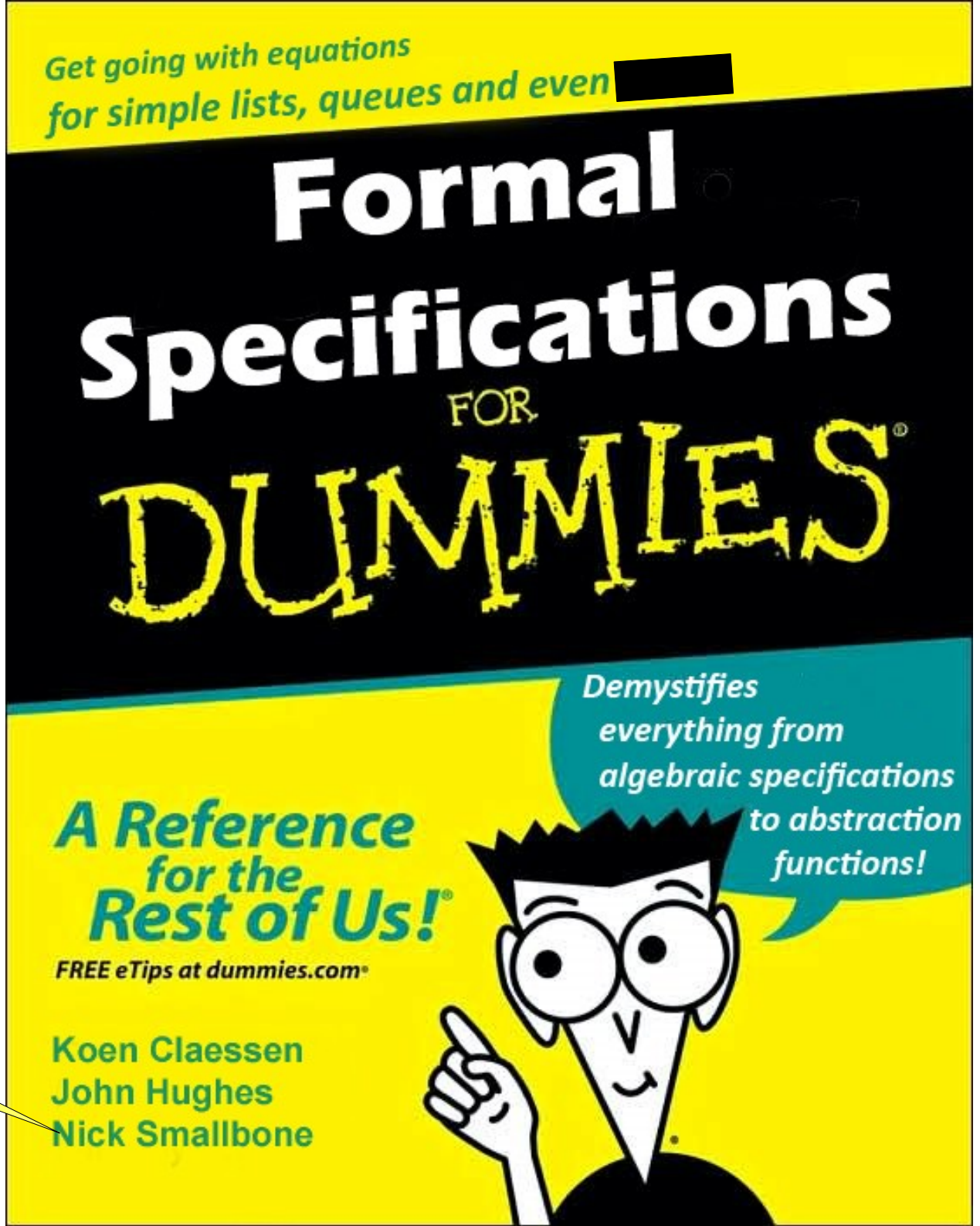

Get going with equations
for simple lists, queues and even [redacted]
Formal Specifications
FOR
DUMMIES®
Demystifies everything from algebraic specifications to abstraction functions!
A Reference for the Rest of Us!®
FREE eTips at dummies.com®
Koen Claessen
John Hughes
Nick Smallbone
(that's me)

A simple API

Lists – append, cons and nil:

```
list() ++ list() → list()
[int()|list()] → list()
[] → list()
```

How can we test these functions?

