# Introduction to Syntax-Guided Synthesis

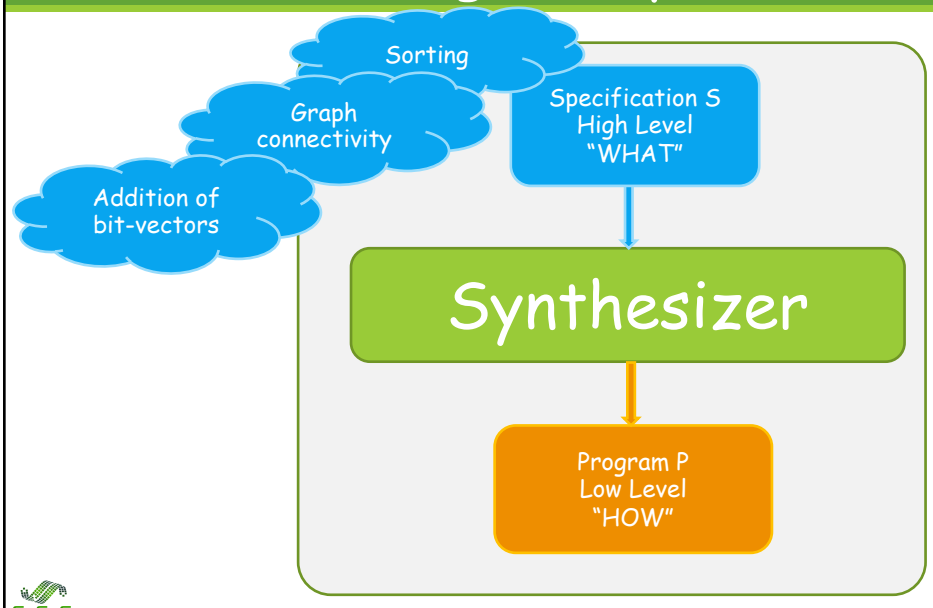

Dana Fisman

SAT/SMT/AR/CP Summer School

## Outline

- SyGuS-The Problem
  - Motivation for Syntax-Guided Synthesis
  - Recent trends in program synthesis
  - SyGuS within the big picture
  - Formalization of Syntax-Guided Synthesis
- Solving SyGuS
  - CEGIS – Counterexample Guided Inductive Synthesis
  - Search by enumeration
  - Symbolic search
  - Stochastic search
  - A glance at competition results

# Program Synthesis

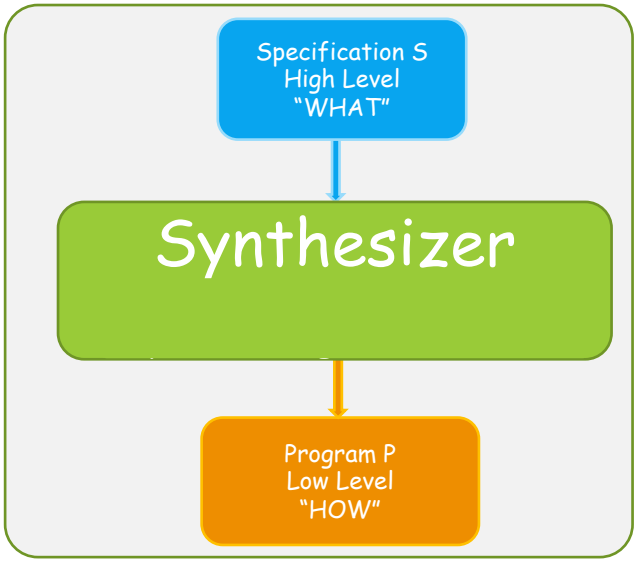



Too good to be true...

