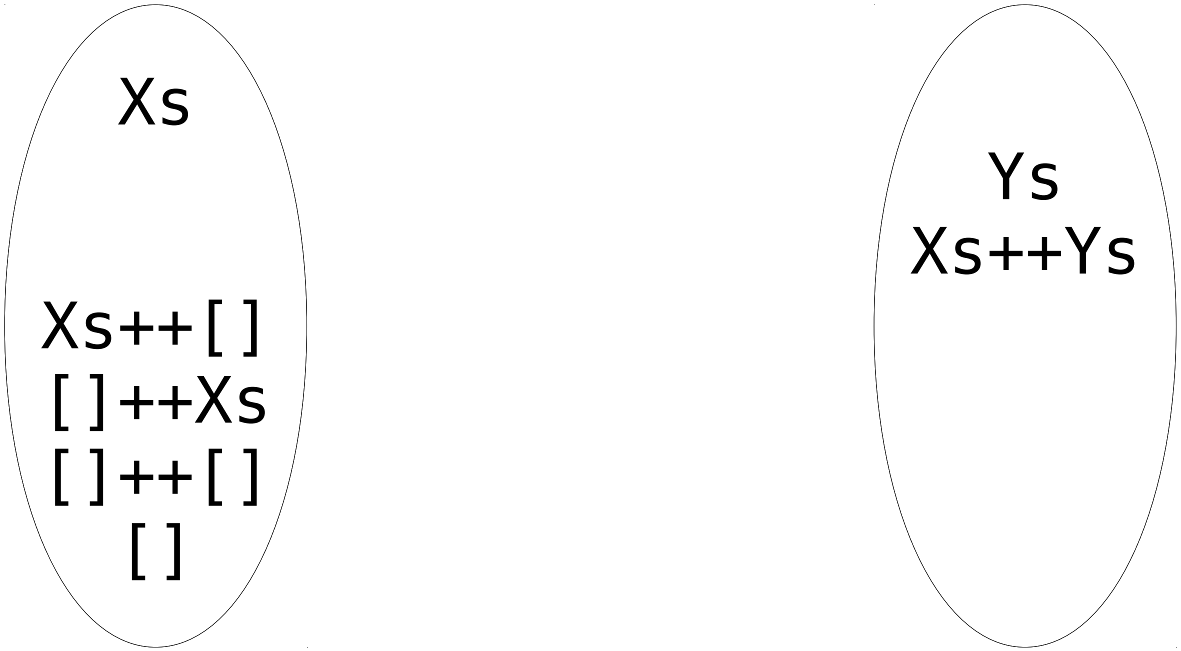


Xs
Ys
Xs++Ys
Xs++[]
[]++Xs
[]++[]
[]

```
Xs = [], Ys = [1]
```

How does QuickSpec work?

Try a random test case.



Xs

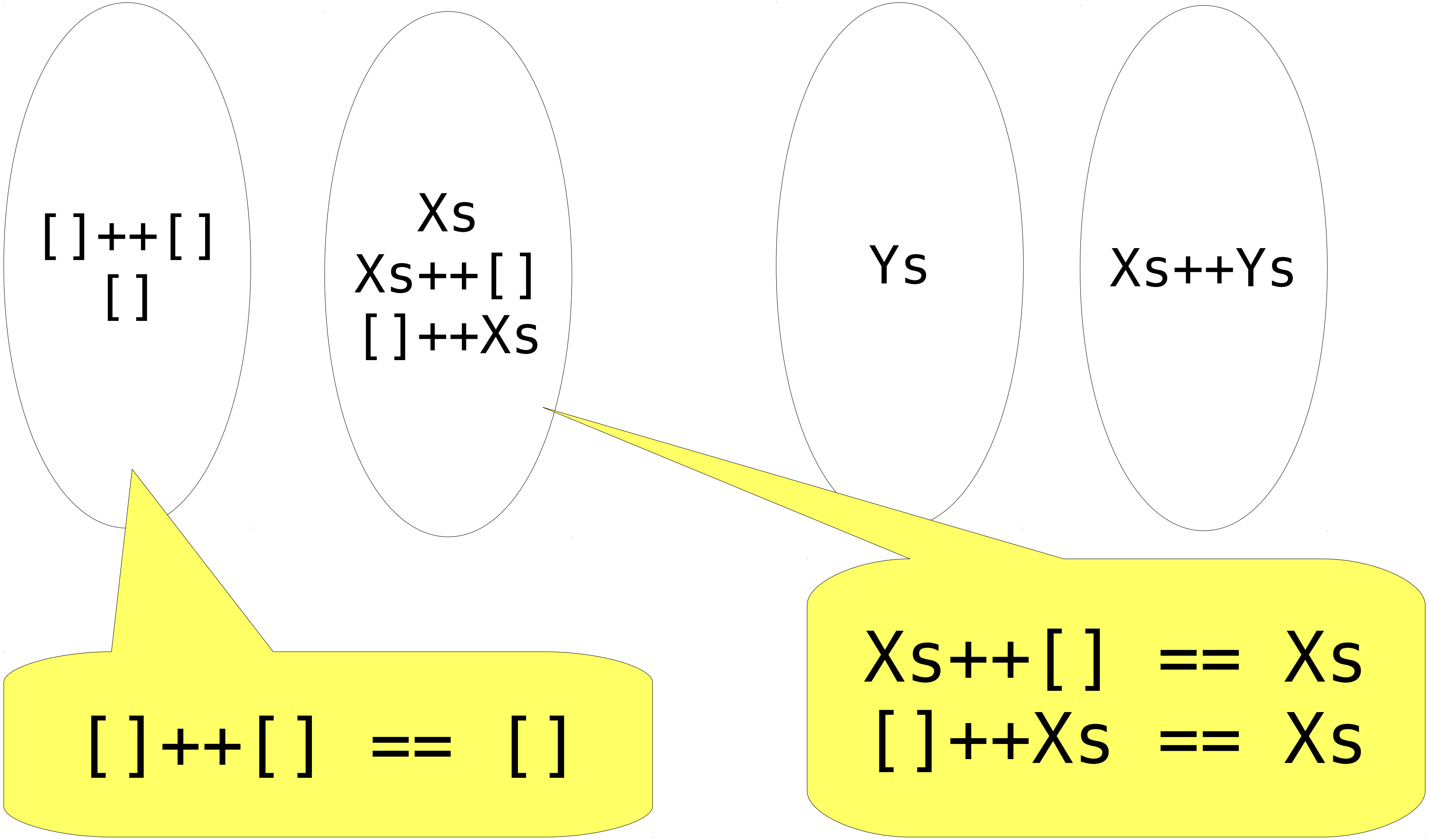
Xs++[]
[]++Xs
[]++[]
[]

