Counterexample-Driven Genetic Programming

Krzysztof Krawiec Iwo Błądek Jerry Swan

Poznan University of Technology, Poland
The University of York, UK

UNIVERSITY of York

GECCO, Berlin, July 18, 2017

Synthesizing programs from examples

GP is for synthesizing programs *from examples* (*input-output pairs, tests*)

- A natural means of expressing user's intent

However:

- No guarantee of correctness.
- The sample of training examples must be large enough to facilitate generalization.
  - Curse of dimensionality
- No formal theory of generalization, e.g., PAC-like (Probably Approximately Correct, Valiant 1984)
- Correctness is essential in safety-critical systems, security, transportation, ...

Synthesizing programs from specifications

The alternative: Spec-based synthesis

- Programs are being synthesized *from a contract*
- Contract = a pair of logical clauses specifying the desired behavior of the program by constraining its inputs and outputs

- **pre**(in) – precondition, constrains the input to the program.

$$e.g. \quad in_1 \geq 0 \ \wedge \ in_2 \geq 0$$

- **post**(in, out) – postcondition, constrains the corresponding output.

$$e.g.\ out \geq in_1 \wedge out \geq in_2 \wedge (out = in_1 \vee out = in_2)$$

Contracts can be turned into *satisfiability problems* and fed into a SAT/SMT solver to perform synthesis:

$$\exists_p \forall_{in} : \mathbf{pre}(in) \implies \mathbf{post}(in, p(in))$$

Satisfiability Modulo Theories (SMT) Solvers

Answer the question: Is the given logical formula satisfiable *under the theory* T, which defines semantics of a certain set of functions?

Examples: $T =$ `QF_LIA` (Quantifier-Free Linear Integer Arithmetic)

Variables: $x, y, z \in \mathbb{Z}, a \in \{false, true\}$

$(10 \cdot x = 20) \ \wedge \ a$
SAT: $x = 2, \ a = true$

$(x < y) \ \wedge \ (y < z) \ \wedge \ (z < x)$
UNSAT

$(x \leq y) \ \wedge \ (y \leq z) \ \wedge \ (z \leq x)$
SAT: $x = 0, \ y = 0, \ z = 0$

Spec-based synthesis

Contracts can be turned into *satisfiability problems* and fed into a SAT/SMT solver to perform synthesis:

$$\exists_p \forall_{in} : \mathbf{pre}(in) \implies \mathbf{post}(in, p(in))$$

Pros:

- The synthesized program is provably correct.

Cons:

- High computational cost (satisfiability is NP-complete)

Question 1

Can we use GP to guide spec-based synthesis?

Counterexample-Driven Genetic Programming (CDGP)

Idea 1

Use GP as *program generator* and employ *program verification*.

$$\forall_{in} : \mathbf{pre}(in) \implies \mathbf{post}(in, p(in))$$

Equivalently:

$$\exists_{in} : \mathbf{pre}(in) \not\Longrightarrow \mathbf{post}(in, p(in))$$

Problem: SMT solvers provide only *binary feedback* on a given program.

- Not useful for guiding search.

Question 2

How to elicit fine-grained fitness from verification?

Counterexample-Driven Genetic Programming (CDGP)