SyGuS

3

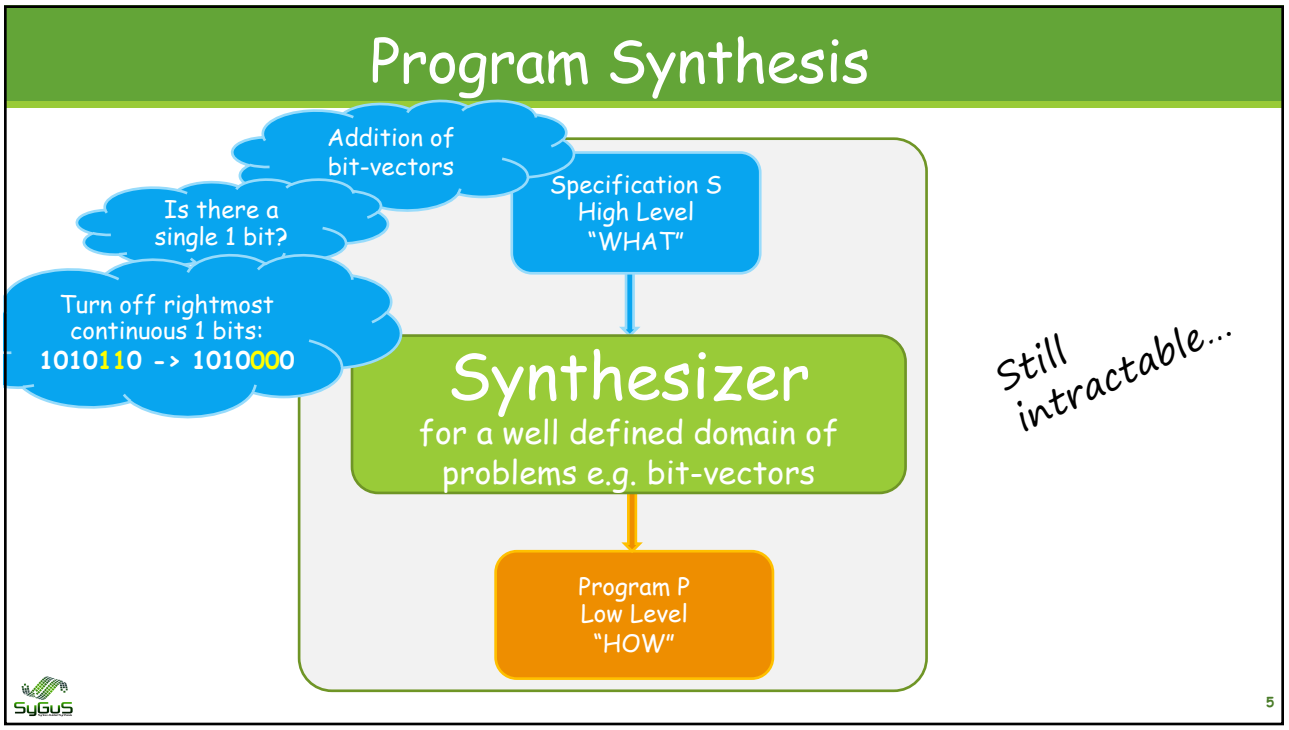
# Program Synthesis

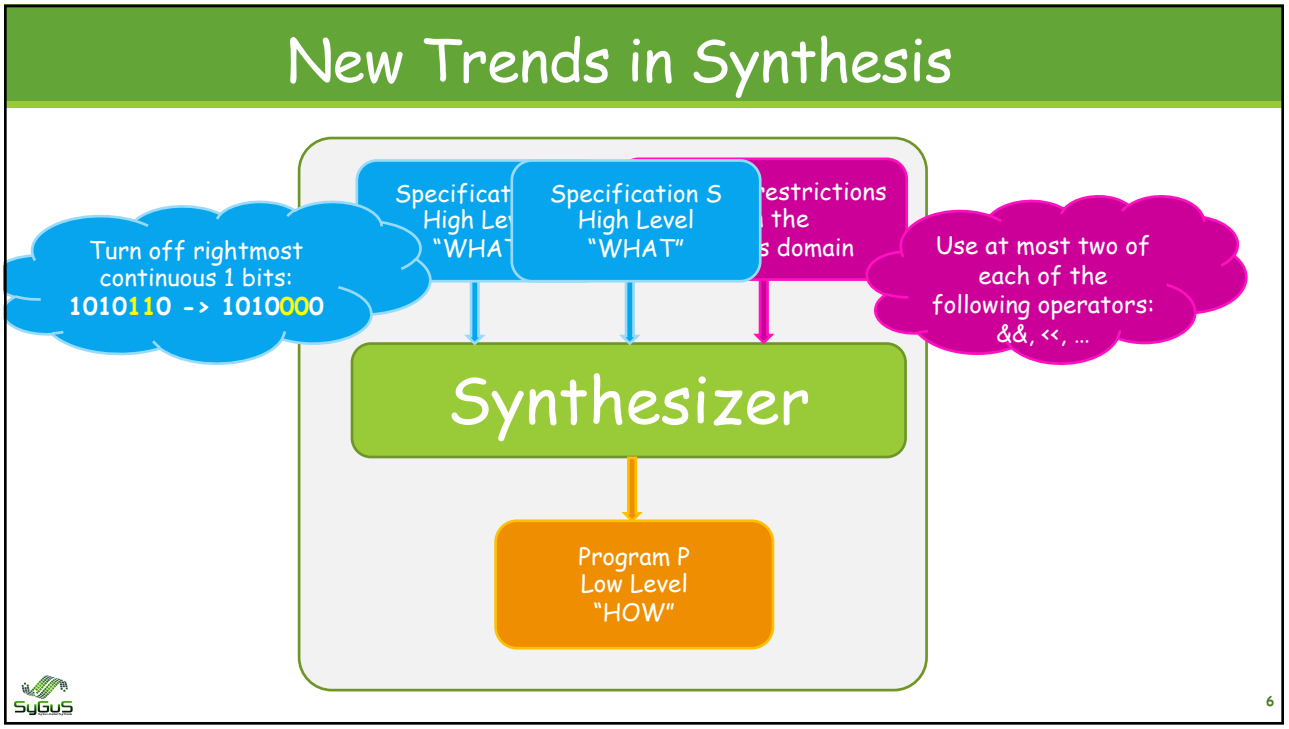



A bit more realistic

SyGuS

4


Program Synthesis
Addition of bit-vectors
Is there a single 1 bit?
Turn off rightmost continuous 1 bits:
1010110 -> 1010000
Specification S
High Level
"WHAT"
Synthesizer
for a well defined domain of problems e.g. bit-vectors
Program P
Low Level
"HOW"
Still intractable…
SyGuS
5


New Trends in Synthesis
Turn off rightmost continuous 1 bits:
1010110 -> 1010000
Specification S
High Level
"WHAT"
Use at most two of each of the following operators:
&&, <<, …
Synthesizer
Program P
Low Level
"HOW"
SyGuS
6

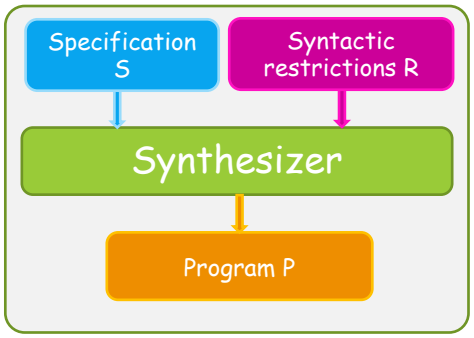
# New Trends in Synthesis

Motivation:

- Tractability
- Combine

  human expert insights with

  computers exhaustiveness & rapidness
- Benefit progress of SAT & SMT Solvers




SyGuS

7

# Ex 1. Parallel Parking By Sketching

The challenge is finding the parameters

```
Err = 0.0;
for(t = 0; t<T; t+=dT){









  simulate_car(car);
  Err += check_collision(car);
}
Err += check_destination(car);
```


Structure of the program is known

[Chaudhuri & Solar-Lezama PLDI 2010]

SyGuS

# Ex 1. Parallel Parking By Sketching

The challenge is finding the parameters

When to start turning?

How much to turn?

```
Err = 0.0;
for(t = 0; t<T; t+=dT){
  if(stage==STRAIGHT){ // (1) Backup straight
    if(t > ??) stage= INTURN;
  }
  if(stage==INTURN){ // (2) Turn
    car.ang = car.ang - ??;
    if(t > ??) stage= OUTTURN;
  }
  if(stage==OUTTURN){ // (3) Straighten
    car.ang = car.ang + ??;
    if(t > ??) break;
  }
  simulate_car(car);
  Err += check_collision(car);
}
Err += check_destination(car);
```


Structure of the program is known

[Chaudhuri & Solar-Lezama PLDI 2010]

