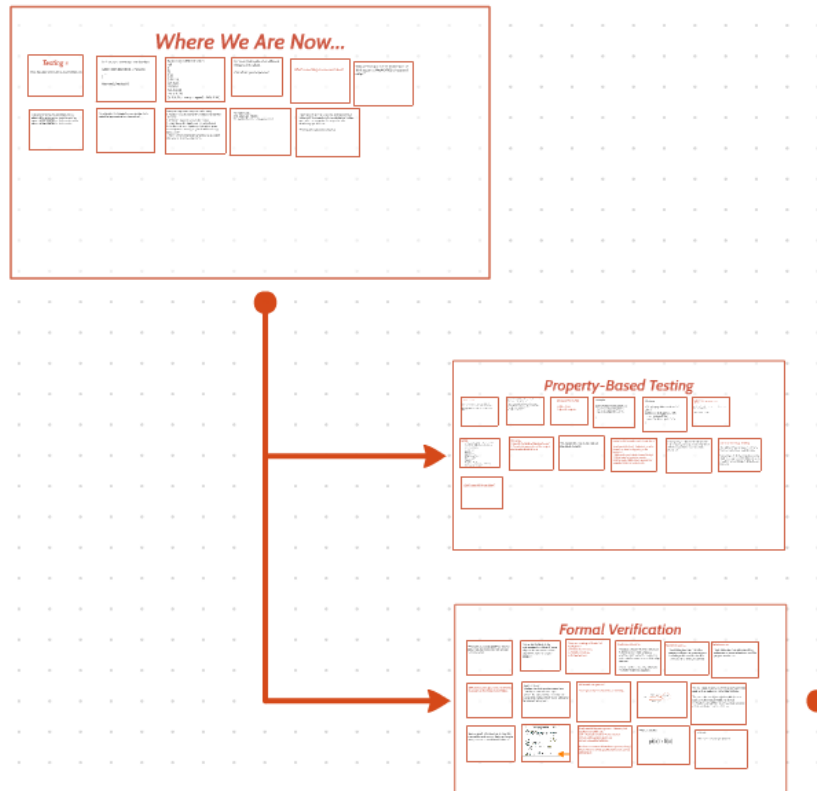
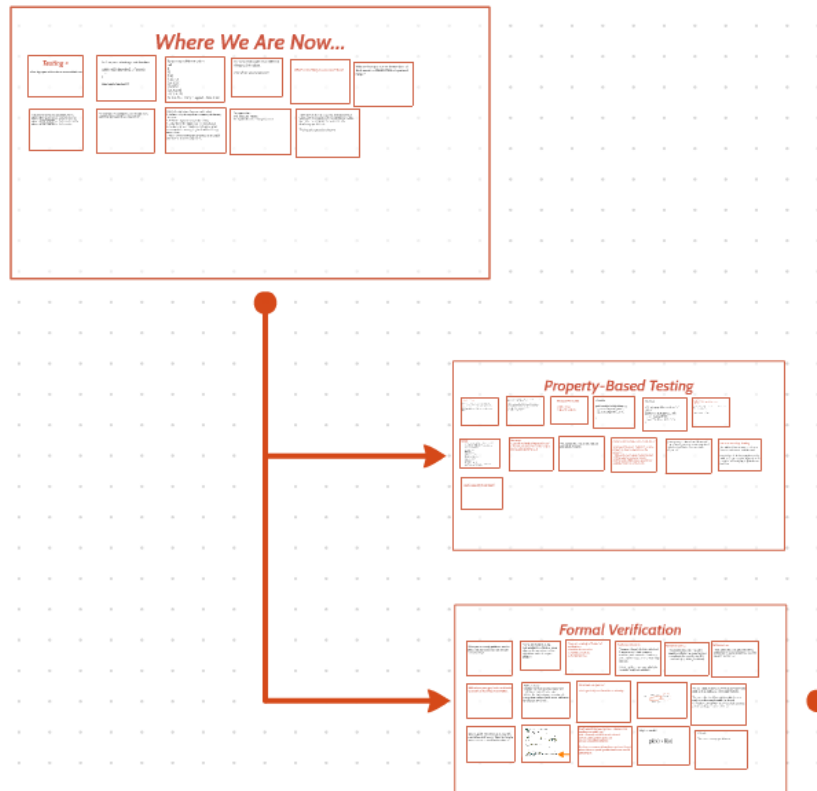


Property-Based Testing and Formal Verification



Property-Based Testing and Formal Verification



Where We Are Now...