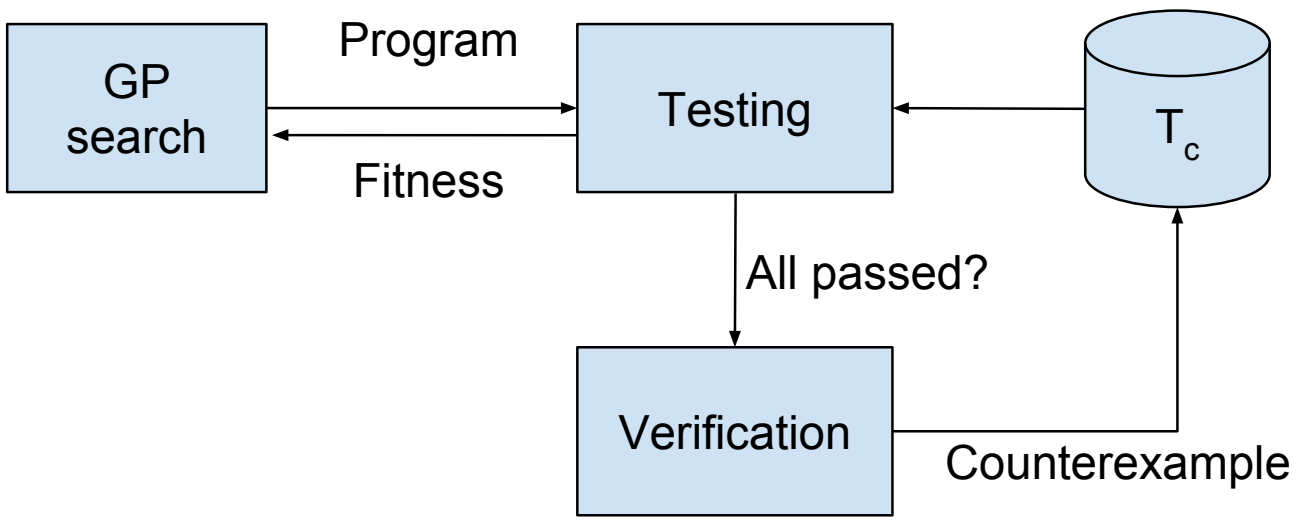
Idea 2

Use *counterexamples*.

- Counterexample = (in,out) pair returned by a solver together with 'SAT'

In evaluation phase:

- Verify programs and collect the resulting counterexamples
- Calculate programs' fitness on the collected counterexamples



CDGP (conservative variant)

```
1:  procedure CDEval(P, T_c, (pre, post))
2:      T ← ∅
3:      for all p ∈ P do
4:          p.eval ← Eval(p, T_c)
5:          if p.eval = |T_c| then
6:              c ← Verify(p, (pre, post))
7:              if c = UNSAT then return p
8:              else
9:                  T ← T ∪ {c}
10:             end if
11:         end if
12:     end for
13:     return (P, T_c ∪ T)
14: end procedure
```

CDGP (conservative variant)

```
1:  procedure CDEval(P, T_c, (pre, post))
2:      T ← ∅
3:      for all p ∈ P do
4:          p.eval ← Eval(p, T_c)
5:          if p.eval = |T_c| then
6:              c ← Verify(p, (pre, post))
7:              if c = UNSAT then return p
8:              else
9:                  T ← T ∪ {c}
10:             end if
11:         end if
12:     end for
13:     return (P, T_c ∪ T)
14: end procedure
```

- T_c updated once per generation.
- With growth of T_c, evaluation becomes more fine-grained.
- T is a set $\implies$ duplicates are eliminated.

CDGP (non-conservative variant)

```
1:  procedure CDEval(P, T_c, (pre, post))
2:      T ← ∅
3:      for all p ∈ P do
4:          p.eval ← Eval(p, T_c)
5:
6:          c ← Verify(p, (pre, post))
7:          if c = UNSAT then return p
8:          else
9:              T ← T ∪ {c}
10:         end if
11:
12:     end for
13:     return (P, T_c ∪ T)
14: end procedure
```

- Verifies *all* evaluated programs.

Related work

- Model-checking in fitness function, applied to synthesize a state-machine controller for a vending machine (Johnson 2007)
- Competitive coevolution of programs with tests (Arcuri & Yao 2007)
- Combining model-checking and GP (Katz & Peled, 2008, 2016)
- Incorporating formal approaches in Genetic Improvement (Burles et al. 2015, Kocsis et al. 2016, Kocsis & Swan 2017)

Experiment

Goal: Assess CDGP on a range of spec-based synthesis benchmarks.

Benchmarks from the Syntax Guided Synthesis (SyGuS) competition

- Signature: $\texttt{I}^n \rightarrow \texttt{I}$
- SMT Theory: Linear Integer Arithmetic (LIA)

```
I ::= I + I | I - I | ite(B,I,I) | -1 | 0 | 1 | v1 | ...  | vn
B ::= and(B,B) | or(B,B) | not(B) | B = B
      | I < I | I <= I | I = I | I >= I | I > I
```

Name	*Arity*	*Semantics*
CountPos	2, 3	The number of positive arguments
IsSeries	3	Do arguments form an arithmetic series?
IsSorted	4	Are arguments in ascending order?
Max	2, 4	The maximum of arguments
Median	3	The median of arguments
Range	3	The range of arguments
Search	2, 4	The index of an argument
Sum	2, 4	The sum of arguments

Experiment

Compared methods:

- Conservative CDGP
- Non-conservative CDGP
- Baseline: GPR: Are random tests are better than counterexamples?
  - Like conservative CDGP, but collects *random tests* in T_c, drawn uniformly from $(-100, 100)^n$

The archive T_c may grow slowly, so we consider also:

- Steady-state algorithm (to add tests more frequently)
- Lexicase selection (to improve diversity)

Some details on setup:

- T_c allowed to grow indefinitely
- Doubled population size for GPR (1000)

https://github.com/kkrawiec/CDGP 

Results (budget = 100 generations)

Success rate:

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	1.00	1.00	1.00	1.00	0.93	1.00	0.83	1.00	0.97	1.00	0.97	1.00
CountPos3	0.77	1.00	0.60	1.00	0.07	0.63	0.03	0.53	0.40	1.00	0.27	1.00
IsSeries3	0.43	1.00	0.43	1.00	0.37	1.00	0.53	0.93	0.00	1.00	0.20	0.00
IsSorted4	0.70	1.00	0.67	1.00	0.27	0.77	0.33	0.93	0.43	1.00	0.33	0.57
Max2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Max4	0.83	1.00	0.97	1.00	0.23	0.97	0.20	0.93	1.00	1.00	0.93	1.00
Median3	0.83	1.00	0.73	1.00	0.20	0.80	0.10	0.87	0.80	1.00	0.80	1.00
Range3	0.47	1.00	0.53	1.00	0.33	0.87	0.17	0.70	0.80	1.00	0.67	0.92
Search2	1.00	1.00	1.00	1.00	0.73	0.97	0.63	1.00	0.57	1.00	0.47	0.65
Search4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sum2	0.77	1.00	0.73	1.00	0.10	0.27	0.20	0.23	0.67	1.00	0.75	1.00
Sum4	0.00	0.12	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Time:

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	37.4	14.7	73.7	49.8	6.5	7.3	30.4	31.1	263.0	636.5	2070.3	3157.1
CountPos3	341.2	98.7	597.7	505.4	39.2	86.7	76.1	281.5	2587.7	4304.5	5102.4	8048.3
IsSeries3	201.4	37.7	447.6	229.4	20.1	20.3	48.2	110.0	783.8	66255.9	2556.3	86402.5
IsSorted4	224.0	77.7	675.4	890.8	29.6	123.6	77.5	179.9	1213.2	13462.2	4256.9	55551.4
Max2	6.0	5.6	29.7	26.9	2.4	2.8	22.4	22.3	16.5	148.2	1532.4	1625.0
Max4	433.8	86.3	1136.2	908.2	33.6	71.3	73.2	192.1	367.2	353.2	2333.0	4383.1
Median3	274.7	82.5	661.7	555.0	34.5	89.4	70.4	229.8	731.7	788.4	2743.3	4143.5
Range3	579.5	100.6	934.9	587.0	23.5	37.1	56.6	158.0	991.0	3466.8	2528.4	20216.0
Search2	109.6	17.9	224.3	90.4	12.9	14.3	40.9	63.8	1218.2	11198.3	3503.4	44266.3
Search4	1326.8	23643.2	1990.8	54181.6	36.7	199.6	71.5	861.9	1539.3	67320.7	2392.8	86402.4
Sum2	151.6	48.5	395.3	330.9	23.7	92.3	51.6	209.9	1049.9	4946.0	2583.9	16110.5
Sum4	5567.0	32720.5	7527.6	50260.7	36.8	224.7	77.6	547.2	1952.0	7575.2	3733.3	22180.8

Results (budget = time)

Success rate:

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.97	0.83	0.60	0.00	0.00
CountPos3	0.80	1.00	0.63	1.00	0.67	0.97	0.60	0.87	0.13	0.67	0.00	0.00
IsSeries3	0.93	1.00	0.73	1.00	0.90	1.00	0.97	0.97	0.27	0.00	0.03	0.00
IsSorted4	0.97	1.00	0.90	0.97	0.97	1.00	0.97	0.93	0.50	0.03	0.07	0.00
Max2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.93	0.03	0.00
Max4	0.83	1.00	0.33	0.63	1.00	1.00	0.97	1.00	0.83	0.90	0.00	0.00
Median3	0.83	1.00	0.70	0.73	0.83	1.00	0.67	0.93	0.77	0.97	0.00	0.00
Range3	0.73	1.00	0.73	0.97	0.63	1.00	0.63	0.83	0.50	0.53	0.07	0.00
Search2	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00	0.60	0.10	0.00	0.00
Search4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sum2	0.97	1.00	1.00	1.00	0.50	0.43	0.13	0.27	0.43	0.10	0.20	0.03
Sum4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Size of T_c:

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	60.6	55.9	59.2	48.2	24.9	23.1	24.1	24.7	1018.0	990.8	988.3	986.6
CountPos3	429.6	305.3	443.4	283.3	53.0	56.6	54.5	57.1	1000.1	1041.2	1000.0	999.9
IsSeries3	335.4	262.4	361.7	218.8	40.6	45.9	37.3	44.7	1109.4	1027.2	1001.3	1000.8
IsSorted4	753.9	520.8	647.5	409.1	64.8	75.5	60.7	74.1	1837.9	1000.0	1006.3	1000.0
Max2	36.5	41.4	30.2	28.2	24.6	23.6	23.2	23.3	987.6	988.6	987.9	987.7
Max4	1092.5	674.2	847.5	685.9	63.7	60.9	63.7	72.6	1077.2	1074.5	1000.0	1000.0
Median3	474.1	288.5	371.2	290.3	51.1	54.4	51.2	55.0	1023.1	1086.5	1000.0	999.9
Range3	527.9	277.0	482.3	274.7	39.6	42.9	39.6	41.1	1050.0	1007.3	999.9	999.9
Search2	143.1	108.5	149.7	104.0	29.1	29.9	28.6	29.1	1003.2	999.9	999.9	999.9
Search4	1137.3	805.9	988.3	578.8	85.1	237.3	77.0	78.0	1000.0	1000.0	1000.0	1000.0
Sum2	396.5	286.0	303.0	273.7	34.7	35.2	31.1	31.1	991.8	987.7	990.3	989.0
Sum4	1625.3	4045.1	1183.8	1362.7	52.5	85.9	55.4	70.7	1000.0	1000.0	1000.0	1000.0

Experiment summary

Major observations:

- Overall winner: CDGP conservative + generational + lexicase
- Both CDGP variants better than GPR time-wise, despite:
  - using a costly SMT solver
  - working with fewer tests (dozens/hundreds vs. roughly 1000)
- Counterexamples more useful than random tests.
  - They identify informative weak points of candidate programs.

Minor observations:

- Conservative (almost) on par with non-conservative when given the same time, but not so when given the same number of generations
  - Too few tests collected?
- Generational better than steady-state (particularly when budget=time)
  - Additional cost of evaluation on the newly created tests
- Lexicase better than tournament selection $\implies$ Inspecting per-test behavior helpful.

Discussion

Observations:

- Relation to test-driven development.
- CDGP gradually transforms a *spec-based problem* into a *test-based problem*.

Broader context:

- A generic way of eliciting gradient from search problems.
- SMT solvers as a means of making GP better informed.

Potential extensions:

- Seeding search process with examples
- Problems with partial specifications
- Search operators informed by verification outcomes

Long-term goal:
Make CDGP more efficient than the conventional spec-based synthesis methods.

Conclusions

Take-home messages:

- GP *can* be used for specification-based synthesis and produce provably correct programs.
- A whole gamut of new application areas and research directions.

Thank You

UNIVERSITY of York

KK and IB acknowledge support from grant 2014/15/B/ST6/05205
funded by the National Science Centre, Poland.

NATIONAL SCIENCE CENTRE
POLAND

SMT Solvers

SMT Solver – checks satisfiability of formulas modulo the given theory.

- CVC4, MathSAT, Z3, ...
- SMT-LIB – standardized language for interacting with solvers

Specification of 4-ary *max* function in SMT-LIB:

```
(set-logic LIA)
(synth-fun max4 ((x Int) (y Int) (z Int) (w Int)) Int)

(declare-var x Int)
(declare-var y Int)
(declare-var z Int)
(declare-var w Int)

(constraint (>= (max4 x y z w) x))
(constraint (>= (max4 x y z w) y))
(constraint (>= (max4 x y z w) z))
(constraint (>= (max4 x y z w) w))
(constraint (or (= x (max4 x y z w))
        (or (= y (max4 x y z w))
        (or (= z (max4 x y z w))
        (= w (max4 x y z w))))))
```