

**Finding loop invariants
with QuickSpec
(work in progress)**

Nick Smallbone

DEMO
Lists.hs

How it works

Enumerate terms, starting from the simplest:

- $xs, ys, zs, [], xs++[], ys++[], \dots,$
 $xs++(ys++zs), \dots, xs++(xs++ys), \dots, (xs++ys)++zs, \dots$

The goal: discover equations between these terms

The algorithm: for each term, work out: *is it equal to a term we've already seen?*

- If we can *reason* that it's equal to a term we've already seen, using the equations discovered so far, discard the term
- Otherwise, if *testing* shows it's equal to a term we've already seen, then we've discovered an equation!
- Otherwise, just add it to the set of seen terms

How it works

Initially: no seen terms or discovered laws

Seen terms

Discovered laws

How it works

First few terms: **xS** , **yS** , **zS** , **$[\]$**

Not equal to each other

Add to seen terms!

Seen terms

xS yS zS $[\]$

Example test case:

$xS = [1, 2, 3]$

$yS = [2, 1]$

$zS = [2]$

Discovered laws

How it works

Next term: $[\]++xs$

Testing reveals it's equal to the already-seen term xs

Hooray, a law!

Seen terms

xs ys zs $[\]$

Example test case:

$xs = [1, 2, 3]$

$ys = [2, 1]$

$zs = [2]$

Discovered laws

How it works

Next term: $[]++XS$

Testing reveals it's equal to the already-seen term XS

Hooray, a law!

Seen terms

XS YS ZS $[]$

Discovered laws

$[]++XS = XS$

How it works

Next term: $[]++ys$

The laws imply it's equal to the already-seen term ys

Throw it out!

Seen terms

xS **ys** zS $[]$

Discovered laws

$[]++xS = xS$

How it works

Next term: **$xs++(ys++zs)$**

Not equal to an already-seen term

Add to seen terms!

Seen terms

xs ys zs $[]$

$xs++(ys++zs)$

Discovered laws

$[]++xs = xs$

How it works

Next term: **$xs++(xs++ys)$**

Not equal to an already-seen term

Add to seen terms!

Seen terms

xs ys zs $[]$

$xs++(ys++zs)$

$xs++(xs++ys)$

Discovered laws

$[]++xs = xs$

How it works

Next term: **$(xs++ys)++zs$**

Equal to $xS++(yS++zS)$ by testing

Hooray, a law!

Seen terms

xS yS zS $[]$

$xS++(yS++zS)$

$xS++(xS++yS)$

Discovered laws

$[]++xS = xS$

How it works

Next term: **$(xs++ys)++zs$**

Equal to $xs++(ys++zs)$ by testing

Hooray, a law!

Seen terms

xs ys zs $[]$

$xs++(ys++zs)$

$xs++(xs++ys)$

Discovered laws

$[]++xs = xs$

$(xs++ys)++zs =$
 $xs++(ys++zs)$

How it works

Next term: **$(xs++xs)++ys$**

Equal to $xs++(xs++ys)$ by the discovered laws

Throw it out!

Seen terms

xs ys zs $[]$

$xs++(ys++zs)$

$xs++(xs++ys)$

Discovered laws

$[]++xs = xs$

$(xs++ys)++zs =$
 $xs++(ys++zs)$

How it works

Next term: **$(xs++xs)++ys$**

Equal to $xs++(xs++ys)$ by the discovered laws

Throw it out!

Seen terms

xs ys zs $[]$

$xs++(ys++zs)$

$xs++(xs++ys)$

Discovered laws

$[]++xs = xs$

$(xs++ys)++zs =$
 $xs++(ys++zs)$

QuickSpec on imperative code

Our aim is to discover assertions that hold at a particular point in the program

- e.g., postcondition, loop invariant

The idea:

- generate random inputs to the program
- run the program to the correct point and observe the program state
- discover properties that hold in all observed program states

QuickSpec on imperative code

1. Generate random inputs satisfying precondition

$\text{arr} = \{0,1,2,3,4\}, x = 3$

$\text{arr} = \{0,1,2,3,4\}, x = 8$

$\text{arr} = \{0,1,2,4,5\}, x = 3$

QuickSpec on imperative code

2. Run the program up to the desired point

arr = {0,1,2,3,4}, x = 3, **lo = 2, hi = 4**

arr = {0,1,2,3,4}, x = 8, **lo = 3, hi = 5**

arr = {0,1,2,4,5}, x = 3, **lo = 2, hi = 4**

QuickSpec on imperative code

3. Infer properties that hold in all these program states

$\text{arr} = \{0,1,2,3,4\}, x = 3, \text{lo} = 2, \text{hi} = 4$

$\text{arr} = \{0,1,2,3,4\}, x = 8, \text{lo} = 3, \text{hi} = 5$

$\text{arr} = \{0,1,2,4,5\}, x = 3, \text{lo} = 2, \text{hi} = 4$

e.g.:

arr is sorted

$\text{lo} \leq \text{hi}$

if x is found in arr then it can be found in $\text{arr}[\text{lo}..\text{hi})$

QuickSpec on imperative code

Much like normal QuickSpec, but:

- Terms can contain *program variables* as well as other functions (such as array lookup)
- Test cases include a *program state*, generated by running the program on a random input

Otherwise the same!

