Directed Random Testing

```
bool var35 = Collections.replaceAll(var20, var23, var28);
java.lang.Object var36 = var18.put(1L, 10L);
int var37 = Arrays.binarySearch(var4, var23);
assert var37 == 3;
Arrays.fill(var1, var3, var37, -1);
BitSet var3        BitSet(var37);
Comparator          llections.reverseOrder();
PriorityQ          new Pr       Queue(var37, var39);
TreeSet v          reeS
bool var42         Empt
TreeSet var43 = (TreeSet            );
assert var43.size() == va
char[] var44 = new char[] {
Set var76 = Collections.unmodifiableSet((Set)var41);
assert var76.equals(var76);
```

Carlos Pacheco

MIT CSAIL

2/17/2009

Software testing

expensive

30-90% of development effort

	Cost of inadequate testing on US economy (billions)
developers	$21.2
users	$38.3
TOTAL	**$59.5**

source: NIST 2002

difficult

complex software

› many behaviors to test

large input spaces

› selecting subset is hard

done mostly by hand

› at Microsoft, ½ of engineers

goal: automation

automate test case creation

› a principal testing activity

› a significant portion of cost

Approach

random testing

simple, effective

› reveals errors

unix utilities	[Miller 1990]
OS services	[Kropp 1998]
GUIs	[Forrester 2000]
functional code	[Claessen 2000]
OO code	[Oriat 2004]
	[Csallner 2005]
flash file systems	[Groce 2007]

suffers from deficiencies

› many useless inputs

› low coverage

directed random testing

harness random testing

but make it better

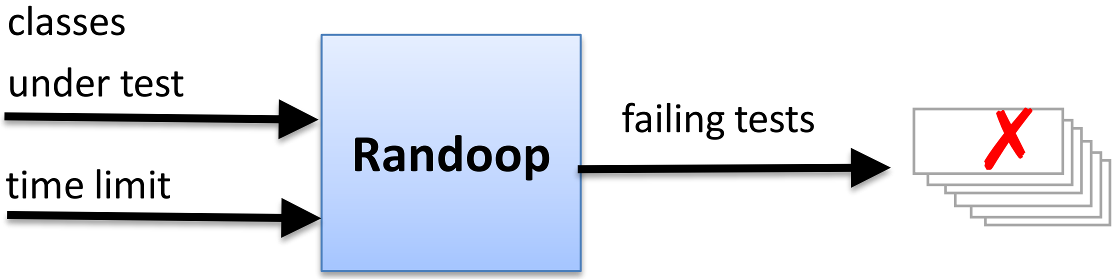
› reveal *more* errors

› produce better test cases

› achieve higher coverage

Randoop: directed random testing for Java

automatically creates unit tests



classes under test

time limit

Randoop

failing tests

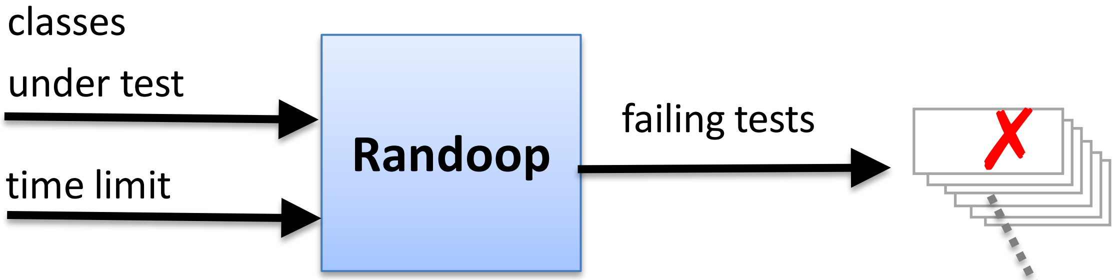
tests built-in properties

user can extend

- Reflexivity of `equals`
  `i.e. o.equals(o)==true`
- Symmetry of `equals`
- Consistency of `equals/hashCode`
- etc.

Randoop: directed random testing for Java

automatically creates unit tests



```
java.util.Collections
java.util.ArrayList
java.util.TreeSet
java.util.LinkedList
...
```

tests built-in properties

user can extend

- Reflexivity of `equals`
  i.e. `o.equals(o)==true`
- Symmetry of `equals`
- Consistency of `equals/hashCode`
- etc.

```
public void test() {

  TreeSet s = new TreeSet();
  s.add("xyz");
  Set u = unmodifiableSet(s);

  // Fails at runtime.
  assertTrue(u.equals(u));
}
```

Randoop demo

Randoop is effective

Reveals unknown errors [ICSE 2007, ISSTA 2008]

- Across many large, widely-used reusable component libraries

distinct errors revealed (Java)

code base	**Randoop**	**JPF** (model checker)	**JCrasher** (random tester)
Sun JDK (272 classes,43KLOC)	***8***	0	1
Apache libraries (974 classes, 114KLOC)	***6***	1	0

distinct errors revealed (.NET)

code base	**Randoop**	symbolic execution unit test generator
.NET library (1439 classes, 185KLOC)	***30***	**0**

Randoop is *cost* effective...

...in a real industrial testing environment, when used by practicing test engineers.

Case study [ISSTA 2008]

› Microsoft test team

› Randoop (.NET version)

› Applied to highly-tested library

  - tested over 5 years by 40 engineers

revealed more errors in **15 hours** than team typically discovers in **1 person-year** of effort

Randoop in research

component in new techniques

dynamic mutability analysis	[Artzi, ASE 07]
concurrency testing	[Yu, Microsoft (unpub.), 07]
regression analysis	[Orso, WODA 08]
change-based test generation	[d'Amorim, under submission]
coverage-driven test generation	[Jaygari, under submission]
test selection	[Jaygari, under submission]

evaluation benchmark

genetic algos. for test gen.	[Andrews, ASE 07]
manual/automatic testing study	[Baccheli, BCR 08]
symbolic execution	[Ikumnsah, ASE 08]
predicting test tool effectiveness	[Daniel, ASE 08]
equality from abstraction	[Rayside, ICSE 09]

Randoop outside research

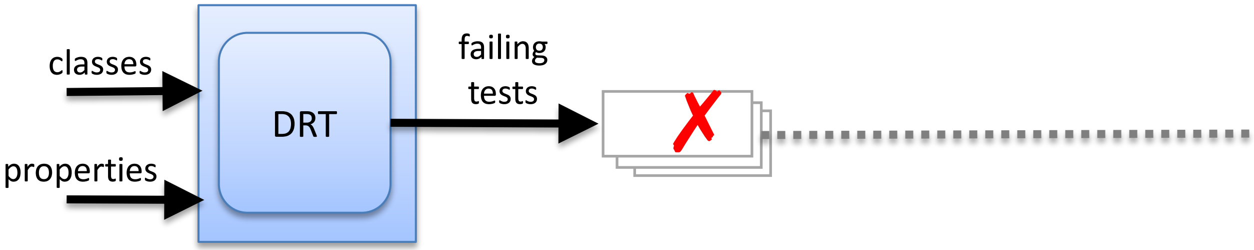
industrial bug finder

Microsoft	Used to find bugs in .NET software
Fraunhofer USA, Inc Center for Experimental Software Engineering Maryland	Applying Randoop to NASA projects

learning vehicle

ILLINOIS	Advanced Topics in Software Engineering
CALTECH	Reliable Software: Testing and Monitoring
UFPE	Software Testing

Contributions (1) *directed random test generation (DRT)*



useful unit test

```
TreeSet s = new TreeSet();
s.add("xyz");
Set u = synchronizedSet(s);
assertTrue(u.equals(u));
```

what it does

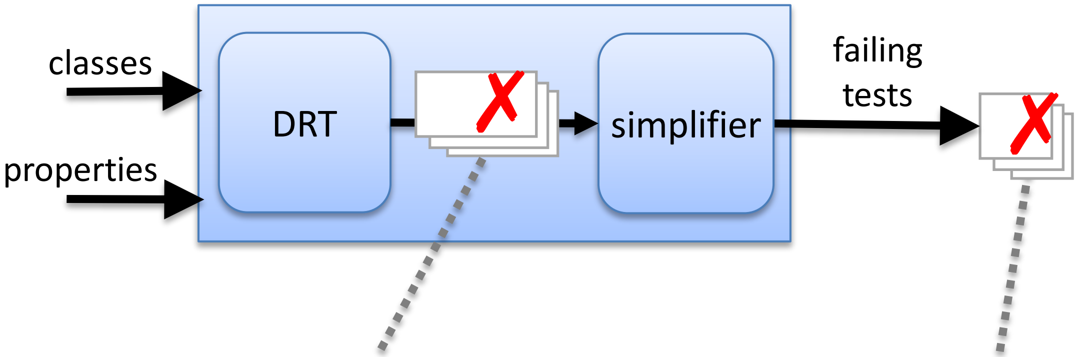
- generates useful inputs
  - reveal errors
  - achieve good code coverage
- avoids useless inputs
  - illegal
  - redundant

how it does it (more details soon)

- uses runtime information
  - about code under test
- prunes input space

Contributions (2)

replacement-based simplification



original input

```
long[] var1 = new long[] { 1L, 10L, 1L };
Vector var2 = new Vector();
PriorityQueue var3 = new PriorityQueue();
Object var4 = var3.peek();
bool var5 = var2.retainAll(var3);
long[] var6 = new long[] ;
Arrays.fill(var6, 10L);
IdentityHashMap var7 = new IdentityHashMap(100);
Object var8 = var7.remove(var7);
Set var9 = var7.keySet();
Vector var10 = new Vector(var9);
Vector var11 = new Vector();
IdentityHashMap var12 = new IdentityHashMap(100);
String var13 = var12.toString();
Vector var14 = new Vector();
IdentityHashMap var15 = new IdentityHashMap(100);
char[] var16 = new char[] { ’ ’, ’#’, ’ ’ };
int var17 = Arrays.binarySearch(var16, ’a’);
Object var18 = var15.remove(var16);
bool var19 = var14.equals(var15);
bool var20 = var10.addAll(0, var14);
Collections.replaceAll(var11, var17, var14);
int var21 = Arrays.binarySearch(var1, var17);
Comparator var22 = Collections.reverseOrder();
TreeSet var23 = new TreeSet(var22);
bool var24 = var23.isEmpty();
Object var25 = var23.clone();
Object[] var26 = new Object [] var21 ;
List var27 = Arrays.asList(var26);
ArrayList var28 = new ArrayList(var27);
bool var29 = var23.add(var28);
Set var30 = Collections.synchronizedSet(var23);
assert var30.equals(var30); //fails at runtime
```

simplified input

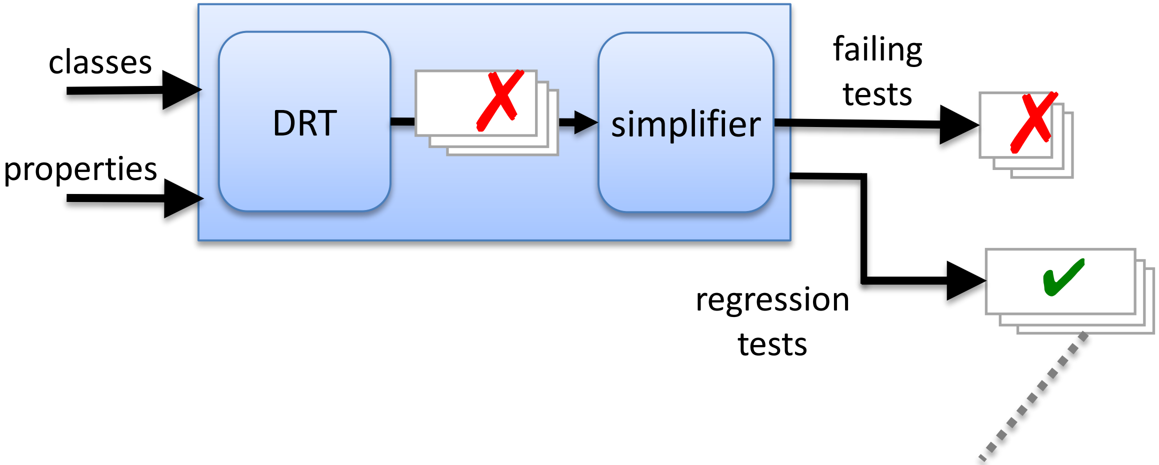
```
TreeSet s = new TreeSet();
s.add("xyz");
Set u = synchronizedSet(s);
assertTrue(u.equals(u));
```

delta debugging

```
Comparator var22 = reverseOrder();
TreeSet var23 = new TreeSet(var22);
long[] var1 = new long[] { 1L,10L,1L };
char[] var16 = new char[] {’ ’,’#’,’ ’};
int var17 = binarySearch(var16, ’a’);
int var21 = binarySearch(var1, var17);
Object[] var26 = new Object [] var21 ;
List var27 = asList(var26);
ArrayList var28 = new ArrayList(var27);
bool var29 = var23.add(var28);
Set var30 = synchronizedSet(var23);
assert var30.equals(var30);
```