# Ex 2. Optimizing Multiplications

## Superoptimizing Compiler

Given a program P, find a "better" equivalent program P'.

```
multiply (x[1,n], y[1,n]) {
   x1 = x[1,n/2];
   x2 = x[n/2+1, n];
   y1 = y[1, n/2];
   y2 = y[n/2+1, n];
   a = x1 * y1;
   b = shift( x1 * y2, n/2);
   c = shift( x2 * y1, n/2);
   d = shift( x2 * y2, n);
   return ( a + b + c + d)
}
```

Replace with equivalent code with only 3 multiplications

9

# Ex 3. Automatic Invariant Generation

Given a program P and a post condition S,
Find invariants $I_1$, $I_2$
with which we can prove program is correct

```
SelecionSort(int A[],n) {
  i1 :=0;
  while(i1 < n-1) {
    v1 := i1;
    i2 := i1 + 1;
    while (i2 < n) {
      if (A[i2]<A[v1])
        v1 := i2 ;
      i2++;
    }
    swap(A[i1], A[v1]);
    i1++;
  }
  return A;
}
```

Invariant: ???

Invariant: ???

post: ∀k : 0 ≤ k <n ⇒ A[k] ≤ A[k+1]

SyGuS



# Ex 3. Template-Based Invariant Generation

Given a program P and a post condition S
Find invariants $I_1$, $I_2$, … $I_k$
with which we can prove program is correct

```
SelecionSort(int A[],n) {
  i1 :=0;
  while(i1 < n-1) {
    v1 := i1;
    i2 := i1 + 1;
    while (i2 < n) {
      if (A[i2]<A[v1])
        v1 := i2 ;
      i2++;
    }
    swap(A[i1], A[v1]);
    i1++;
  }
  return A;
}
```

Invariant:
∀k1,k2. ??? ∧ ???

Invariant:
??? ∧ ??? ∧
(∀k1,k2. ??? ∧ ???) ∧
(∀k. ??? ∧ ?)

Constraint Solver

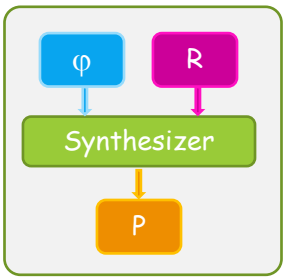
post: ∀k : 0 ≤ k <n ⇒ A[k] ≤ A[k+1]

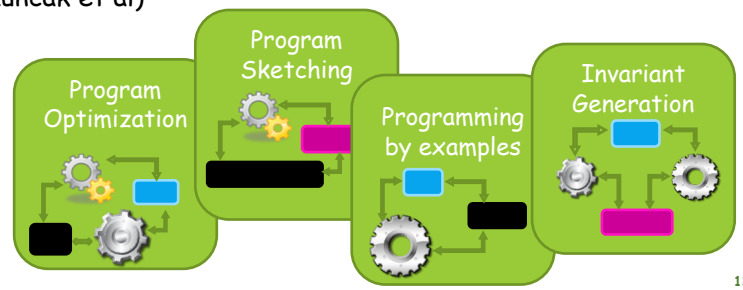
SyGuS

# Syntax-Guided Program Synthesis

- Common theme to many efforts ~ 20010-2015
  - Sketch (Bodik, Solar-Lezama et al)
  - FlashFill (Gulwani et al)
  - Super-optimization (Schkufza et al)
  - Invariant generation (Many recent efforts…)
  - TRANSIT for protocol synthesis (Udupa et al)
  - Oracle-guided program synthesis (Jha et al)
  - Implicit programming: Scala^Z3 (Kuncak et al)
  - AutoProf (Singh et al)