Binary search invariant

Boolean
formulas!

$\forall i. (\text{arr}[i] = x \rightarrow \exists j. (\text{lo} \leq j \wedge j < \text{hi} \wedge \text{arr}[j] = x))$

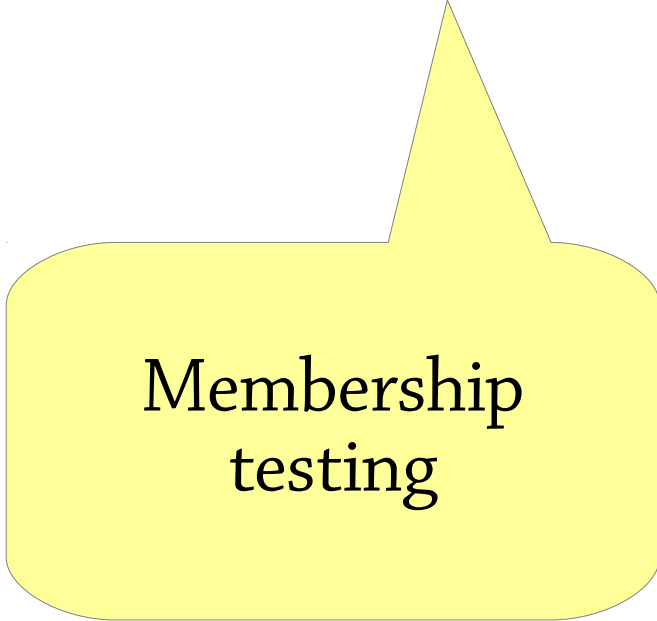
Preconditions!

Quantifier
alternation!


Binary search, QuickSpec style

Instead of complicated formulas, *simple* formulas over an *expressive* term language

$$x \in \text{arr} \leftrightarrow x \in \text{arr}[\text{lo}..\text{hi})$$



Membership
testing



Slicing

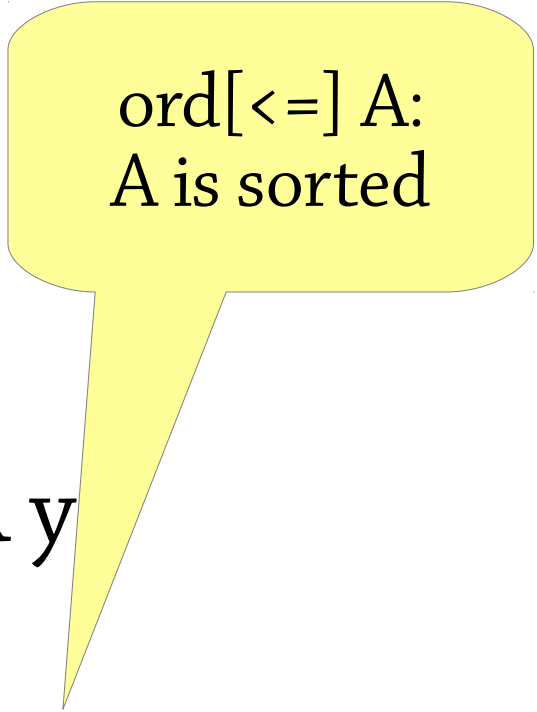
Reynolds: "Reasoning About Arrays"

Array operations

- $A[i], A[i := x]$
- $\text{length}(A)$
- $\text{image}(A)$
- $A|X$ (restriction)

Set operations

- $X \cup Y$
- $[i..j)$
- $\{x\}$



$\text{ord}[\leq] A$:
A is sorted

Pairwise relations:

$X R^* Y$ means $\forall x \in X, y \in Y. x R y$

Ordering:

$\text{ord}[R] A$ means $\forall i, j. i < j \rightarrow A[i] R A[j]$

Binary search, Reynolds style

$$(\{x\} \neq^* \text{arr}) = (\{x\} \neq^* (\text{arr} \mid [\text{lo}..\text{hi}]))$$

x not in arr

x not in
slice of array

demo1
demo2part1

Generating relevant invariants

Counterexample-directed invariant generation

When we fail to verify

```
while {I} E do ... end {P}
```

how should we strengthen the invariant I?