Contributions (3)

regression generation



regression test case

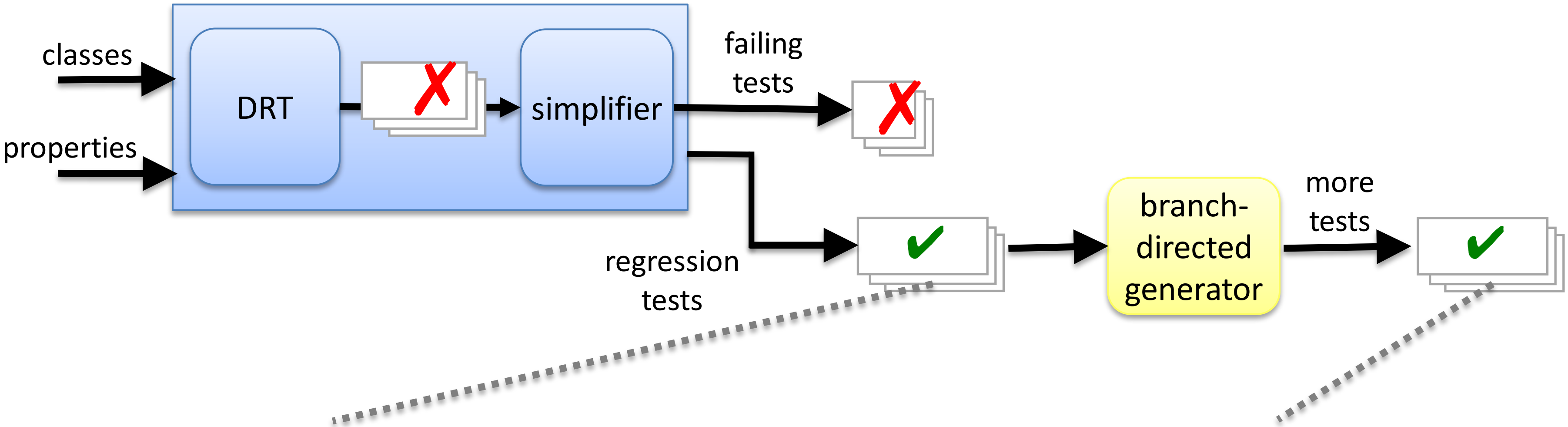
```
Object o = new Object();
LinkedList l = new LinkedList();
l.sumFirst(o);
l.sum(o);
assertEquals(2, l.size());
assertEquals(false, l.isEmpty());
```

observer methods capture current behavior

Randoop's regression tests reveal serious inconsistencies among Sun JDK 1.5, Sun JDK 1.6, and IBM JDK 1.5

Contributions (4)

branch-directed generation



original input

```
Stack s = new Stack();
s.setSize(1);
s.peek();
```

1. analyze data flow
2. modify arguments

new input

```
Stack s = new Stack();
s.setSize(0);
s.peek();
```

uncovered statement

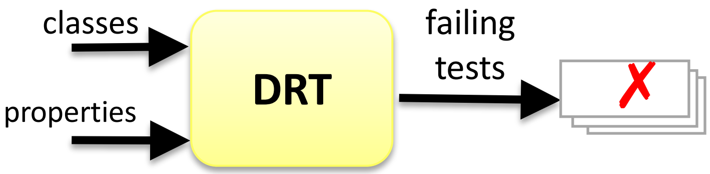
```
Object peek() {
 len = size();
 if (len == 0)
   throw new EmptyStackException();
 return elementAt(len - 1);
}
```

```
Object peek() {
 len = size();
 if (len == 0)
   throw new EmptyStackException();
 return elementAt(len - 1);
}
```

Rest of talk

DRT: directed random test generation

- uses runtime information
- prunes input space



experiments

- coverage
- error-revealing effectiveness
- benefits of pruning

industrial case study

DRT roadmap

➡ **traditional random testing**

DRT

1. incremental generation

2. adding guidance

3. automating guidance

Example: a polynomial library

From MIT 6.170,
Laboratory in Software Engineering

```
class Mono {

    int num, den, exp;

     Mono(int num, int den, int
     exp)
}
```

$$\frac{num}{den}\ x^{exp}$$

```
class Poly {
    List<Mono> elements;

    Poly()

    Poly sum(Mono m)

    Poly deriv()

    Poly integral(int coeff)

    ...
}
```

Constructs the "0" polynomial.

representation invariant

elements sorted
in order of decreasing exponent

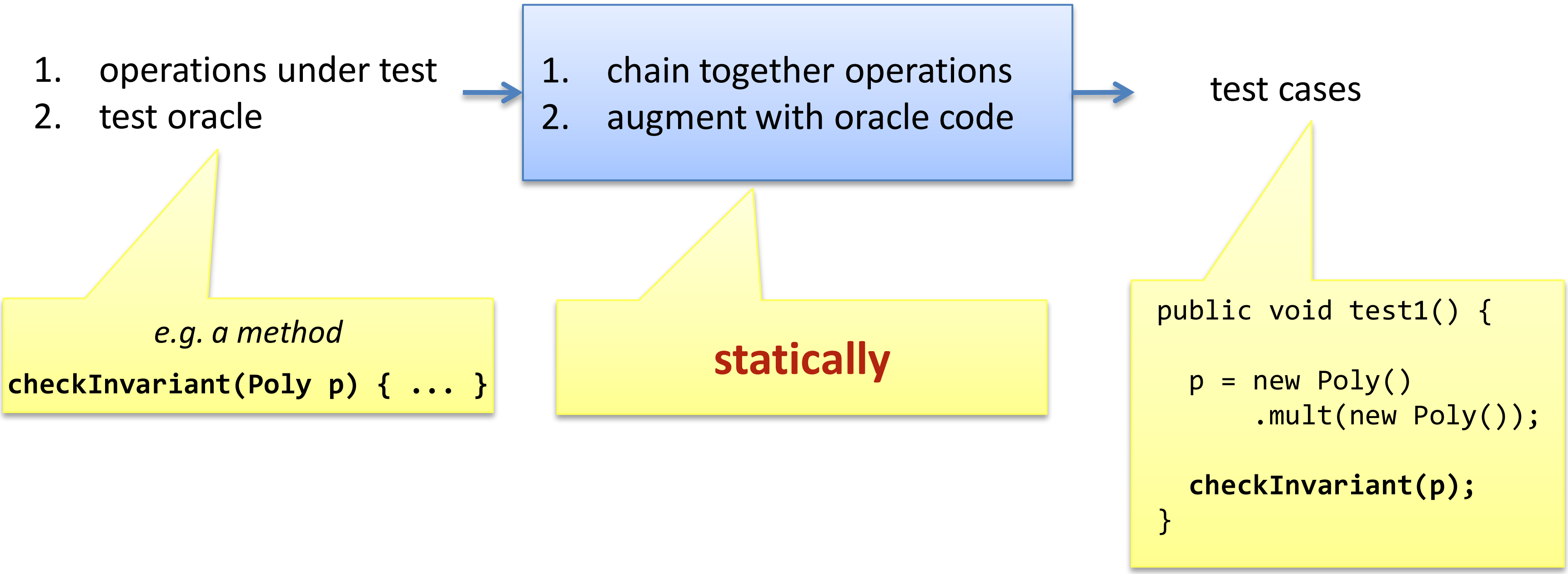
Previous work on random test generation

Unit test generators

- Jartege [Oriat 03]
- JCrasher [Csallner 04]

Create unit tests randomly, statically

- each test independent of previous ones (no feedback)
- user compiles, run tests to see if they reveal errors



1. operations under test
2. test oracle

1. chain together operations
2. augment with oracle code

test cases

e.g. a method

`checkInvariant(Poly p) { ... }`

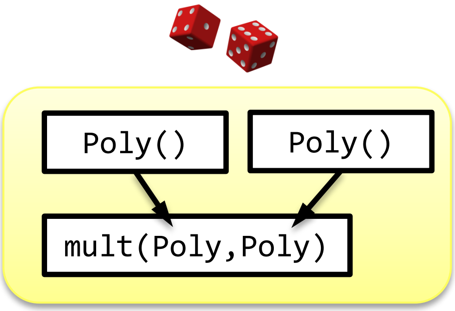
statically

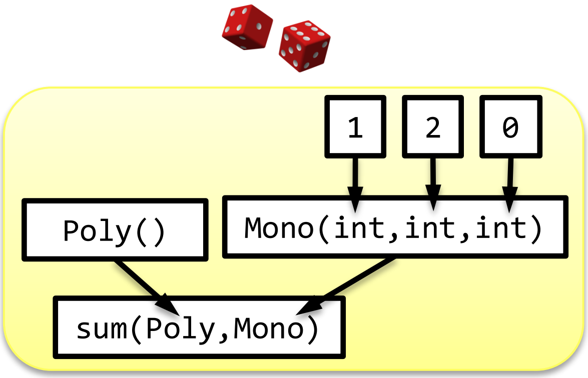
```
public void test1() {

   p = new Poly()
         .mult(new Poly());

   checkInvariant(p);
}
```

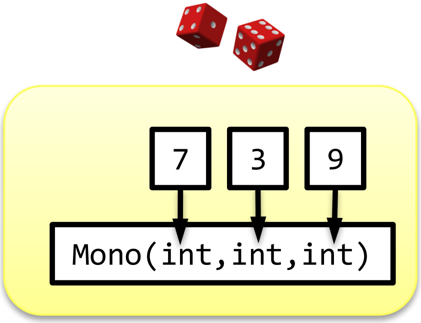
Previous work on random test generation

operation	input	output
Mono(int,int,int)	3 ints	a new Mono
Poly()	none	a new Poly
Poly plus(Mono)	a Poly, a Mono	a new Poly

random terms





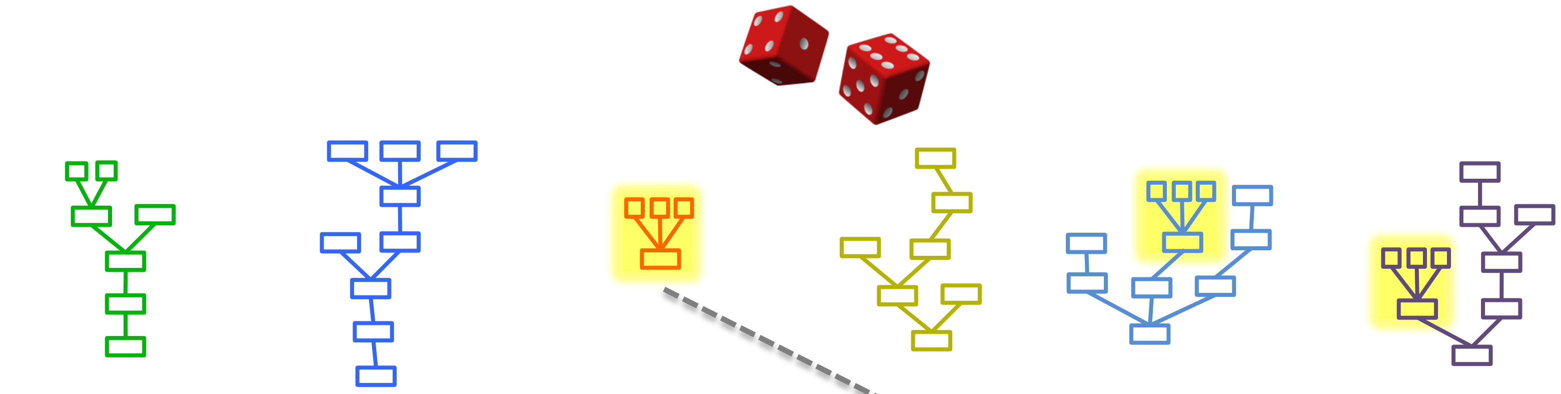


```
public void test1() {

  p = new Poly()
      .mult(new Poly());

  checkInvariant(p);
}
```

```
public void test2() {

  p = new Poly()
      .sum(new Mono(1,2,0));

  checkInvariant(p);
}
```

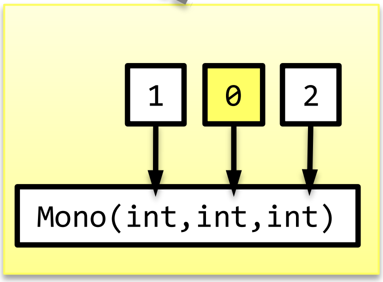
```
public void test3() {

  m = new Mono(7,3,9);


  checkInvariant(m);
}
```

Problems with previous work



generates useless inputs

- illegal, repetitive



illegal input to Mono

throws IllegalArgException

`Mono(int num, int den, int exp)`

Expects den != 0 and exp >= 0

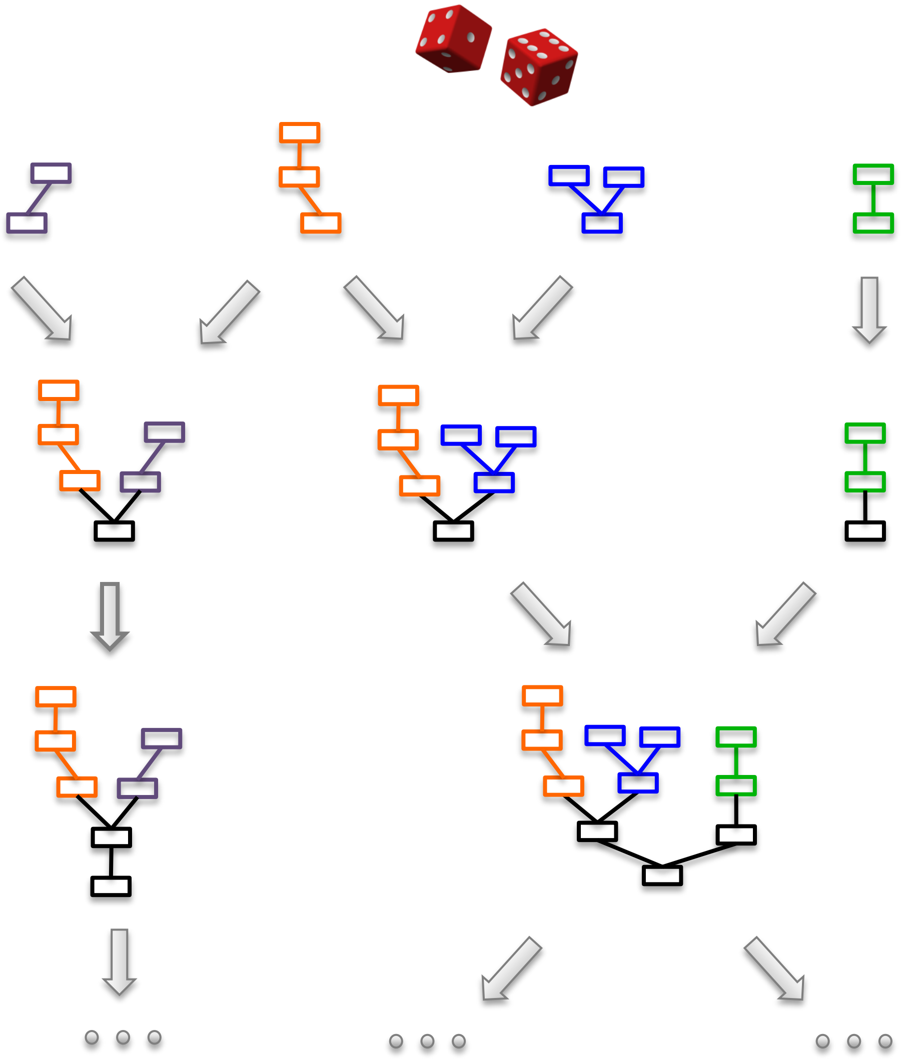
Directed random test generation

[Pacheco ICSE 07, OOPSLA 08, ISSTA 08]

uses randomness in generation

builds inputs incrementally

- new inputs combine old



Directed random test generation

[Pacheco ICSE 07, OOPSLA 08, ISSTA 08]