Write properties!

Properties in all their wonder

```
prop_appendnil() →
   ?FORALL(Xs, list(int()),
   Xs ++ [] == Xs).

1> eqc:quickcheck(prop:prop_appendnil()).
.....................................
.....................................
OK, passed 100 tests
```

But how do we know what properties to write?

QuickSpec to the rescue!

DEMO

listsig.erl

What could you use this for?

Formulating properties

Understanding someone else's code

Finding bugs

Pure functions only!

How does QuickSpec work?

I'm not telling you yet :)

Equations are tested on random input

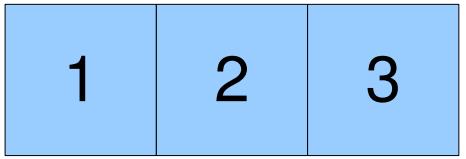
Equations relate *simple enough* expressions built from functions and variables

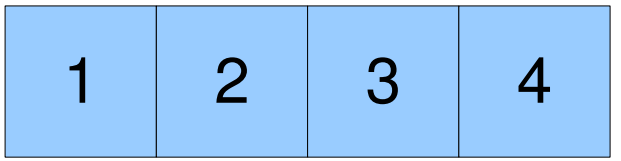
```
Xs                          depth 1
Xs ++ []                    depth 2
Xs ++ (Ys ++ Zs)            depth 3
Xs ++ ((Ys ++ Zs) ++ Zs)    depth 4
```

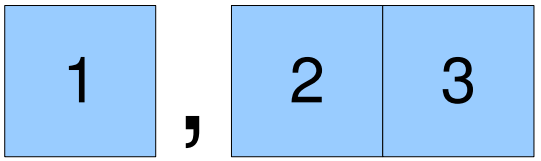
Too complicated

Completeness: all true equations about simple enough expressions are found.

The queue API

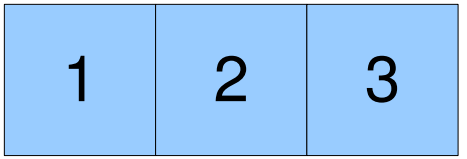
Q = 

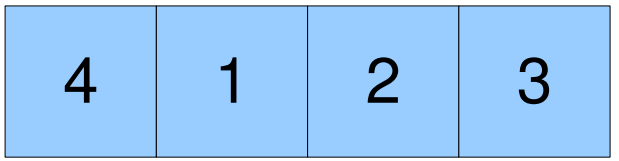
in(4, Q) = 

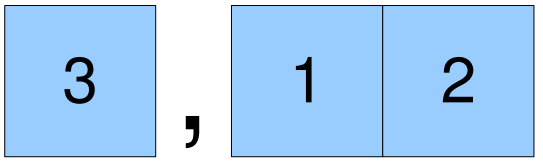
{head(Q), tail(Q)} = { 

}

The queue API, in reverse

Q = 

in_r(4, Q) = 

{daeh(Q), liat(Q)} = { 

}

Or
lait(Q)!

DEMO

queuesig.erl

What's wrong?

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

But the queues should have the same contents.

In other words,

```
to_list(lait(in(X, Q)) ==
    to_list(Q).
```

Why don't we compare those instead?

What's wrong?

We want to see this law:

```
is_empty(new()) == true()
```

But QuickSpec doesn't know about `true`!

What's wrong?

We want to see this law:

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

In that case, we need the `reverse` function on lists!

How did we get nice laws?

Compare the elements of a queue, not its internal representation:

```
observe(Q, queue) ->
    queue:to_list(Q).
```

Add `reverse` so that we see the symmetry between `tail` and `lait`.

Use lists as a model of queues.