Testing =

Checking expected behavior vs observed behavior

Let's say we are testing a sort function.

```
public int[] billSort(int[] arr){  
    ...  
}
```

How would we test it?

Try passing in different values:

```
null  
[]  
[1]  
[1,2]  
[1,2,3,4,5]  
[5,4,3,2,1]  
[0,0,0,0]  
[9,3,4,2,1,6]  
[-9, 2, 4, -3]  
[9, 4, 2, 19, ... (many integers) ... 982, 4, 23]
```

But wow, that's quite a few different things to think about.

And what if you forget one?

What's something else we could check?

Why not hop up a level of abstraction and think about the PROPERTIES of input and output?

Instead of checking expected behavior vs observed behavior per se, we can check the expected PROPERTIES of the behavior vs the observed PROPERTIES of the behavior.

For example, what properties do we expect of a sorted list compared to a list passed in?

1. Output array same size as passed-in array
2. Values in output array always increasing or staying the same
3. Value in output array never decreasing
4. Every element in input array is in output array
5. No element not in input array is in output array
6. Idempotent - running it again should not change output array
7. Pure - running it twice on same input array should always result in same output array

For input values:
1. All values are integers.
2. Length is 0 or more integers, or null

Now that we have the properties of expected input values, and the properties of expected output values, we can let the computer do the grunt work of developing specific tests.

This is called property-based testing.



Testing =

Checking expected behavior vs observed behavior

Let's say we are testing a sort function.

```
public int[] billSort(int[] arrToSort) {  
    ....  
}
```

How would we test it?

Try passing in different values:

null

[]

[1]

[1,2]

[1,2,3,4,5]

[5,4,3,2,1]

[0,0,0,0]

[9,3,4,2,1,6]

[-9, 2, 4, -3]

[9, 4, 2, 19, ... (many integers) ... 982, 4, 23]

But wow, that's quite a few different things to think about.

And what if you forget one?

What's something else we could check?

Why not hop up a level of abstraction and think about the **PROPERTIES** of input and output?

Instead of checking expected behavior vs observed behavior per se, we can check the expected PROPERTIES of the behavior vs the observed PROPERTIES of the behavior.

For example, what properties do we expect of a sorted list compared to a list passed in?

1. Output array same size as passed-in array
2. Values in output array always increasing or staying the same
3. Value in output array never decreasing
4. Every element in input array is in output array
5. No element not in input array is in output array
6. Idempotent - running it again should not change output array
7. Pure - running it twice on same input array should always result in same output array

For input values:

1. All values are integers
2. Length is 0 or more integers, or null

Now that we have the properties of expected input values, and the properties of expected output values, we can let the computer do the grunt work of developing specific tests.

This is called *property-based testing*.

Property-Based Testing

A New Kind of Testing

Presented at KFP 2010 in the paper "QuickCheck & LightenUp: Test for Random Testing of Haskell Programs"
<http://www.cs.tu.edu/~cs/cv252/archives/papers/hughes/quick.pdf>

Much more popular in functional programming world than the imperative world, for a variety of reasons:

1. Pure functional code is much easier to test in this manner
2. Organizations using imperative languages less familiar with it
3. Requires some mathematical background
4. Considered "weird" by some
5. Considered "niche" by some

Not useful for testing:

1. Side effects
2. Specific outputs

Example:

```
public void printGlobalStats() {  
    System.out.print("Stat 1");  
    System.out.println(_stat1);  
    ...  
}
```

Example:

```
// Login page title should be "hi"  
@Test  
public void testLoginSaysHi() {  
    title = page.getTitle();  
    assertTrue(title.equals("hi"));  
}
```

Useful for testing:

A variety of inputs → specific kinds of outputs
Pure code

Since Haskell is "purely functional", with no side effects of any kind, this is a perfect match.