Well-known approach:

- look for a program state S which is a counterexample to $I \wedge \neg E \rightarrow P$
- strengthen I so that S does not satisfy it (explain why S is unreachable)

QuickSpec can do this too!

- Find S (currently by random generation)
- Discover P = properties that hold in real executions
- and P_s = properties that hold in S
- Then choose a property that occurs in P but not in P_s , and add it to the invariant

Counterexample-directed invariant generation

For binary search we got:

$lo \leq hi$

$ord[\leq](arr)$

$member(x, arr) = member(x, arr \mid [lo..hi))$

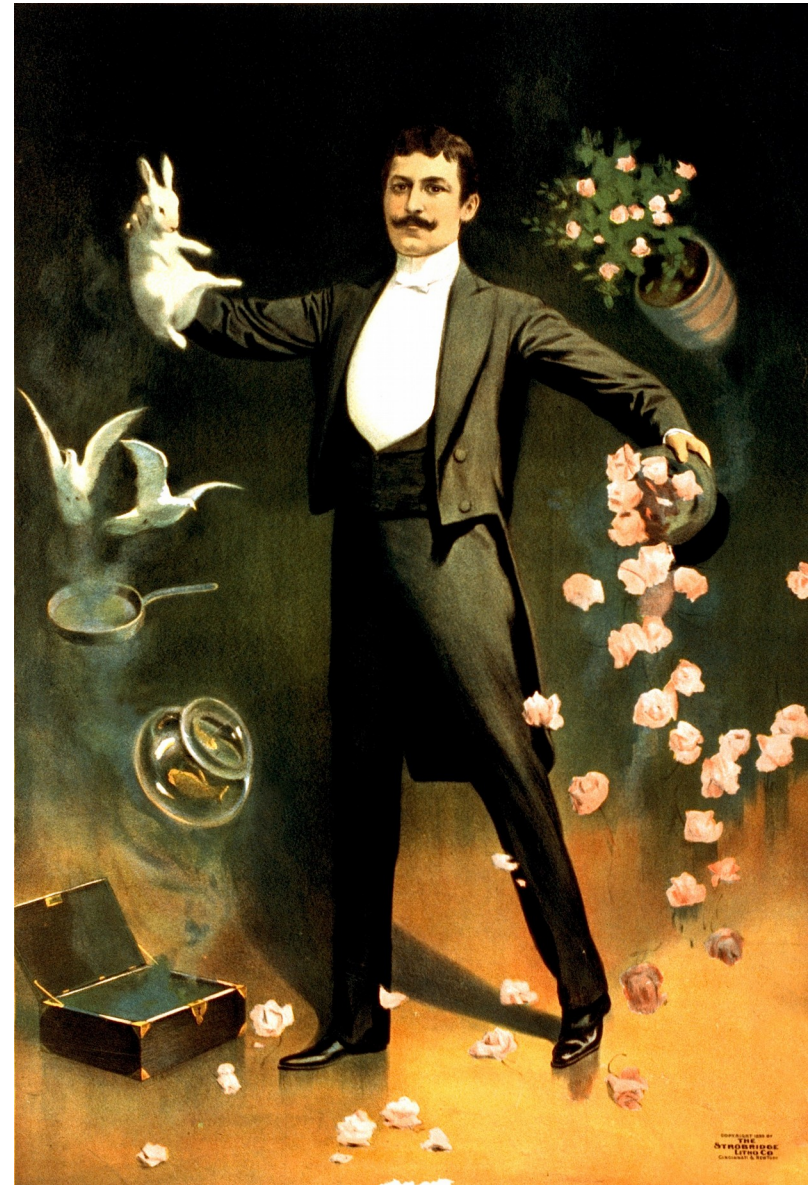
(...and some false properties...)

Psychic testing

A party trick. The conjuror takes:

- A buggy program
- A passing test suite

The conjuror reads the programmer's mind to discover what the program was *supposed* to do, and finds the bugs.



Psychic testing – the reveal

found = member(x, arr)
is false here

- A passing test suite

found = member(x, arr)
is true here

what the program
was *supposed* to do,
and finds the bugs.

Find two sets of
properties:

- P – the properties which really hold
- P_{test} – the properties which hold when test data is drawn *only from the test suite*

Report properties
which are in P_{test}
but not in P

Summary

Play to QuickSpec's strengths: expressive terms instead of complex formulas

Can generate *relevant* loop invariants

Need a not-a-toy version to evaluate properly!

- Something cleverer than random testing
- Built-in reasoning about background theories
- Use something that can discover clauses e.g. Koen's TurboSpec?