What on earth is this?

```
bar(X,foo()) == apa()
bar(X,monkey(X,P,I)) == P
bar(Y,monkey(X,P,foo())) == bar(X,monkey(Y,P,foo()))
bar(X,monkey(Y,apa(),foo())) == apa()
monkey(X,P,monkey(X,Q,I)) == monkey(X,P,I)
monkey(Y,P,monkey(X,P,I)) == monkey(X,P,monkey(Y,P,I))
```

...and why does the highlighted law hold?

...it's **arrays**!

```
get(I,new()) == default_element()
get(I,set(I,X,A)) == X
get(J,set(I,X,new())) == get(I,set(J,X,new()))
get(I,set(J,default_element(),new())) == 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))
```

What on earth is this?

```
bar(X,foo()) == apa()
bar(X,monkey(X,P,I)) == P
bar(Y,monkey(X,P,foo())) == bar(X,monkey(Y,P,foo()))
bar(X,monkey(Y,apa(),foo())) == apa()
monkey(X,P,monkey(X,Q,I)) == monkey(X,P,I)
monkey(Y,P,monkey(X,P,I)) == monkey(X,P,monkey(Y,P,I))
```

...and why does the highlighted law hold?

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

Regular expressions

Things that match strings.

abc matches the string “abc”

ab*c matches “abbbbbbbc”

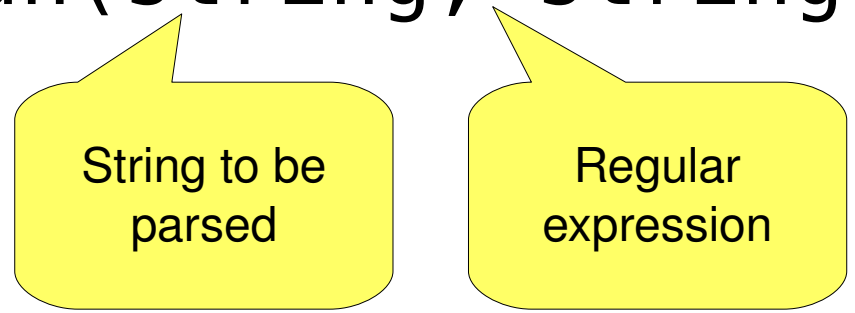
a(b|c)* matches “abccccbbc”

ab+ matches “abb” but not “a”

a? matches only “a” or “”

Regular expression functions

```
run(string, string) → boolean
```



Too unstructured!

Regular expression operators

```
run(string, regex) → boolean
star(regex) → regex
char(ch  ) → regex
any_c
conca                    → 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?

DEMO

re_sig.erl

The killer regular expression

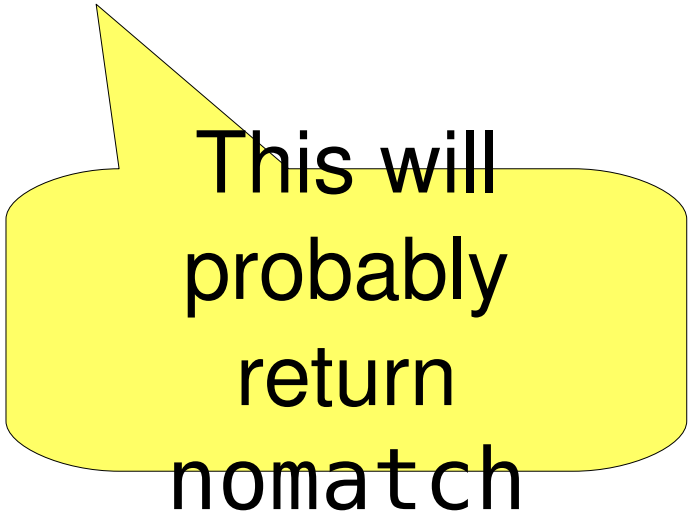
```
2> re:run("abc", "((a|())+|a)+").
<<computer goes into a sulk>>
```

For now: remove * and +.

Are two regular expressions equal?

Input data is too random!