* Outside of monads, of course

NOTE:

The concept originated in functional programming but is still useful in imperative languages! There are a variety of QuickCheck-like testing solutions out there in different languages, e.g.

Java: jUnit-quickcheck
Ruby: randy
Scala: scalacheck
Python: pyunit-quickcheck
Node.js: node-quickcheck
Clojure: simple-check
C++: QuickCheck++
.NET: F#Check
Golang: Golang/QuickCheck
The only one I couldn't find is a version for PHP. A good item project for someone, perhaps!

Two steps:

1. Specify the kinds of inputs allowed
2. Specify the properties of the output that should ALWAYS hold

These properties that always hold are also called *invariants*.

QuickCheck then makes our test suite for us!

Think about the levels of abstraction we've jumped up since the beginning of the semester:

1. Write and execute tests (manual testing)
2. Write tests, let computer execute
3. Write what KINDS of tests we want, let computer write tests and execute

This is gaining traction outside the world of functional programming, as some aspects of FP leak into production systems (Scala, Clojure, etc.)

One more cool thing.. shrinking

If QuickCheck finds an issue, it will try to find the smallest version of that issue.

Say sorting with 4s doesn't work correctly. It will try to figure out that [4] breaks, even though it first found [5,6,2,1,4] falsifies an invariant.

Let's see it in action!

A New Kind of Testing

Presented at ICFP 2000 in the paper "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs"

<http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf>

Much more popular in functional programming world than the imperative world, for a variety of reasons

1. Pure functional code is much easier to test in this manner
2. Organizations using imperative languages less familiar with it
3. Requires some mathematical background
4. Considered "overkill" by some
5. Considered "weird" by some

Not useful for testing:

1. Side effects

2. Specific outputs

Example:

```
public void printGlobalStats() {  
    System.out.print('Stat 1');  
    System.out.println(_stat1);  
    ...  
}
```

Example:

```
// Login page title should be "hi"  
@Test  
public void testLoginSaysHi() {  
    title = page.getTitle();  
    assertTrue(title.equals("hi"));  
}
```

Useful for testing:

A variety of inputs -> specific kinds of outputs

Pure code

Since Haskell is "purely functional", with no side effects of any kind*, this is a perfect match.

* Outside of monads, of course.

NOTE:

The concept originated in functional programming but is still useful in imperative languages! There are a variety of QuickCheck-like testing solutions out there in different languages, e.g.

Java: junit-quickcheck

Ruby: rantly

Scala: scalacheck

Python: pytest-quickcheck

Node.js: node-quickcheck

Clojure: simple-check

C++: QuickCheck++

.NET: FsCheck

Erlang: Erlang/QuickCheck

The only one I couldn't find is a version for PHP. A good term project for someone, perhaps?

Two steps:

- 1. Specify the kinds of inputs allowed*
- 2. Specify the properties of the output that should ALWAYS hold*

These properties that always hold are also called *invariants*.

QuickCheck then makes our test suite for us!

Think about the levels of abstraction we've jumped up since the beginning of the semester:

- 1. Write and execute tests (manual testing)***
- 2. Write tests, let computer execute***
- 3. Write what KINDS of tests we want, let computer write tests and execute***

This is gaining traction outside the world of functional programming, as some aspects of FP leak into production systems (Scala, Clojure, etc.)

One more cool thing.. shrinking

If QuickCheck finds an issue, it will try to find the smallest version of that issue.

Say sorting with 4s doesn't work correctly. It will try to figure out that [4] breaks, even though it first found [5,6,2,1,4] falsifies an invariant.

Let's see it in action!

Formal Verification

When you absolutely, positively need to prove that your code is correct, you can formally verify it.