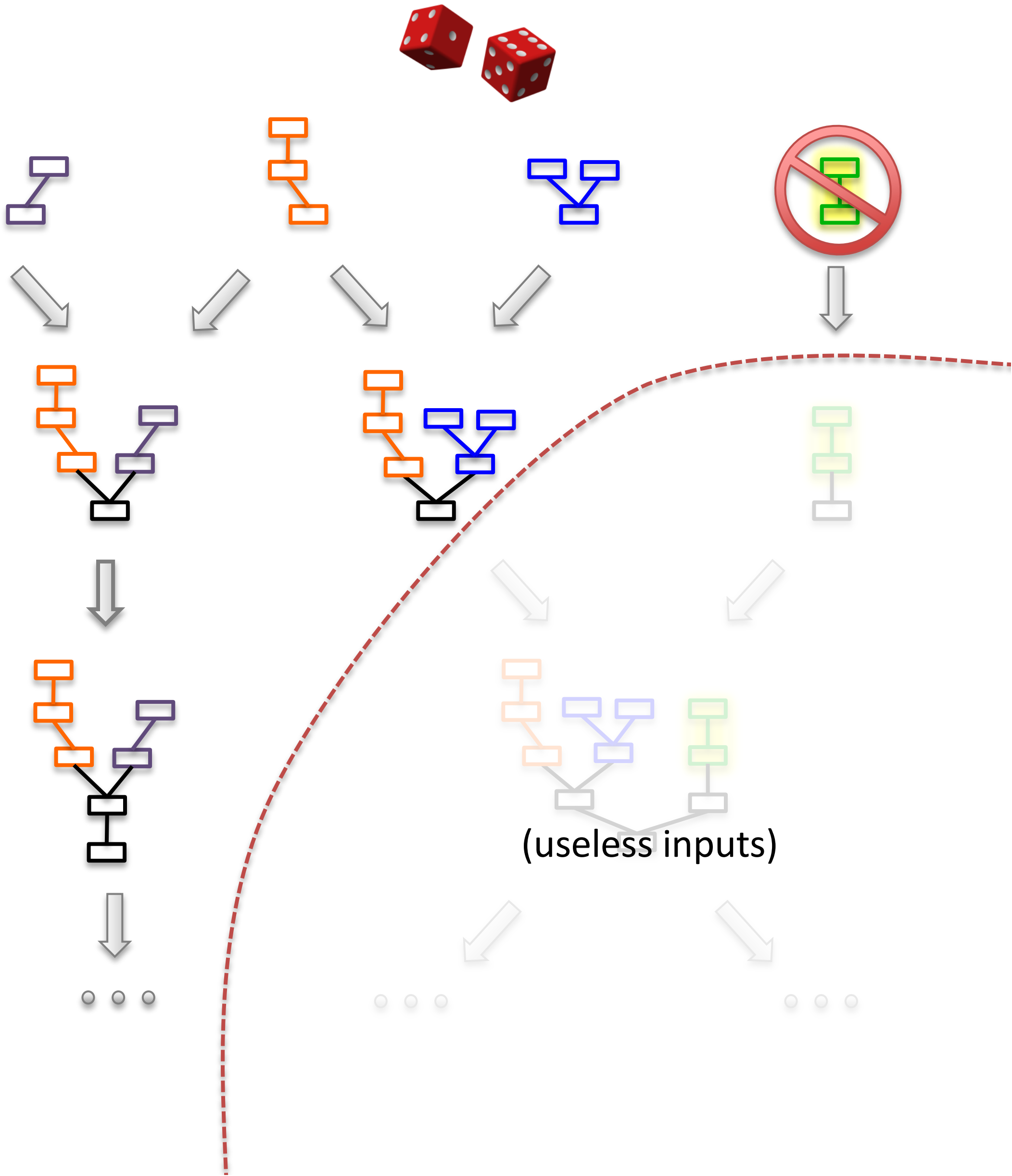
uses randomness in generation

builds inputs incrementally

- new inputs combine old

executes inputs

- discards ones useless **for extension**
  - illegal
  - redundant
  - error-revealing
- prunes input space



DRT roadmap

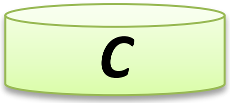
traditional random testing

directed random test generation

1. incremental generation
2. adding guidance
3. automating guidance

An incremental generator

select op
$m(T_1 .. T_k)$



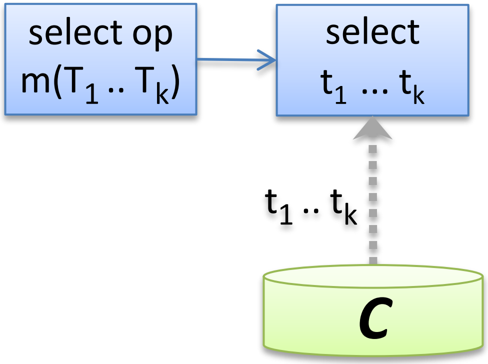
component set of terms

C = { `0, 1, 2, null, false, etc.` }

Example:

```
Mono(int,int,int)
```

An incremental generator



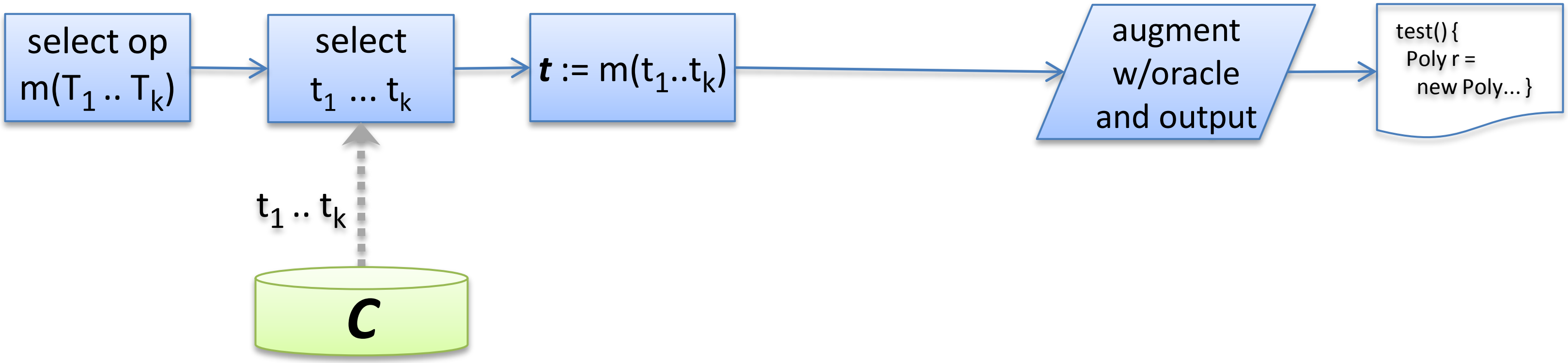
component set of terms

C = { `0, 1, 2, null, false, etc.` }

`1` `2` `0`

Example: `Mono(int,int,int)`

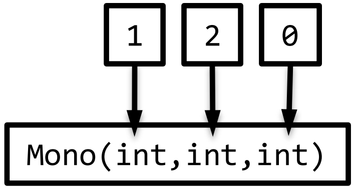
An incremental generator



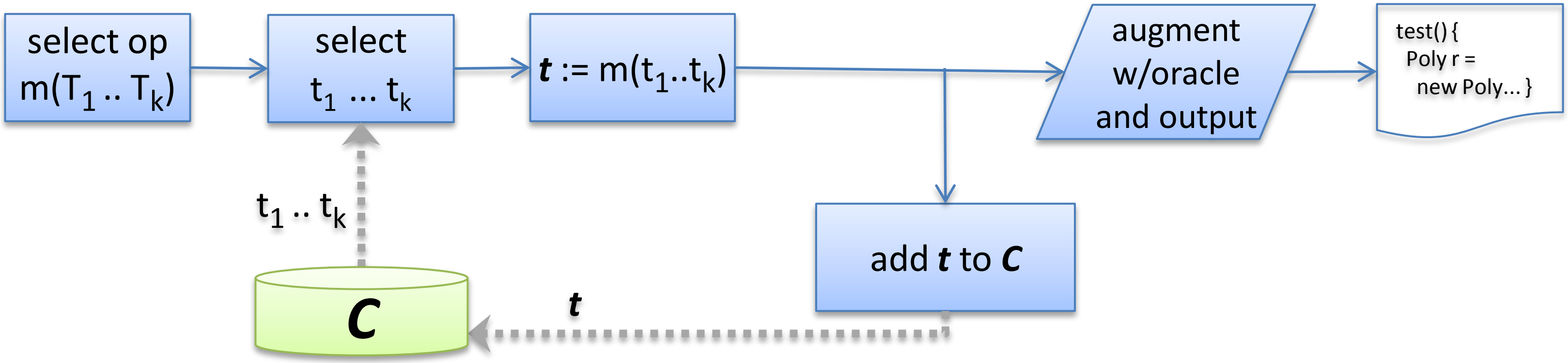
component set of terms

C = { `0, 1, 2, null, false, etc.` }

Example:



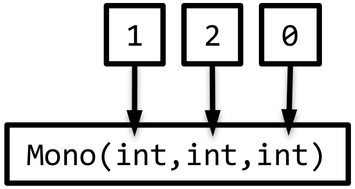
An incremental generator



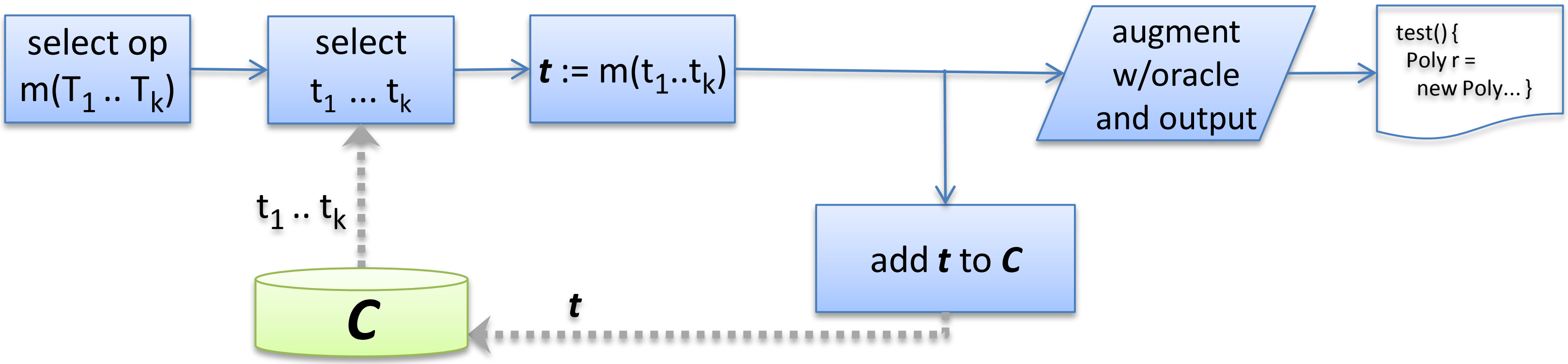
component set of terms

C = { `0, 1, 2, null, false, Mono(1,2,0)` }

Example:



An incremental generator



select op $m(T_1 .. T_k)$

select $t_1 \ldots t_k$

$\boldsymbol{t} := m(t_1..t_k)$

augment w/oracle and output

```
test() {
  Poly r =
    new Poly... }
```

$t_1 .. t_k$

add ***t*** to ***C***

C

t

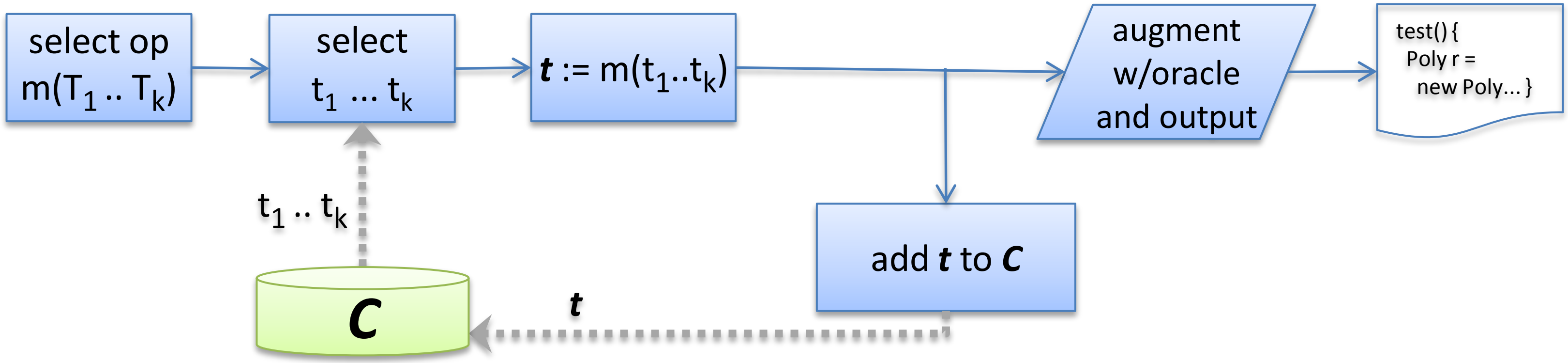
component set of terms

C = { `0, 1, 2, null, false, Mono(1,2,0)` }

Example:

`Poly()`

An incremental generator



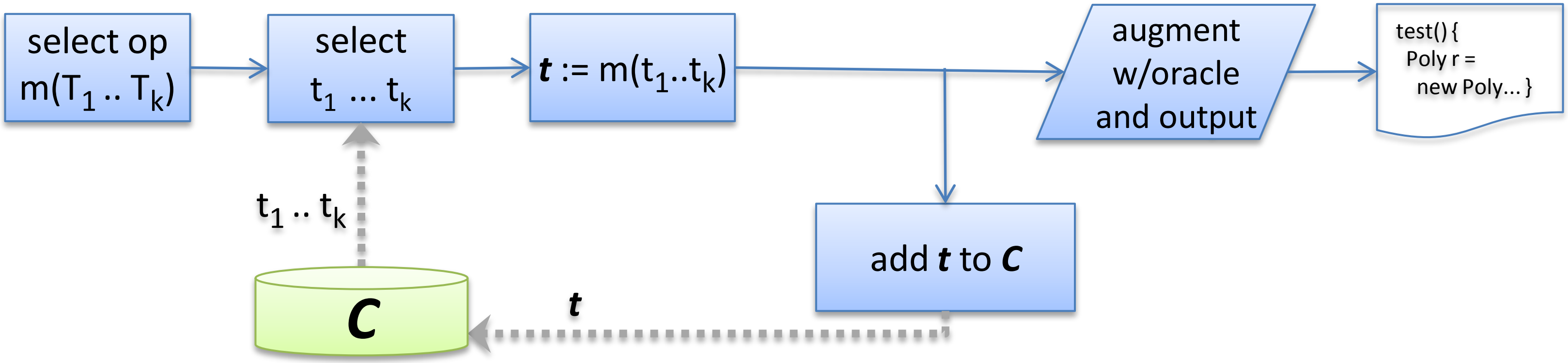
component set of terms

C = { `0, 1, 2, null, false, Mono(1,2,0), Poly()` }

Example:

`Poly()`

An incremental generator



select op $m(T_1 .. T_k)$

select $t_1 \ldots t_k$

t := $m(t_1..t_k)$

augment w/oracle and output

```
test() {
  Poly r =
    new Poly... }
```

add ***t*** to ***C***

$t_1 .. t_k$

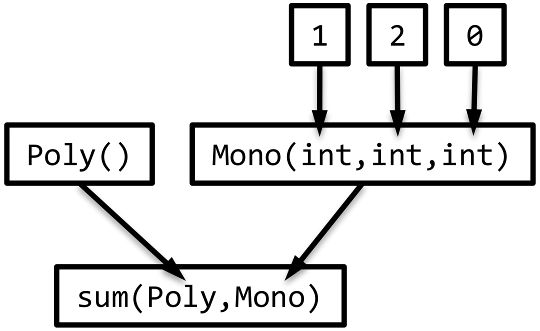
C

t

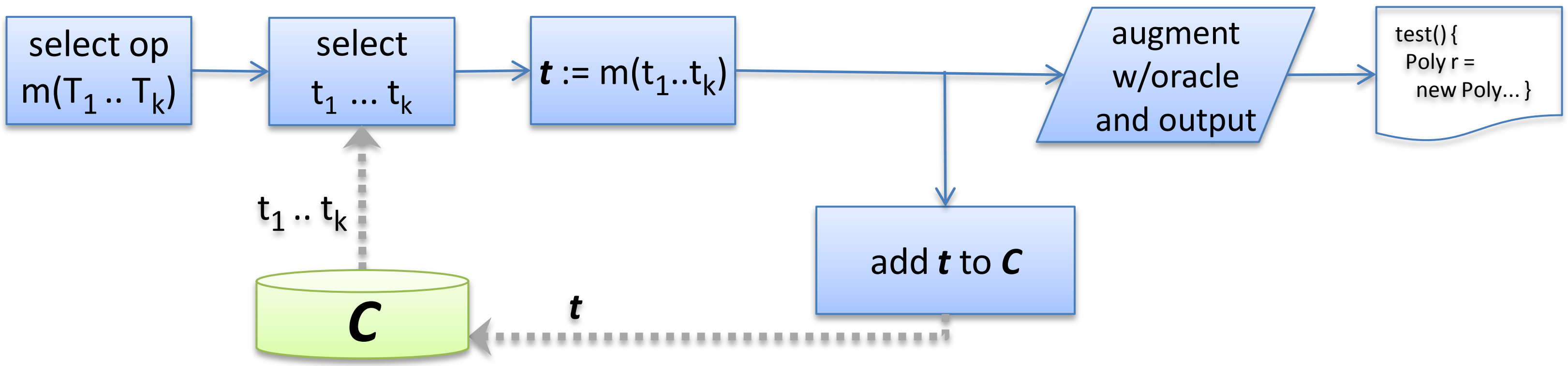
component set of terms

C = { `0, 1, 2, null, false, Mono(1,2,0), Poly()` }

Example:



An incremental generator

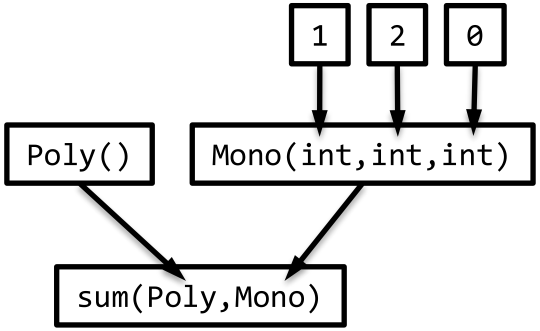

select op m(T_1 .. T_k)
select t_1 … t_k
t := m(t_1..t_k)
augment w/oracle and output
add ***t*** to ***C***
t_1 .. t_k
C
t

```
test() {
  Poly r =
    new Poly...}
```

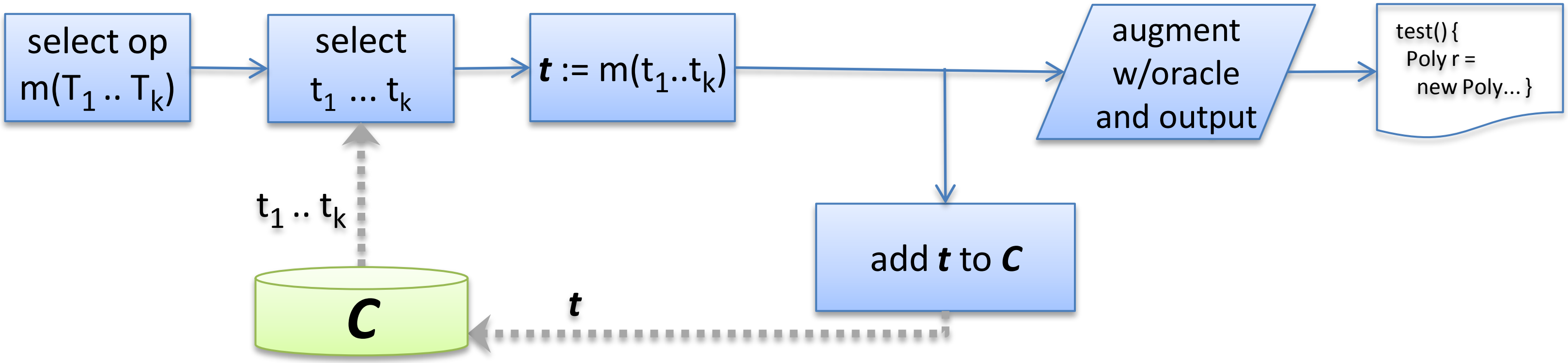
component set of terms

C = { `0, 1, 2, null, false, Mono(1,2,0), Poly(), sum(Poly(),Mono(1,2,0))` }

Example:



An incremental generator


select op m(T_1 .. T_k)
select t_1 … t_k
t := m(t_1..t_k)
augment w/oracle and output
add ***t*** to ***C***
t_1 .. t_k
C
t

```
test() {
  Poly r =
    new Poly... }
```

component set of terms

C = { 0, 1, 2, null, false, Mono(1,2,0), Poly(), **sum(Poly(),Mono(1,2,0))** }

next idea

restrict component set → guide generation

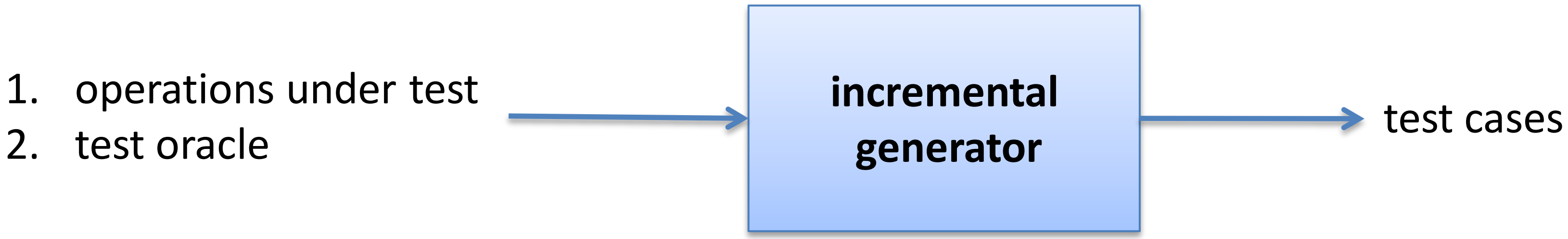
DRT roadmap

traditional random testing

directed random test generation

1. incremental generation
2. adding guidance
3. automating guidance

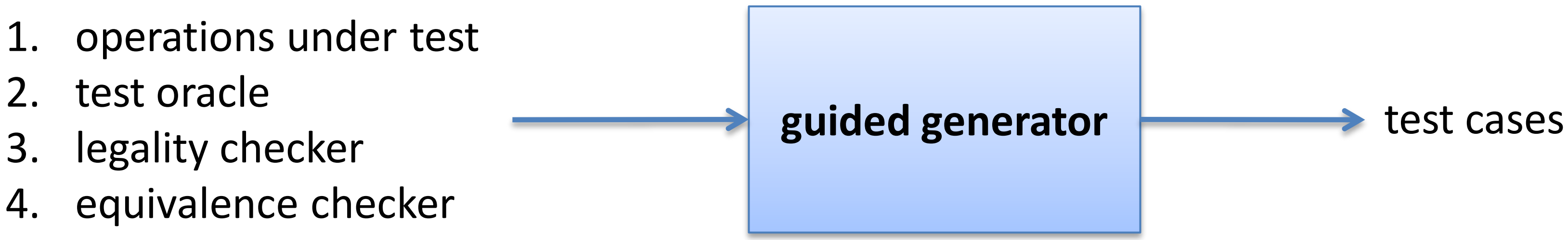
Adding guidance



1. operations under test
2. test oracle

incremental generator

test cases

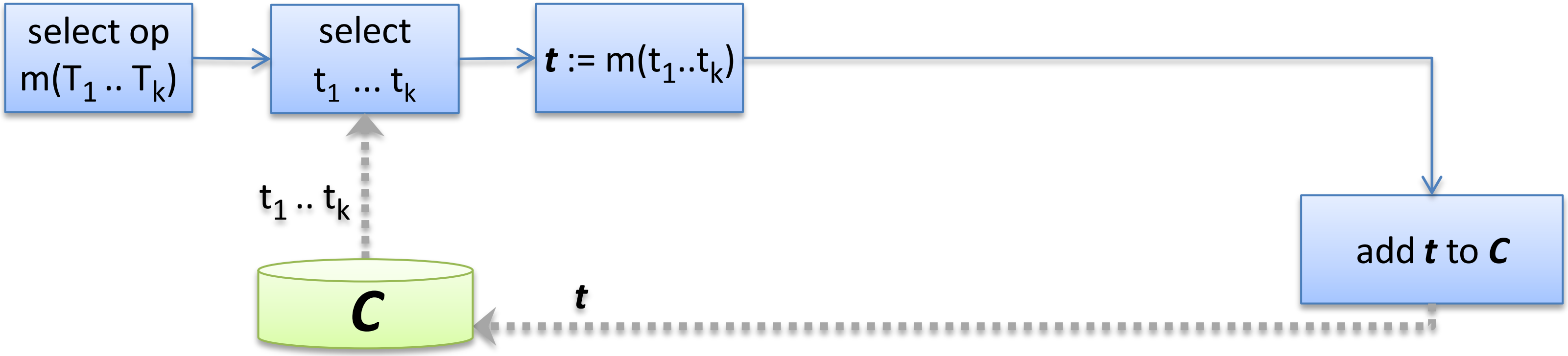


1. operations under test
2. test oracle
3. legality checker
4. equivalence checker