Ys
Xs++Ys

`Xs = [1], Ys = []`

How does QuickSpec work?

Print out equations:



[]++[]
[]

Xs
Xs++[]
[]++Xs

Ys

Xs++Ys

[]++[] == []

Xs++[] == Xs
[]++Xs == Xs

but

Some numbers

- A binary heaps example:
  - 5392 terms generated
  - 3653 equivalence classes
  - **1739** valid equations
  - **27** equations printed
- How do we get from 1739 to 27?

Pruning

```
Xs++[]  ==  Xs
[]++Xs  ==  Xs
```

~~`[]++[]  ==  []`~~

Some equations are redundant; don't print them.

But it's really hard to tell if an equation is redundant!

We borrow some magic data structures from the nice theorem proving people.

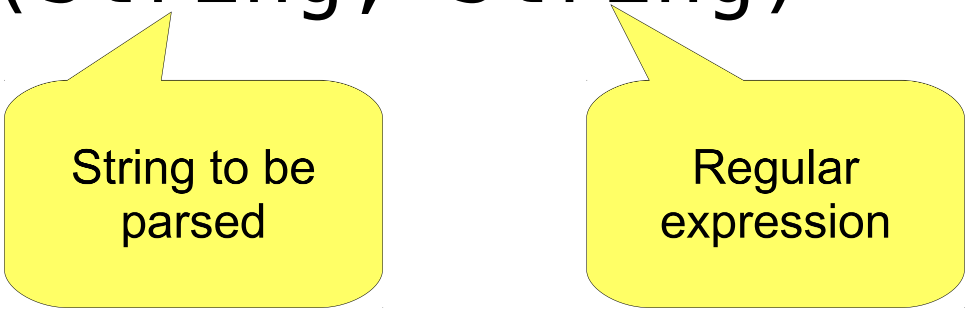
Regular expressions

Things that match strings.

- abc matches the string “abc”
- ab*c matches “abbbbbbc”
- a(b|c)* matches “abccccbbc”
- ab+ matches “abb” but not “a”
- a? matches only “a” or “”

Regular expression functions

`run(string, string) → boolean`



String to be parsed

Regular expression

e.g. `run(“aaaa”, “a*”)`

Too unstructured!

Regular expression operators

```
run(string, regex) → boolean
star(regex) → rege
char(c  r) → regex
any_
conc              ) → regex
choice(regex, regex) → regex
```

A regex is still a string really

```
star(R) ->
  “(“ ++ R ++ “)*”.
```

For example,

```
concat(char($a), star(char($b)))
```

Are two regular expressions equal?

Easiest way to find out: test on random input

First try:

```
observe(R, regex) →
    re:run(..., R).
```

Are two regular expressions equal?

Easiest way to find out: test on random input

Second try:

```
observe(R, regex, S) →
    re:run(S, R).

context() →
    list(char()).
```

OK, let's try it out!

```
2> laws:laws(re_sig).
Classifying terms of depth 0... 2 terms....
2 classes.
Classifying terms of depth 1... 10 terms....
10 classes.
Classifying terms of depth 2... 78 terms.......
<<computer goes into a sulk>>
```

What could be wrong?