Formal verification is using mathematical models to prove or disprove the correctness of the algorithms and code of your program.

There are a variety of "levels" of verification -

1. Predictable execution
2. Partial correctness
3. Full correctness

Predictable Execution

The code is free of what we would call "runtime errors": that is, no array overflows, null pointer dereferencing, uninitialized variable access, division by zero, etc.

NB not that these are very unlikely, but
PROVEN TO BE IMPOSSIBLE.

Partial Correctness

Predictable execution, PLUS the program will give the correct answer according to the specification IF it terminates (yay Halting Problem!).

Full Correctness

Predictable execution, plus everything referenced in partial correctness, plus the program terminates.

Difficulty to prove gets harder and harder as you go up the levels of verification.

How is it done?

1. Before-the-fact: construct code from formally-proven subset of logic
2. After-the-fact: attempt to verify code using static analysis (much easier with some languages than others...)

What tools can you use?

You're probably familiar with one already...



There are tools to convert a Finite State Machine to code, or show code as a Finite State Machine.

There are other tools/descriptions which can be used as mathematical models for formal verification, such as Petri nets, Hoare logic, process calculus, and operational semantics.

Sounds great! Why don't we just use this everywhere and the only things we have to worry about are the odd meteor impact?

```

307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
```

- Really overkill for most systems. However, it is used commercially, e.g.:
 - seL4 - Formally verified OS microkernel
 - Various cryptographic protocols
 - Various embedded software

Much more common in hardware systems, though, where I/O is very well specified and errors can be catastrophic

Why is it overkill?

$$\text{pi}(x) > \text{li}(x)$$

It's hard.

There are so many special cases.

When you absolutely, positively need to prove that your code is correct, you can formally verify it.

Formal verification is using mathematical models to prove or disprove the correctness of the algorithms and code of your program.

There are a variety of "levels" of verification -

- 1. Predictable execution***
- 2. Partial correctness***
- 3. Full correctness***

Predictable Execution

The code is free of what we would call "runtime errors"; that is, no array overflows, null pointer dereferencing, uninitialized variable access, division by zero, etc.

NB not that these are very unlikely, but
PROVEN TO BE IMPOSSIBLE.

Partial Correctness

Predictable execution, PLUS the program will give the correct answer according to the specification IF it terminates (yay Halting Problem!).

Full Correctness

Predictable execution, plus everything referenced in partial correctness, plus the program terminates.

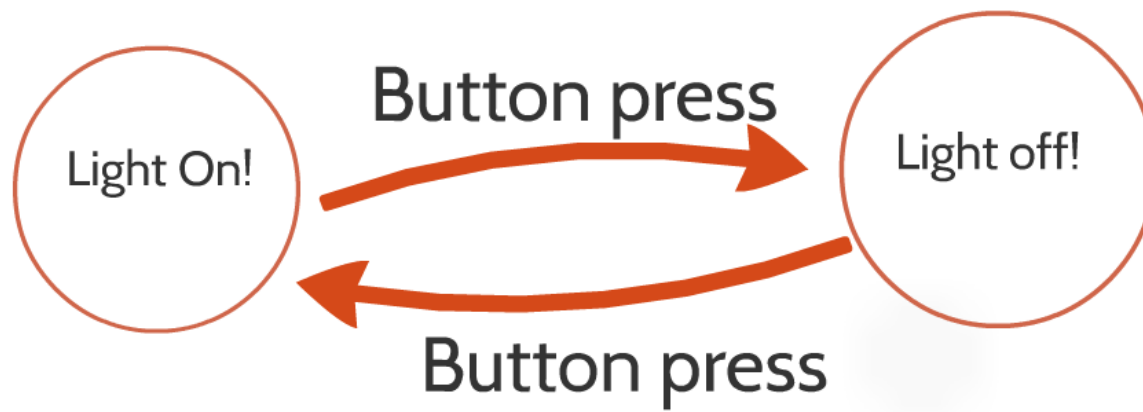
***Difficulty to prove gets harder and harder
as you go up the levels of verification.***

How is it done?