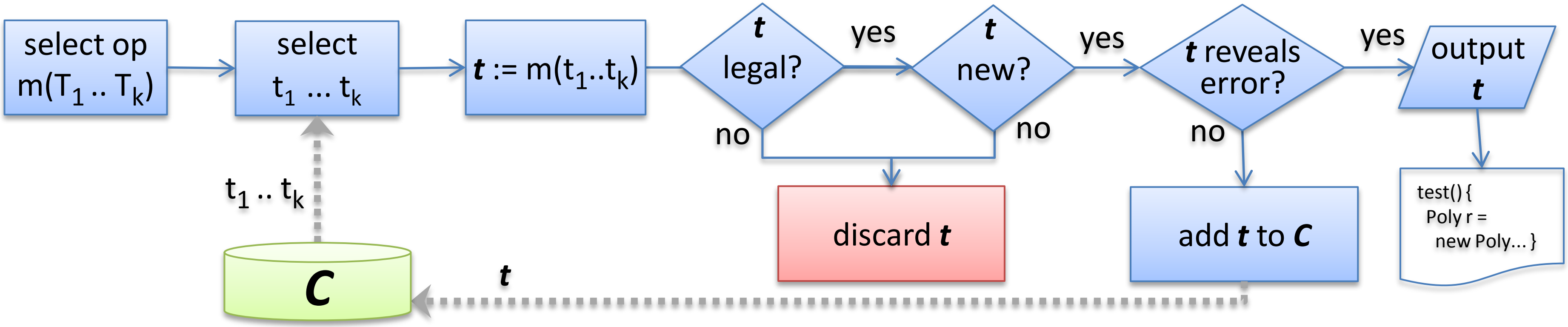
guided generator

test cases

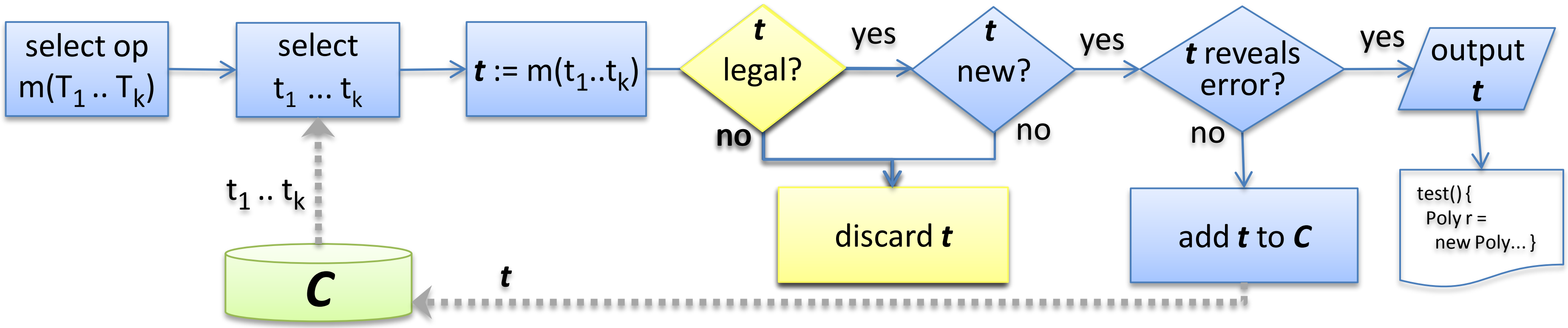
Guided generator



Guided generator

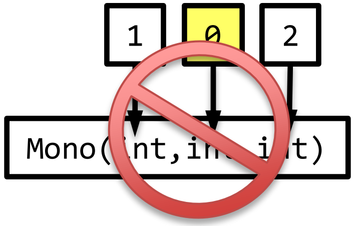


Legality



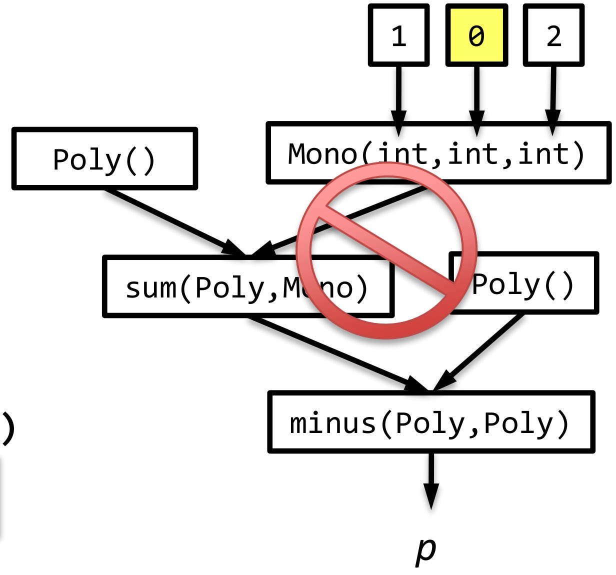
legality checker

- determines if a term is legal/illegal
- discard illegal terms

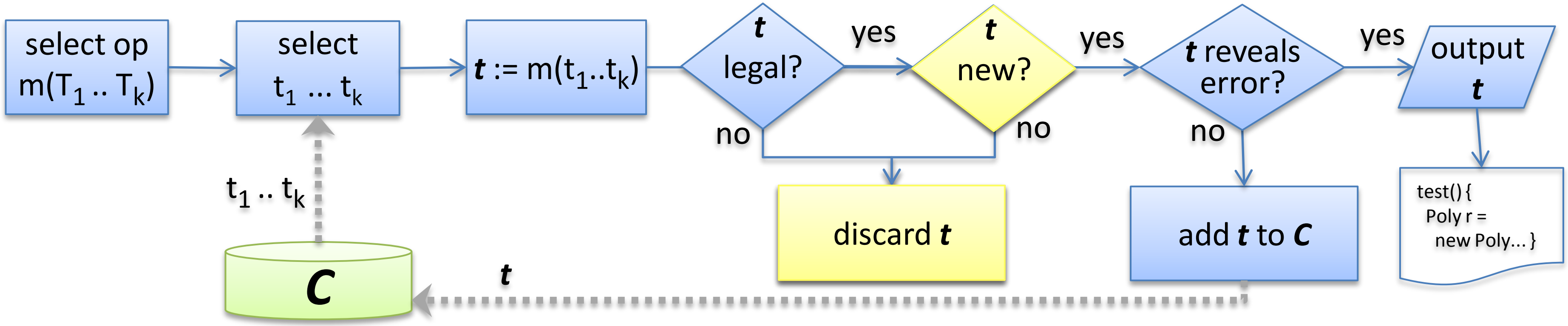


`Mono(int num, int den, int exp)`

Expects `den != 0` and `exp >= 0`.



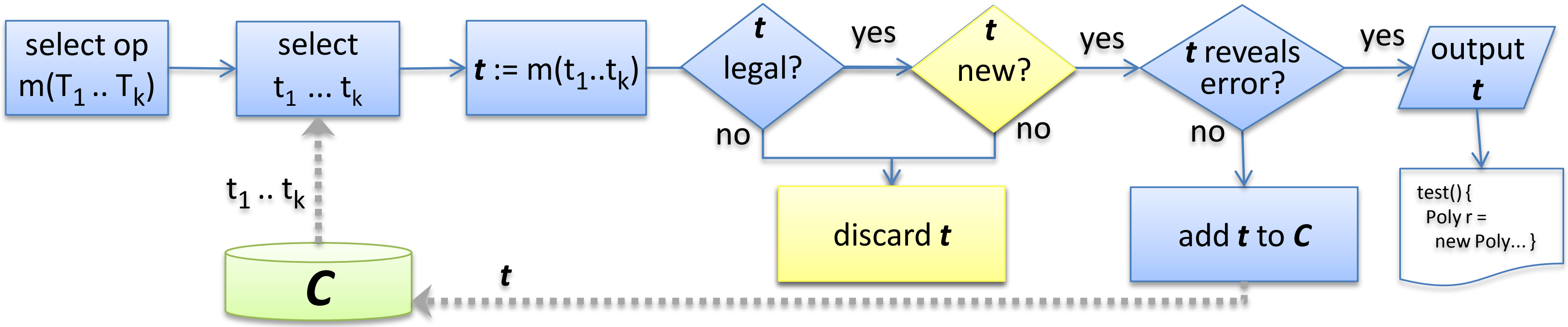
Equivalence



equivalence checker

- determines if two terms are equivalent
- discard term if equivalent to one in *C*

Equivalence



```
test() {
  Poly r =
    new Poly... }
```

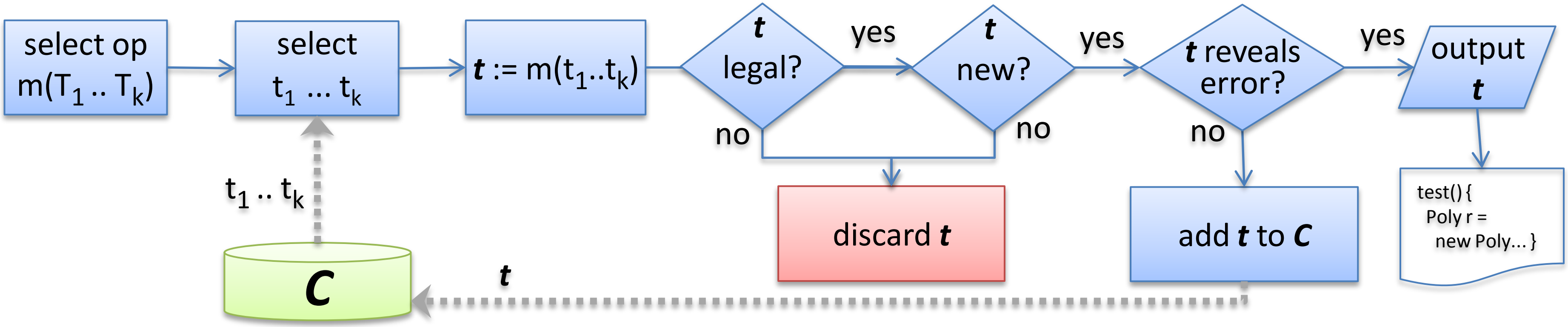
A simple equivalence checker:

t ≈ t'   iff   t = t'

C = { `0, 1, 2, Poly(), Poly()` }

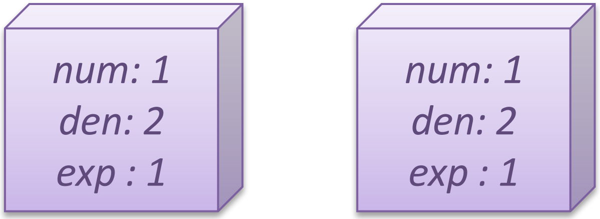
`Poly()`

Equivalence

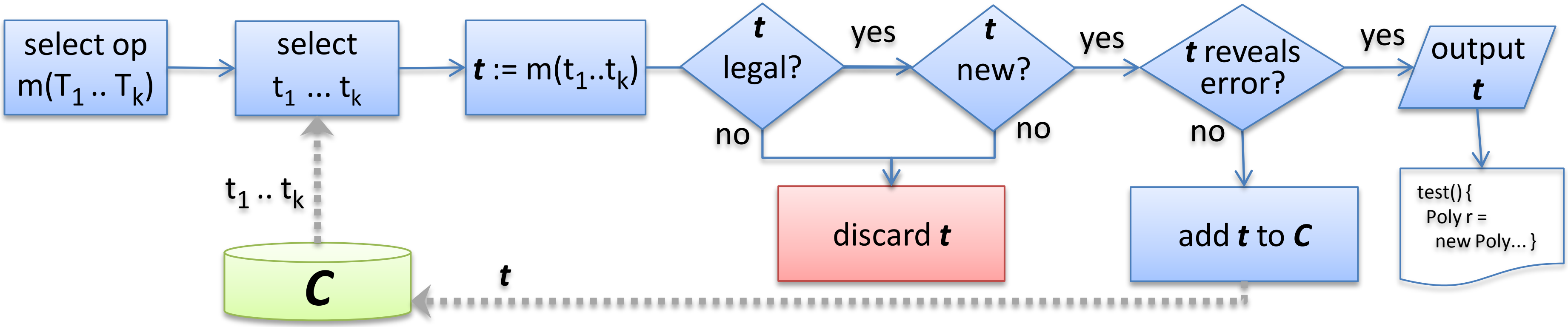


`Mono(1,2,1)` ≈ `Mono(2,4,1)`

at runtime:



Equivalence



select op m(T_1 .. T_k)

select t_1 … t_k

t := m(t_1..t_k)

t legal?

yes

no

t new?

yes

no

t reveals error?

yes

no

output ***t***

discard ***t***

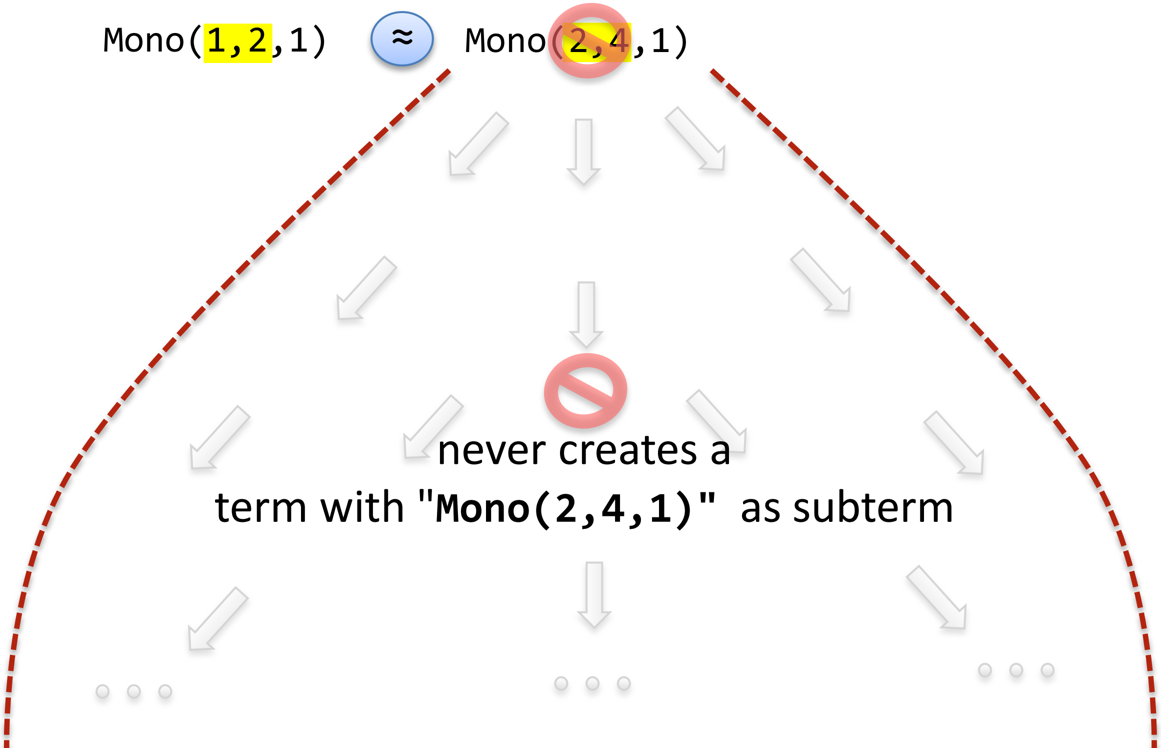
add ***t*** to ***C***

C

t_1 .. t_k

t

```
test() {
  Poly r =
    new Poly... }
```



Mono(1,2,1) ≈ Mono(2,4,1)

never creates a term with "**Mono(2,4,1)**" as subterm

DRT roadmap

traditional random testing

DRT

1. incremental generation

2. adding guidance
    a. discarding illegal inputs
    b. discarding equivalent inputs