```
re:run("cpjsd¬!!££$", R).
```



Test on strings of as and bs.

The story so far

Well, it all seems to work OK...

But what about *?

I don't want to fix PCRE

In principle we could avoid dodgy regular expressions

Let's switch to a non-breaky regular expression library instead :)

DEMO

nfa_re_sig.erl

It's a bug!

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

RRRRRSSS

RSRRSSR

R|S* = (R|S)*

R
or
SSSSS

RSRRSSR

What could we use this for

Understanding a library (fixed-point arithmetic)

Finding bugs

Extracting tests

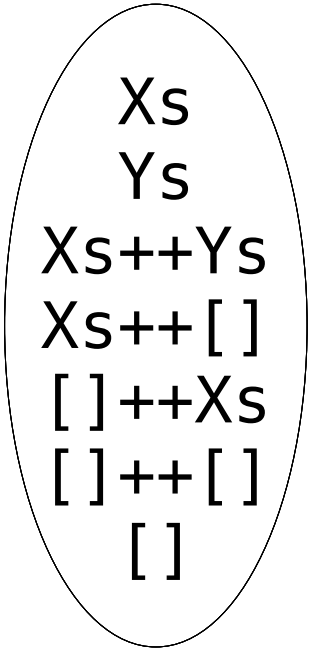
Optimisation?

The paper

- Lots of examples (Erlang and Haskell)
- Case studies: binary heaps and fixed-point arithmetic
- How everything works

How does QuickSpec work?

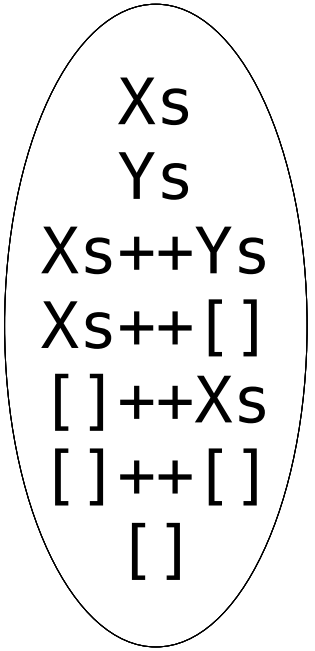
Generate all expressions up to a given depth:



We assume that two expressions are equal until we find a counterexample.

How does QuickSpec work?

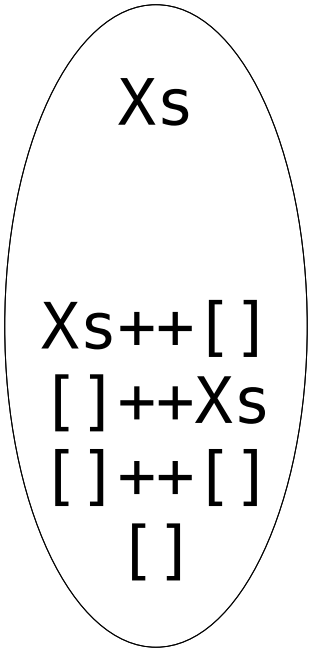
Pick some random values for testing:

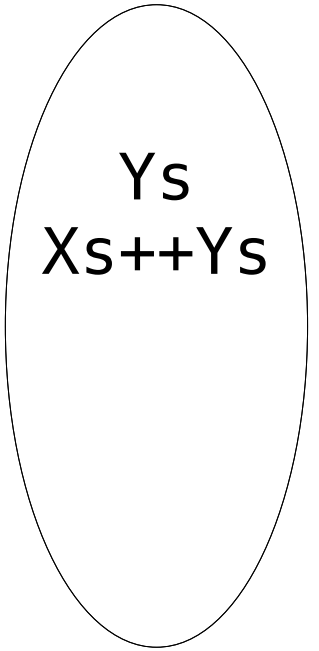