1. Before-the-fact: construct code from formally-proven subset of logic
2. After-the-fact: attempt to verify code using static analysis (much easier with some languages than others...)

What tools can you use?

You're probably familiar with one already...





Light On!



Light off!

There are tools to convert a Finite State Machine to code, or show code as a Finite State Machine.

There are other tools/descriptions which can be used as mathematical models for formal verification, such as Petri nets, Hoare logic, process calculus, and operational semantics.

Sounds great! Why don't we just use this everywhere and the only things we have to worry about are the odd meteor impact?

*54.42. $\vdash :: \alpha \in 2, \supset : \beta \subset \alpha, \mathfrak{H}! \beta, \beta \neq \alpha, \equiv, \beta \in t''\alpha$

Dem.

$\vdash, *54.4, \supset \vdash :: \alpha = t'x \cup t'y, \supset :$

$\beta \subset \alpha, \mathfrak{H}! \beta, \equiv : \beta = \Lambda, \vee, \beta = t'x, \vee, \beta = t'y, \vee, \beta = \alpha : \mathfrak{H}! \beta :$

[*24.53-56, *51.161] $\equiv : \beta = t'x, \vee, \beta = t'y, \vee, \beta = \alpha$ (1)

$\vdash, *54.25, \text{Transp.}, *52.22, \supset \vdash : x \neq y, \supset, t'x \cup t'y \neq t'x, t'x \cup t'y \neq t'y :$

[*13.12] $\supset \vdash : \alpha = t'x \cup t'y, x \neq y, \supset, \alpha \neq t'x, \alpha \neq t'y$ (2)

$\vdash, (1), (2), \supset \vdash :: \alpha = t'x \cup t'y, x \neq y, \supset :$

$\beta \subset \alpha, \mathfrak{H}! \beta, \beta \neq \alpha, \equiv : \beta = t'x, \vee, \beta = t'y :$

[*31.235] $\equiv : (\exists x), x \in \alpha, \beta = t'x :$

[*37.6] $\equiv : \beta \in t''\alpha$ (3)

$\vdash, (3), *11.11.35, *54.101, \supset \vdash, \text{Prop}$

*54.43. $\vdash : \alpha, \beta \in 1, \supset : \alpha \cap \beta = \Lambda, \equiv, \alpha \cup \beta \in 2$

Dem.

$\vdash, *54.26, \supset \vdash : \alpha = t'x, \beta = t'y, \supset : \alpha \cup \beta \in 2, \equiv, x \neq y,$

[*51.231] $\equiv, t'x \cap t'y = \Lambda,$

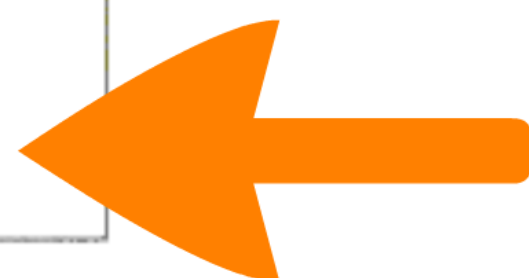
[*13.12] $\equiv, \alpha \cap \beta = \Lambda$ (1)

$\vdash, (1), *11.11.35, \supset$

$\vdash : (\mathfrak{H}x, y), \alpha = t'x, \beta = t'y, \supset : \alpha \cup \beta \in 2, \equiv, \alpha \cap \beta = \Lambda$ (2)

$\vdash, (2), *11.54, *52.1, \supset \vdash, \text{Prop}$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$



Really overkill for most systems. However, it is used commercially, e.g.:

seL4 - Formally verified OS microkernel

Various cryptographic protocols

Various embedded software

Much more common in hardware systems, though, where I/O is very well specified and errors can be catastrophic

Why is it overkill?

$$\pi(x) > \text{li}(x)$$

It's hard.

There are so many special cases.

Property-Based Testing and Formal Verification