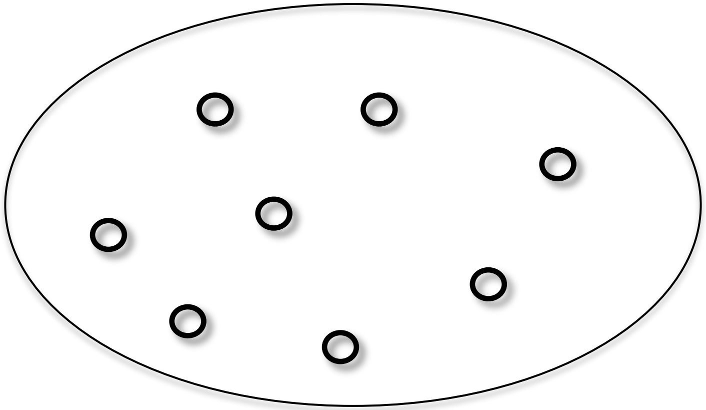
        desired qualities of equivalence

3. automating guidance

Desired qualities of equivalence

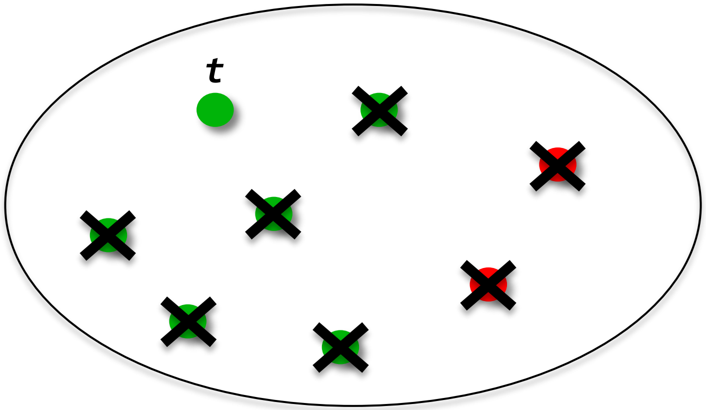
- large equivalence classes

everything is equivalent



Desired qualities of equivalence

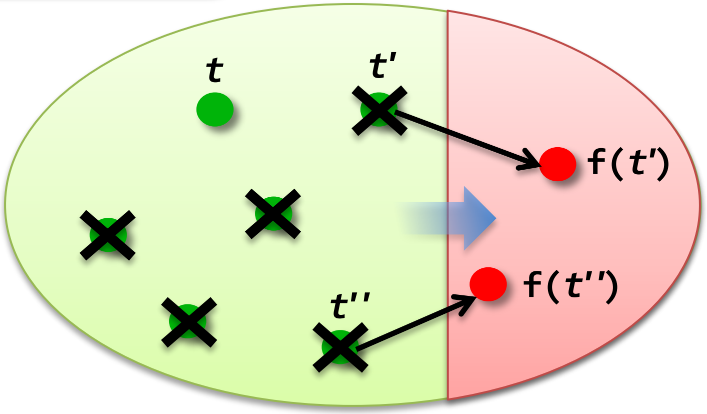
- large equivalence classes
- distinguishes normal/error terms



Desired qualities of equivalence

- large equivalence classes
- distinguishes normal/error terms
- error partition stays reachable

safe equivalence



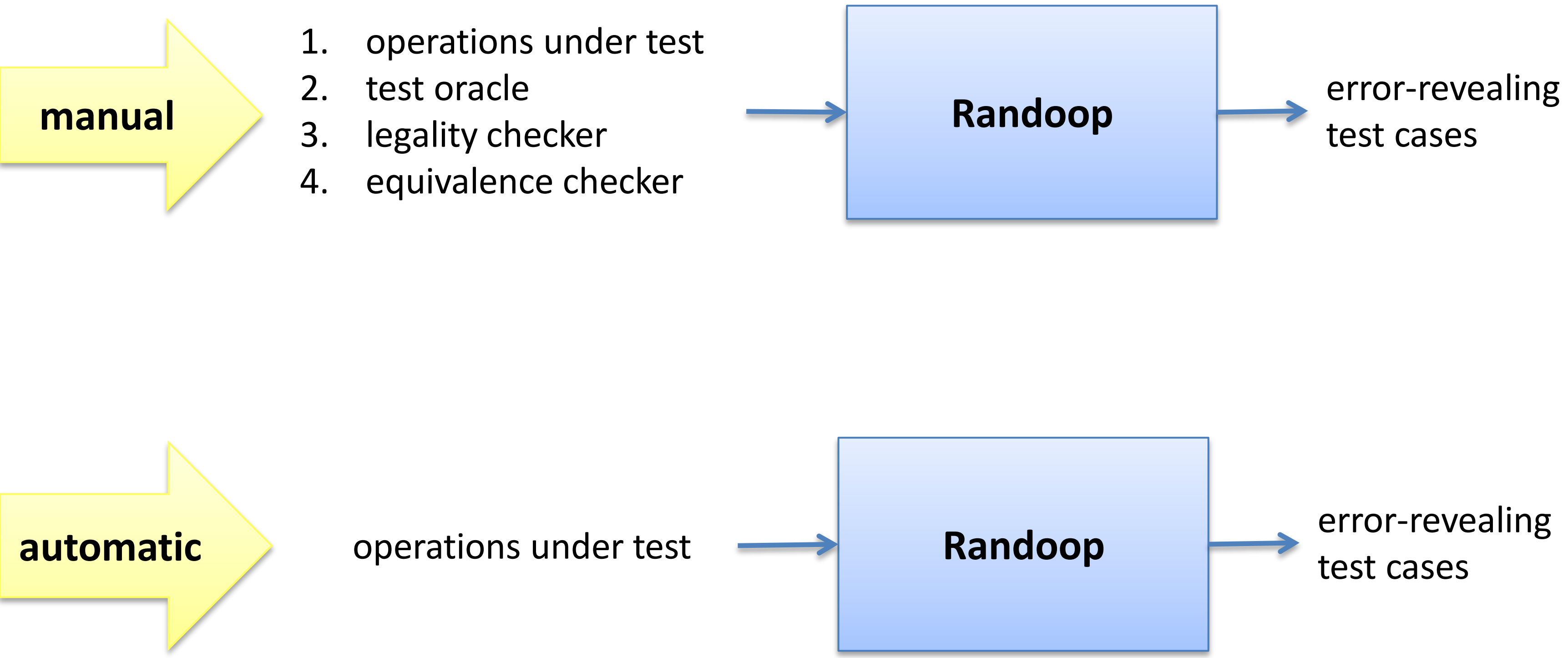
DRT roadmap

traditional random testing

DRT

1. incremental generation
2. adding guidance
3. automating guidance

Randoop: two usage modes



manual

1. operations under test
2. test oracle
3. legality checker
4. equivalence checker

Randoop

error-revealing test cases

automatic

operations under test

Randoop

error-revealing test cases

Randoop's oracles and heuristic guidance

oracles

- based on published API
- check basic properties of Java/.NET classes

legality, equivalence checkers

- legality: based on **exceptions**
- equivalence: based on **equals method**

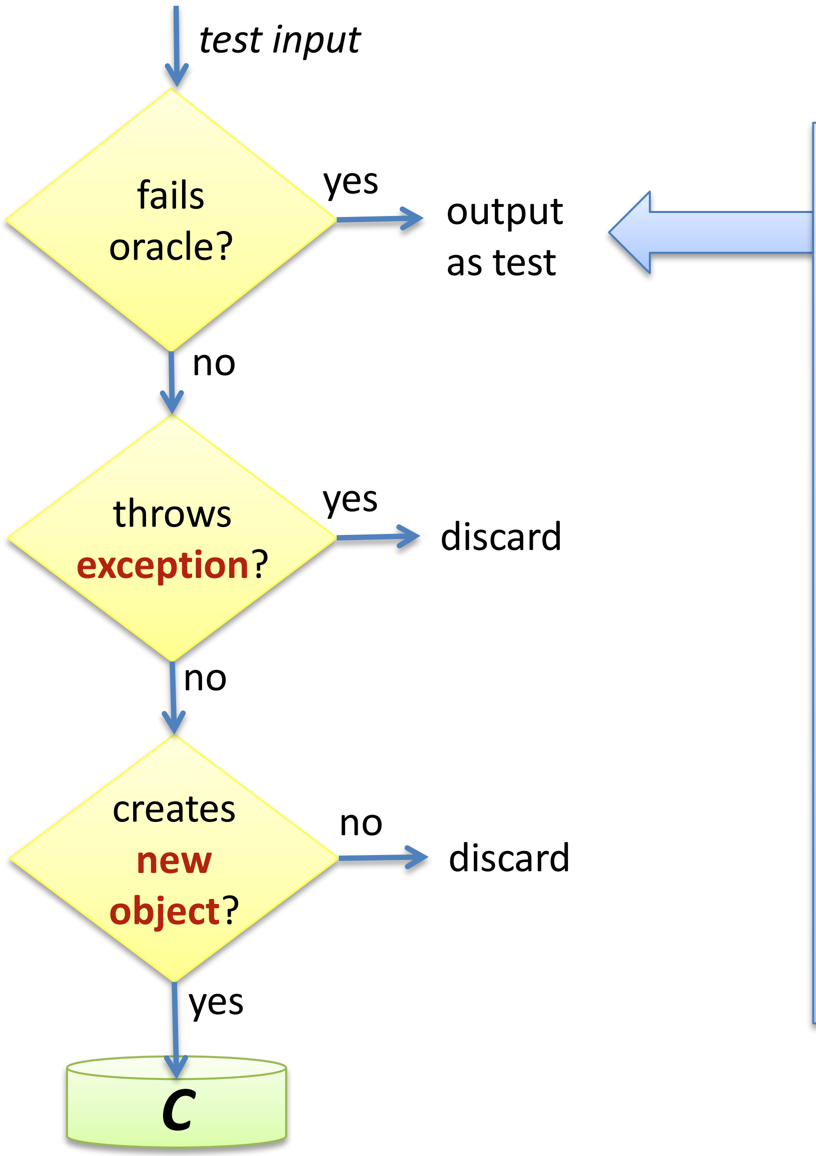
☞ *use behavior of **code under test** to guide generation*

Checkers are heuristic

- may discard useful inputs
- may not discard useless inputs

☞ *guidance need **not** be **perfect** to be **useful***

Randoop's heuristic guidance



test input

fails oracle?

yes

output as test

no

throws **exception**?

yes

discard

no

creates **new object**?