The killer regular expression

```
2> re:run("abc", "((a|())+)+").
<<computer goes into a sulk>>
```

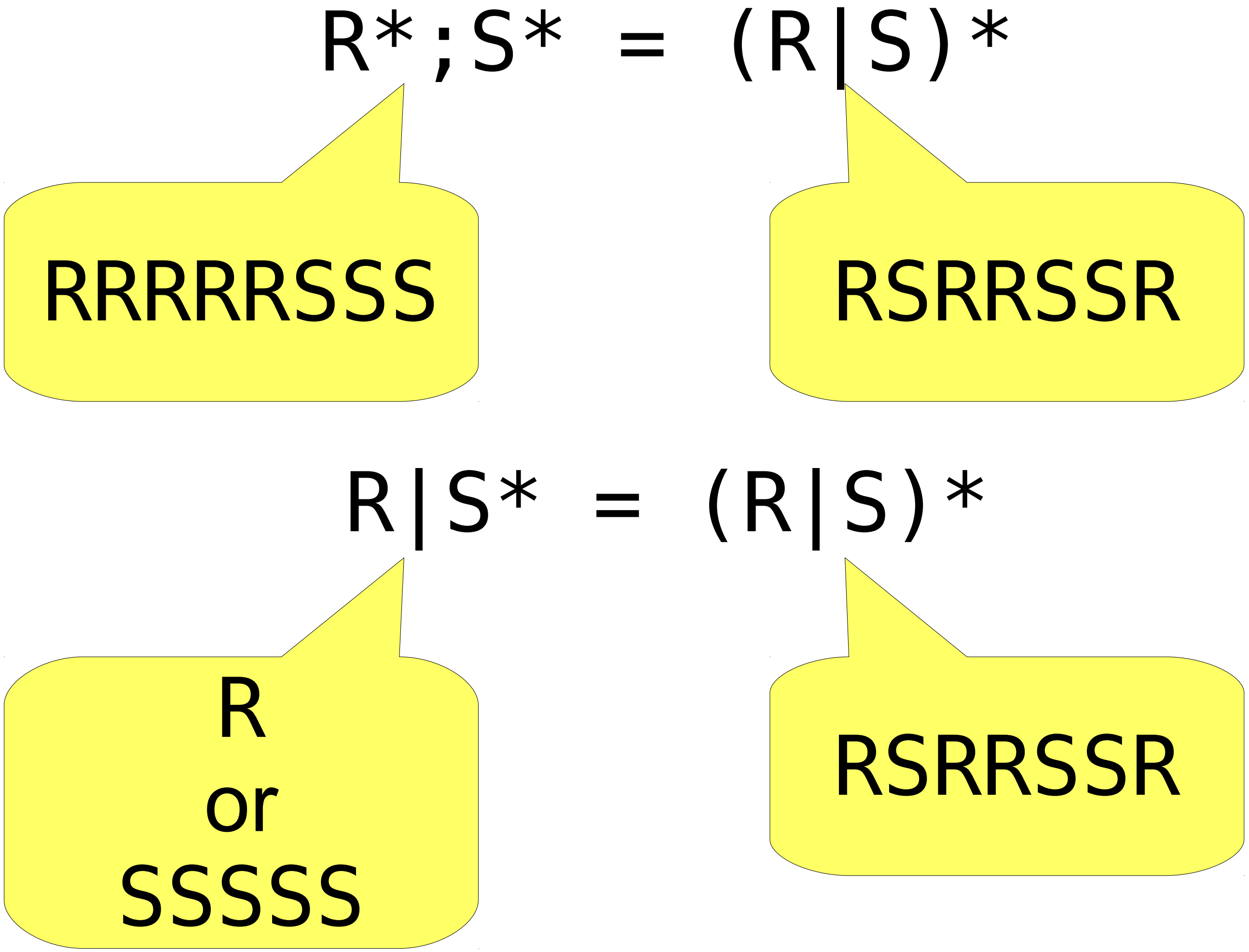
What to do?

- Fix the regular expression library
- Avoid generating iffy regular expressions
- Write my own regex library

DEMO

nfa_re_sig.erl

It's a bug!



```
R*;S* = (R|S)*
```

RRRRRSSS

RSRRSSR

```
R|S* = (R|S)*
```

R
or
SSSSS

RSRRSSR

Future improvements

Conditional equations: arrays

```
get(I,set(I,X,A)) == X

get(J,set(I,X,new())) == get(I,set(J,X,new()))

set(I,X,set(I,Y,A)) == set(I,X,A)

set(J,X,set(I,X,A)) == set(I,X,set(J,X,A))
```

We would rather get

```
I /= J =>
  set(J,Y,set(I,X,A)) == set(I,X,set(J,Y,A))
```

Testing imperative programs

- Imperative queues...
  - empty(), empty() == empty()
  - X = front(), Y = front(), add(Z)
==
X = front(), add(Z), Y = front()
- Ostrich approach: pretend that the program is a pure function (from input state to output state) and everything works as before
- In practice, this puts a lot of stress on QuickSpec

Want to try it out?

http://tinyurl.com/quickspec-talk

Install QuickCheck mini from your CD (it's free!) or get it from http://quviq.com/downloads/eqcmini.zip

Comes with examples and tutorial slides and a paper