```
Xs = [], Ys = [1]
```

How does QuickSpec work?

Pick some random values for testing:

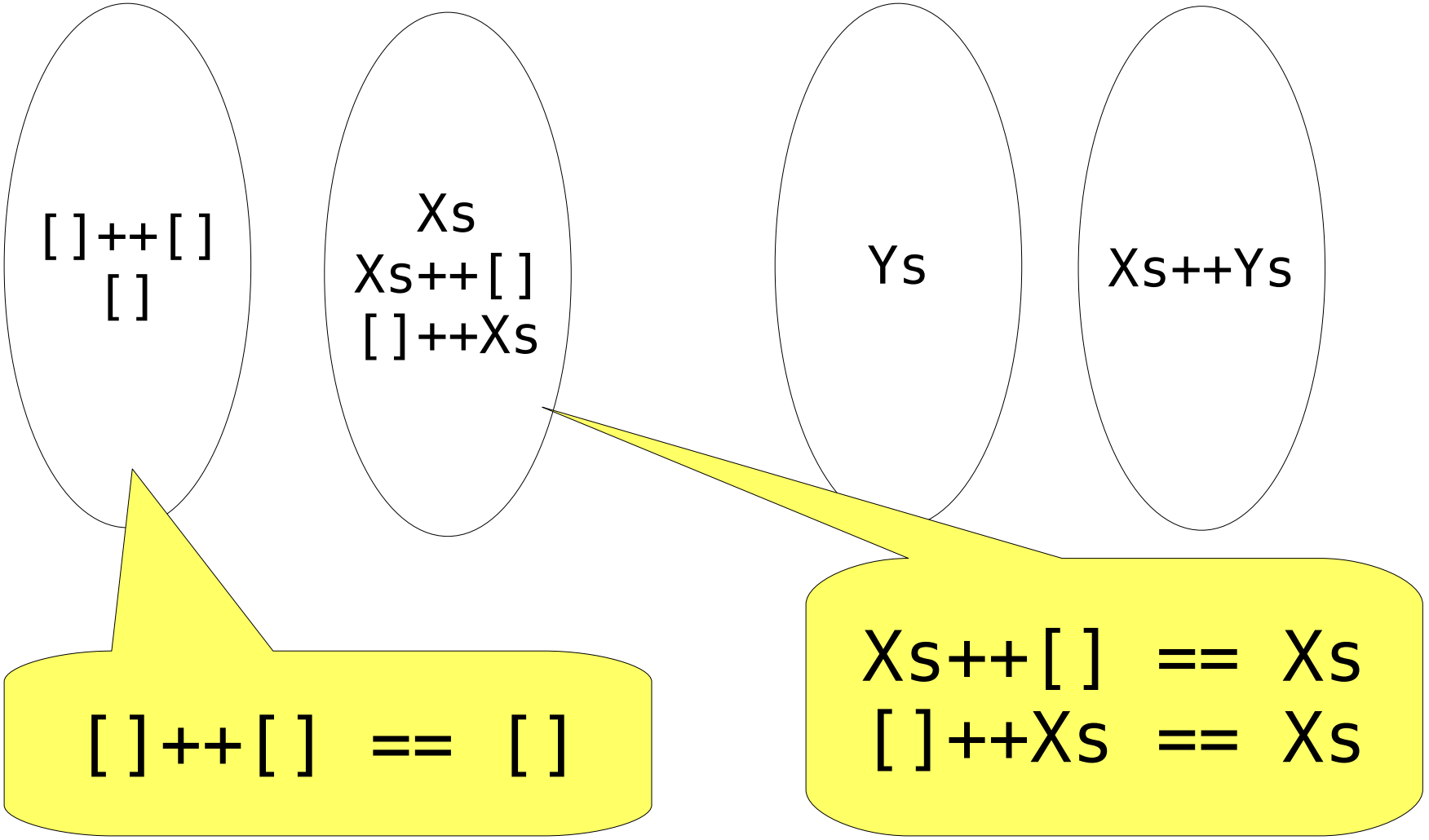





```
Xs = [1], Ys = []
```

How does QuickSpec work?

Produce some equations:



How does QuickSpec work?

Filter out the redundant equations (then print the rest):

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

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

This is the hard bit!

Caveats

Your functions need to be pure and terminating

You might need to include some auxiliary functions

`observe` is quite delicate

```
observe(Q, queue) →
    queue:len(Q).
in_r(X,Q) “==” in_r(Y,Q)
head(in_r(X,Q)) /= head(in_r(Y,Q))
```

Should be able to spot this automatically.

What's to come next?

Testing imperative modules

Conditional equations:

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

(demo: Haskell arrays)