no

discard

yes

C

Built-in oracles

- based on published API

equals reflexive

equals symmetric

equals-hashCode

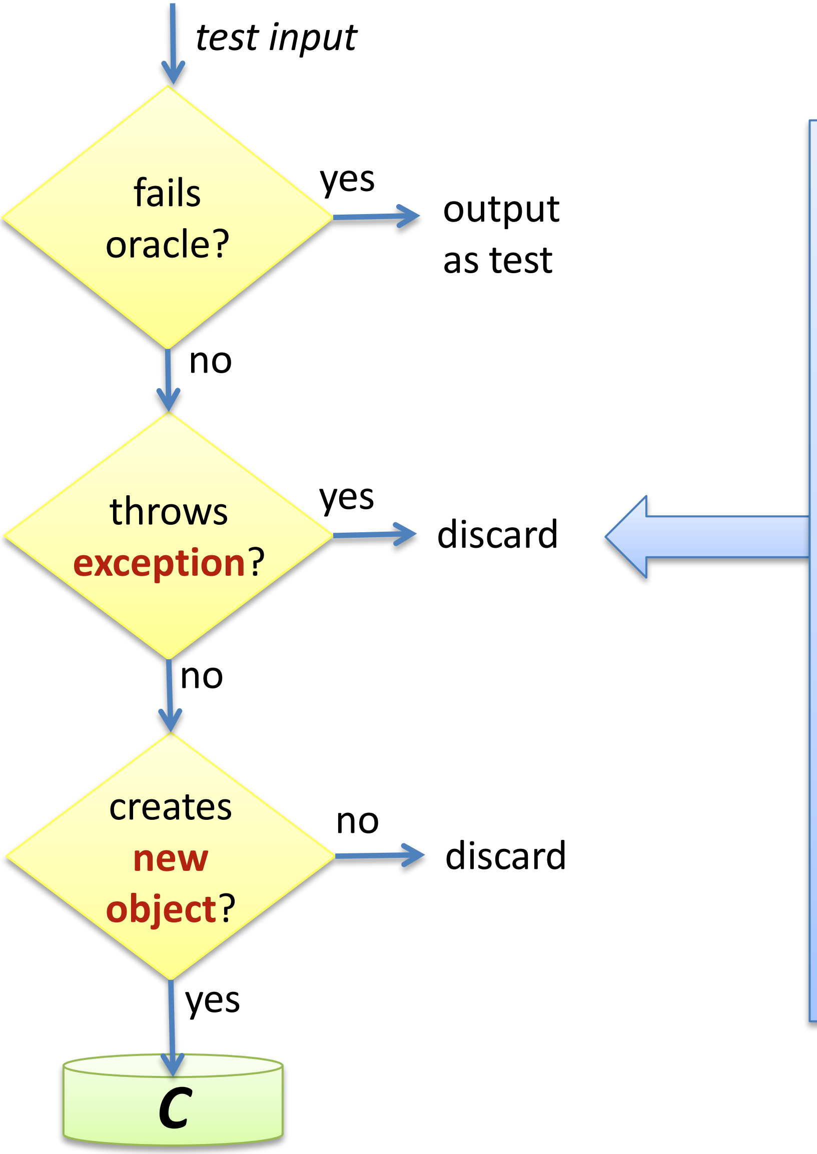
no NPE, if no null inputs

no assertion violation

`.NET only:`

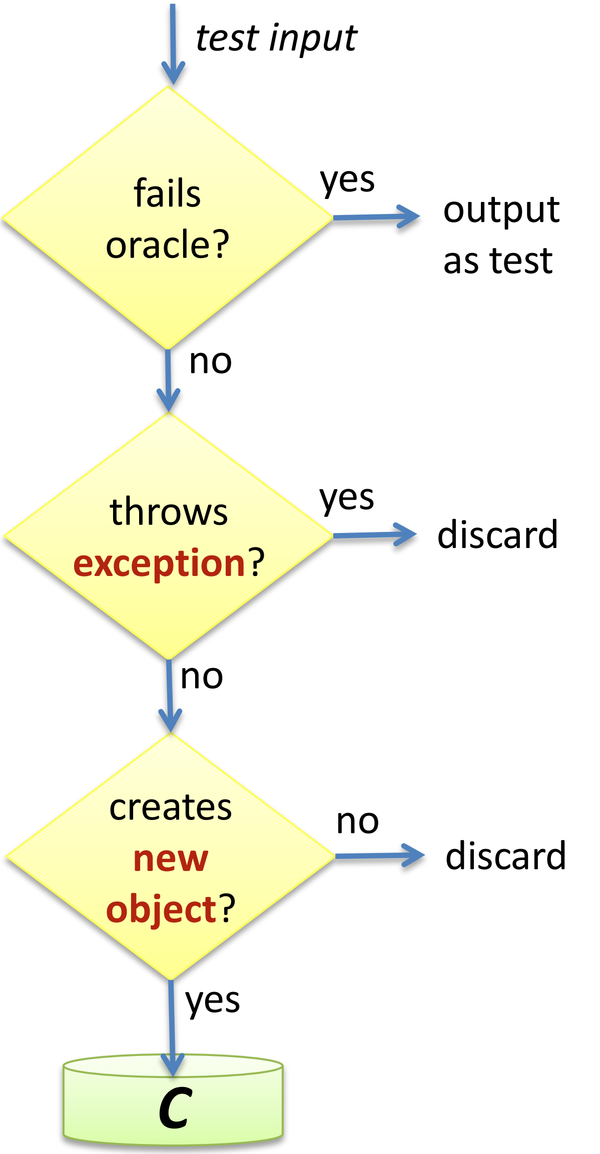
no `IllegalMemAccess` exception

Randoop's heuristic guidance



- Exceptions often indicate
  - illegal inputs to a method
  - unexpected environment
  - some other abnormal condition
- **Rarely useful to continue executing**

Randoop's heuristic guidance



- During execution, maintain **object set**
  - all objects created
  - across entire run
- New input redundant if
  - Creates an object equal to one already in object set

Revealing unknown errors

Applied Randoop to 13 libraries

- built-in oracles
- heuristic guidance
- default time limit
  (2 minutes/library)

				tests	errors
	library	**LOC**	**classes**	**output**	**revealed**
JDK	java.util	39K	204	20	6
	java.xml	14K	68	12	2
Apache commons project	chain	8K	59	20	0
	collections	61K	402	67	4
	jelly	14K	99	78	0
	logging	4K	9	0	0
	math	21K	111	9	2
	primitives	6K	294	13	0
.NET	mscorlib	185K	1439	19	19
	system.data	196K	648	92	92
	system.security	9K	128	25	25
	system.xml	150K	686	15	15
	web.services	42K	304	41	41
	TOTAL	***750K***	***4451***	***411***	***206***

Outputs one test per violating method

.NET libraries specification:

"no method should throw NPEs, assertion violations, or IllegalMemAccess exception"

Errors revealed

JDK

- 6 methods that create objects violating reflexivity of equality
- 2 well-formed XML objects cause **`hashCode/toString NPEs`**

Apache

- 6 constructors leave fields unset, leading to NPEs

.NET

- 175 methods throw forbidden exceptions
- 7 methods that violate reflexivity of **`equals`**

.NET

- library hangs given legal sequence of calls

without guidance

none revealed

66% fewer revealed

70% fewer revealed

not revealed

JCrasher

JCrasher [Csallner 04]

- random unit test generator (Java)
- reports exceptions
- augmented with Randoop's properties

results

- reported 595 error test cases
- but only 1 actual error
- compare 14 found by Randoop (for Java libraries)

IllegalArgumentException	332
NullPointerException	166
ArrayIndexOutOfBoundsException	77
MissingResourceException	8
ClassCastException	6
NegativeArraySizeException	3
NumberFormatException	2
IndexOutOfBoundsException	2
RuntimeException	1
IllegalAccessError	1

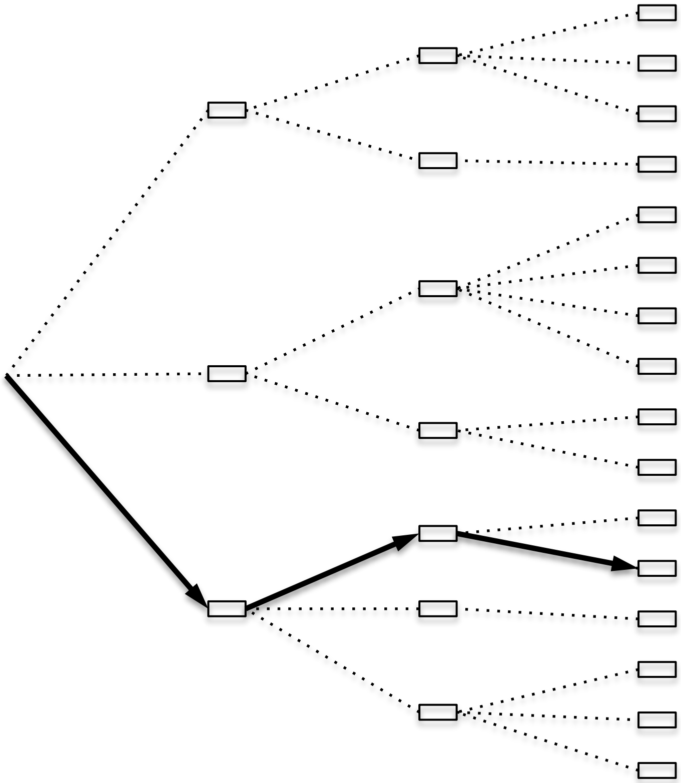
Why is Randoop more effective?

- Prunes **useless** inputs
- Generates **longer** tests

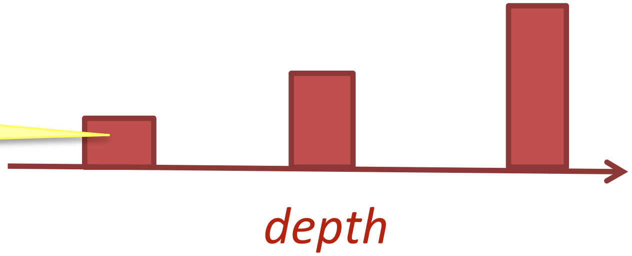
Test length vs. effectiveness

random testing is more effective when generating **long chains** of operations

```
m=Mono(1,1,1)

p=Poly()

p2=p.add(m)
```



chances of an operation revealing an error



Experiment

Random walk generator

- start from empty sequence
- take random steps
- restart if error or exception

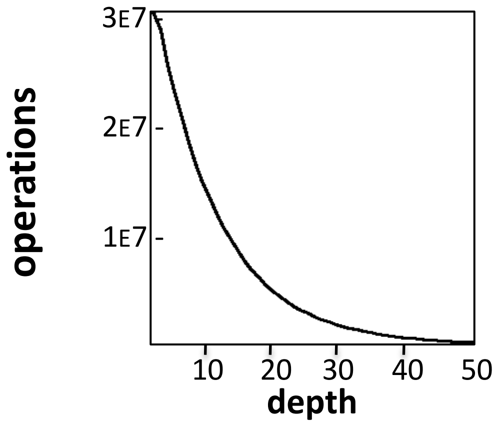
10M operations per library

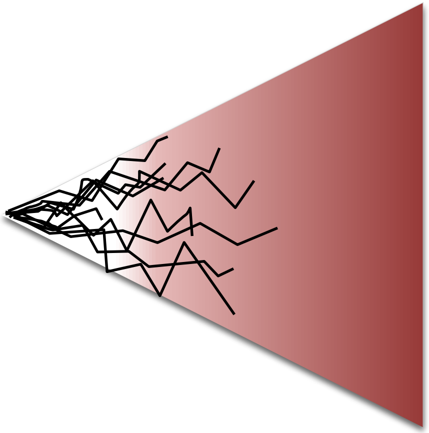
- several days

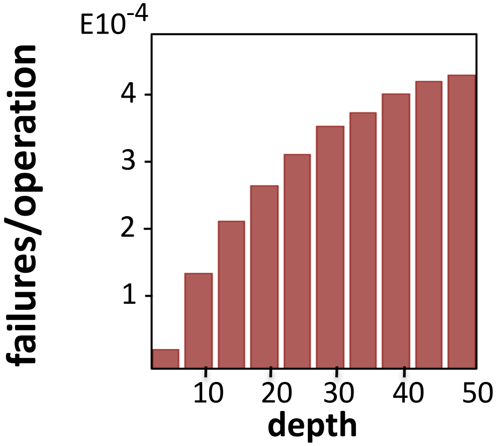
library	classes	LOC
java.util	204	39K
collections	402	61K
primitives	294	6K
trove	336	87K
jace	164	51K

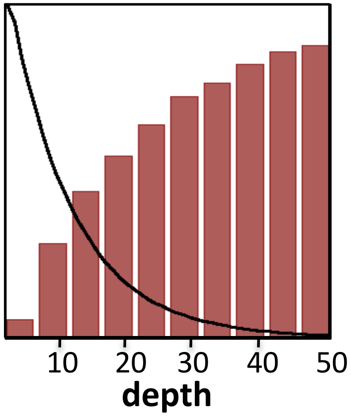
Results

cannot create long chains (due to exceptions),
but failure rate is higher at greater depths
→ performs most operations where failure rate is lowest









Randoop

Goal: evaluate benefits of pruning

- legality
- equivalence

Reran experiment two more times

1. Randoop

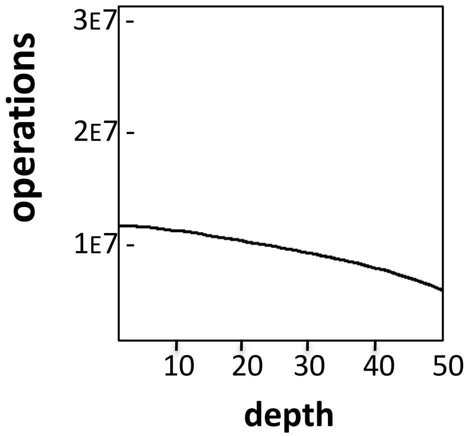
- only legality checks
  (no equals checks)

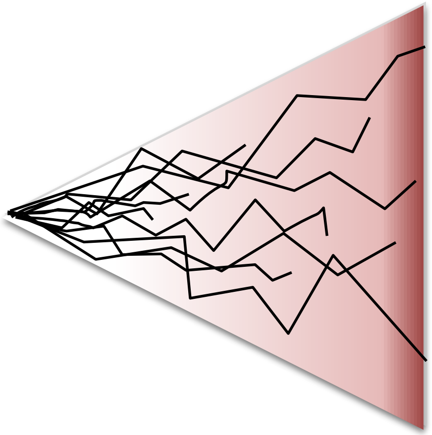
2. Randoop

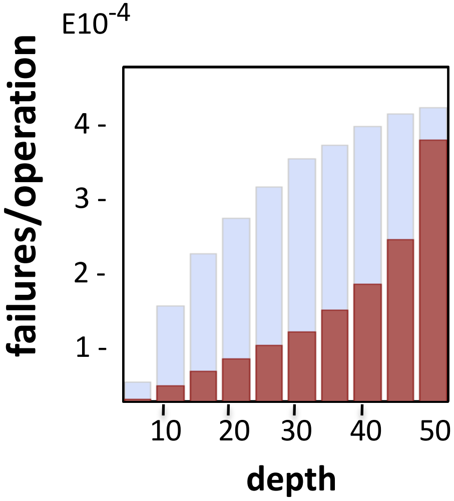
- all checks

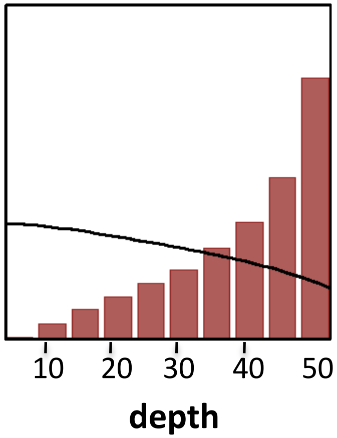
Randoop (only legality checks)

can create long chains
but lower failure rate (repetitive sub-chains)



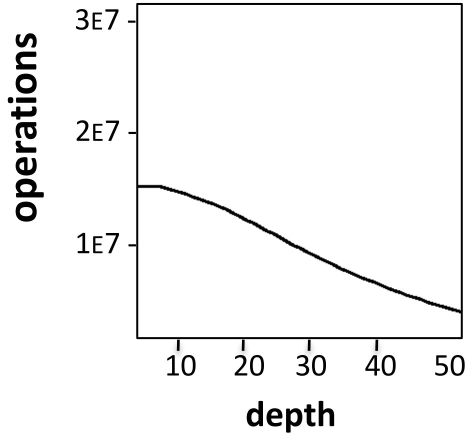


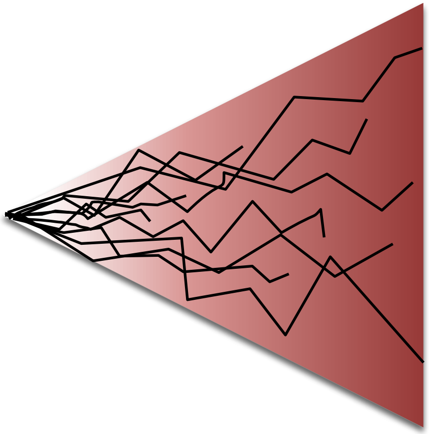


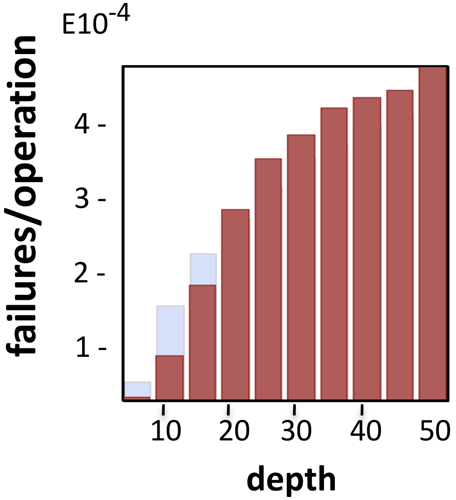


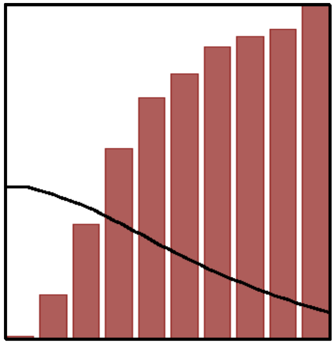
Randoop (full checks)

best of both worlds:
long chains, **and** high failure rate (equivalence pruning)









Errors revealed

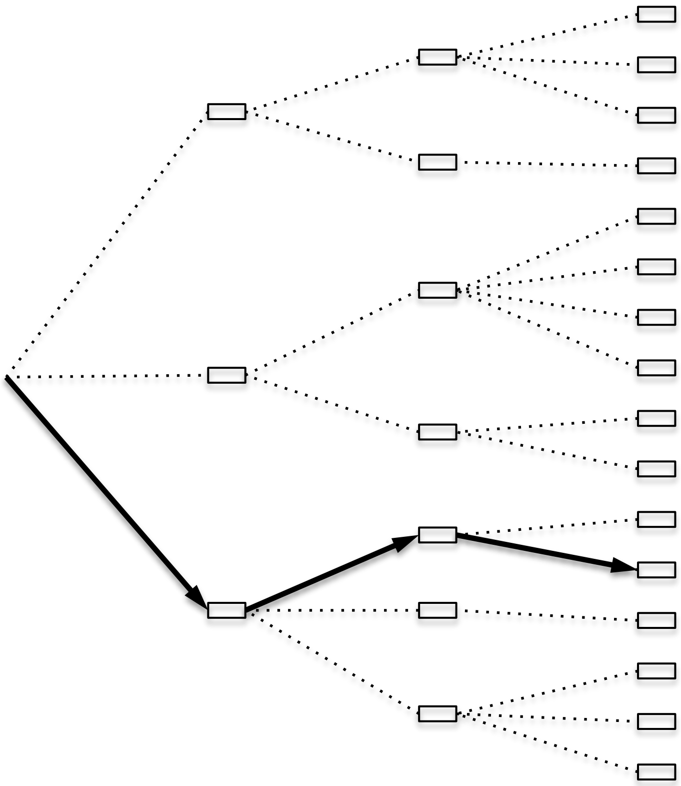
library	random walk	Randoop only leg.	Randoop
java.util	20	21	27
collections	28	37	48
primitives	16	13	19
trove	20	27	27
jace	15	15	26
TOTAL	**99**	**113**	**147**

(assume errors associated 1-1 with violating methods)

Systematic testing

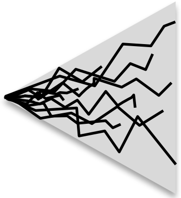
JPF (model checker for Java)

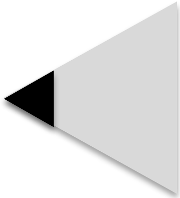
- Breadth-first search
- Depth-first search
- max seq. length 10

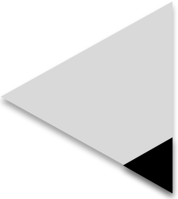


Results

	Randoop (2 minutes/library)		JPF, BFS (200 minutes/library)		JPF, DFS (200 minutes/library)	
	tests output	distinct errors	tests output	distinct errors	tests output	distinct errors
JDK	32	8	0	0	24	0
Apache	187	6	0	0	79	1
TOTAL	***219***	***14***	***0***	***0***	***103***	***1***







For large libraries,
random, sparse sampling
can be more effective than
dense, local sampling

Container data structures

benchmarks from Visser et al. (2006)

block coverage achieved by 7 techniques on 4 data structures

		data structure			
		bintree	binheap	fibheap	RBT
technique	model checking (MC)	78%	77%	80%	69%
	MC/state matching	78%	77%	96%	72%
	MC/abstract matching	78%	95%	100%	72%
	symbolic execution (SE)	78%	95%	96%	72%
	SE/abstract matching	78%	95%	100%	72%
	random testing	78%	95%	100%	72%
	Randoop (DRT)	**78%**	**95%**	**100%**	**72%**

Randoop achieves coverage in:

- 1/3 time of systematic techniques
- 3/4 time of random testing

similar results for other techniques/containers

- BET [Marinov 2003]
- symstra (symbolic execution) [Xie 2005]
- rostra (exhaustive enumeration) [Xie 2004]

Outside the laboratory

Assess effectiveness in industrial setting

- Error-revealing effectiveness
- cost effectiveness
- Usability

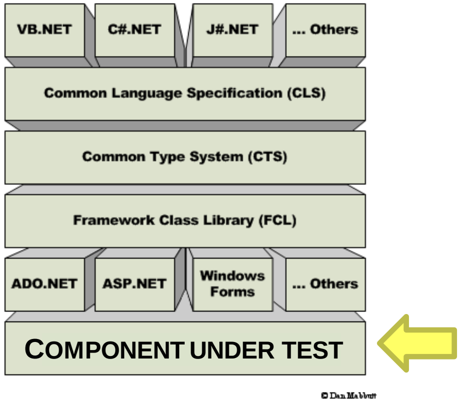
Case study

- Microsoft test team
- used Randoop to check:
  - assertion violations
  - invalid memory accesses
  - program termination
- used tool for 2 months
- met with team every 2 weeks
  - gather experience and results

Subject program

core .NET component library

- 100KLOC
- large API
- low on .NET framework stack
- used by all .NET applications



highly stable

- high reliability critical
- 200 person-years testing (40 testers over 5 years)
- presently, ~20 new errors **per year**

many techniques applied

- Manual testing
- Random testing

Study statistics

Human time interacting with Randoop	15 hours
CPU time running Randoop	150 hours
Total distinct method sequences	4 million
New errors revealed	30

- interacting with Randoop
- inspecting the resulting tests
- discarding redundant failures

Randoop

- 30 new errors in 15 hours of human effort
- 1 new error for ½ hour effort

existing team methods

- 20 new errors per year
- 1 new error for 100 hours effort

Example errors

error in code with 100% branch coverage

- component has memory-managed and native code
- if native code manipulates references, must inform garbage collector
- native code informed GC of new reference, gave invalid address

error in component *and* test tool

- on exception, component looks for message in a resource file
- rarely-used exception missing message in file
- lookup led to assertion violation
- two errors:
  - missing message in resource file
  - in tool that tested resource file

concurrency errors

- used Randoop test inputs to drive concurrency testing tool

Other techniques did not reveal the errors

Random

fuzz testing

- files
- protocols

Different domain

Static method sequence generation

- a la JCrasher
- longer methods required

Systematic

symbolic-execution based unit test generator

- developed at MSR
- conceptually more powerful than Randoop

no errors over the same period of time

achieved higher coverage on classes that

- can be tested in isolation
- do not go beyond managed code realm

later version *has* revealed errors

Coverage plateau

- initial period of high effectiveness

- eventually, Randoop ceased to reveal errors

- After the study
  - test team made a parallel run of Randoop
  - dozens of machines
  - different random seeds
  - Found <10 errors

- Randoop unable to cover some code

Summary

Directed random testing

- random testing + pruning
- fully automated
- scalable

Reveals errors

- large, widely-used libraries
- outperforms systematic testing (sparse sampling)
- outperforms random testing (pruning, long sequences)

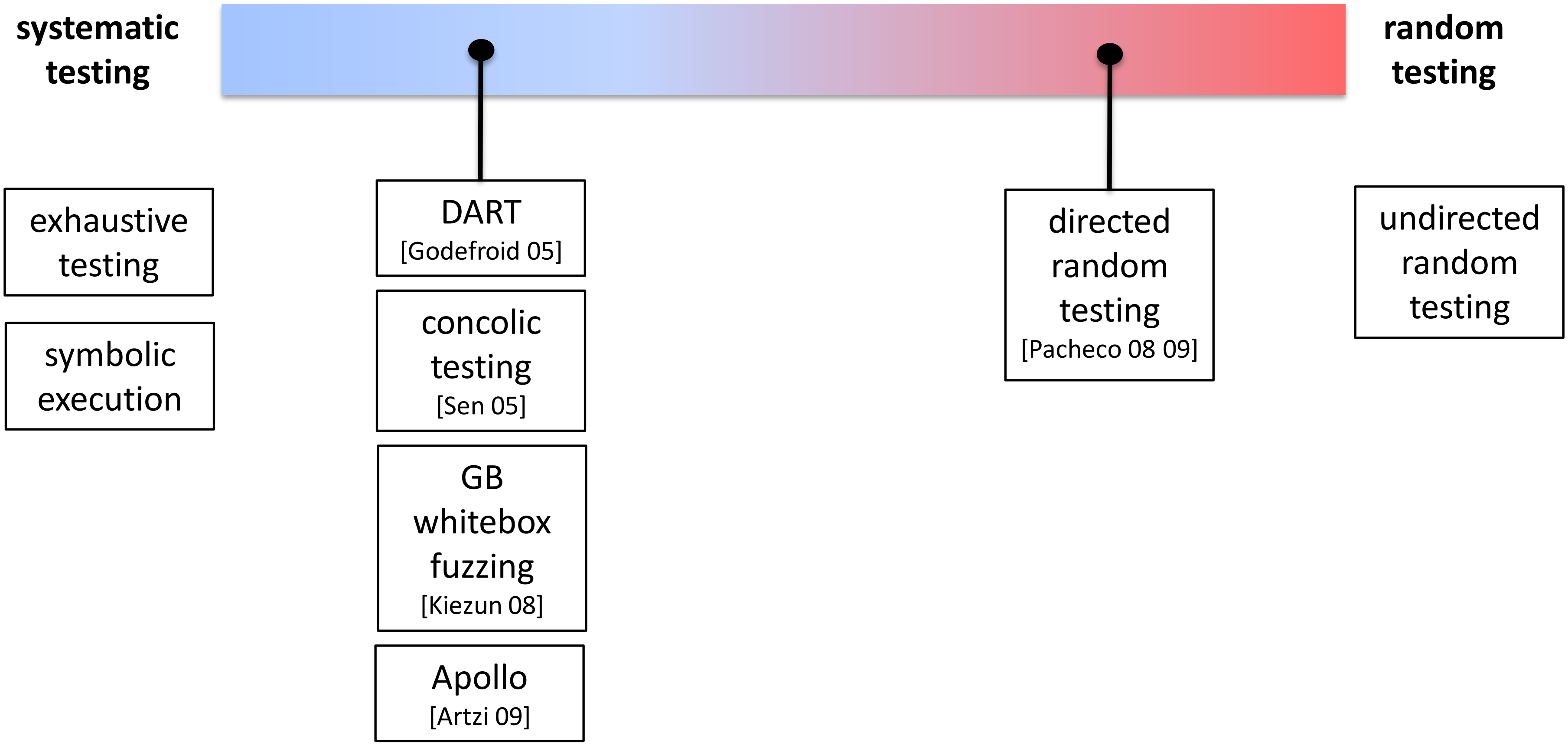
Is cost effective

- can increase productivity 100-fold

Conclusion: a spectrum of testing techniques

Directed Random Testing:

a promising point in the spectrum



systematic testing

random testing

exhaustive testing

symbolic execution

DART [Godefroid 05]

concolic testing [Sen 05]

GB whitebox fuzzing [Kiezun 08]

Apollo [Artzi 09]

directed random testing [Pacheco 08 09]

undirected random testing