But no way to have
a generic solver for all ☹

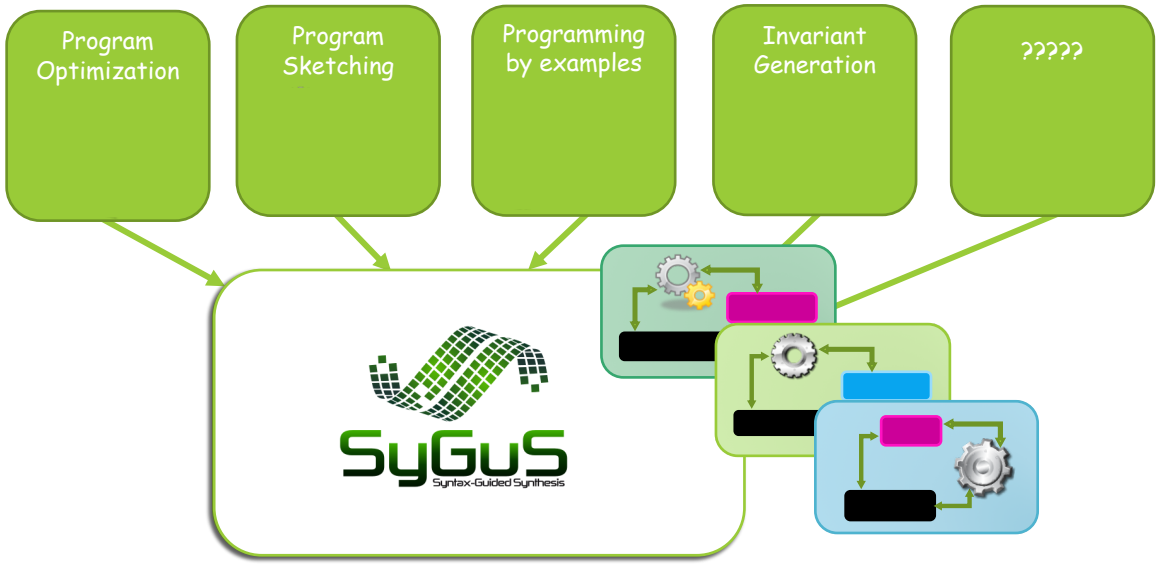



SyGuS

12

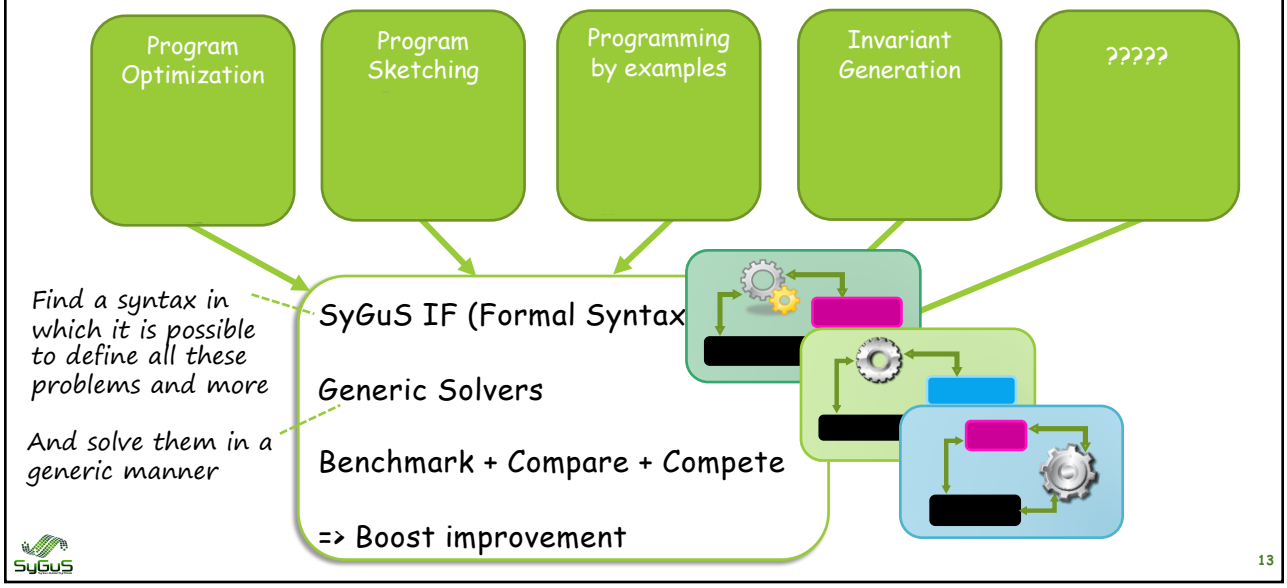
# SyGuS – The Idea [Alur et al.]




SyGuS

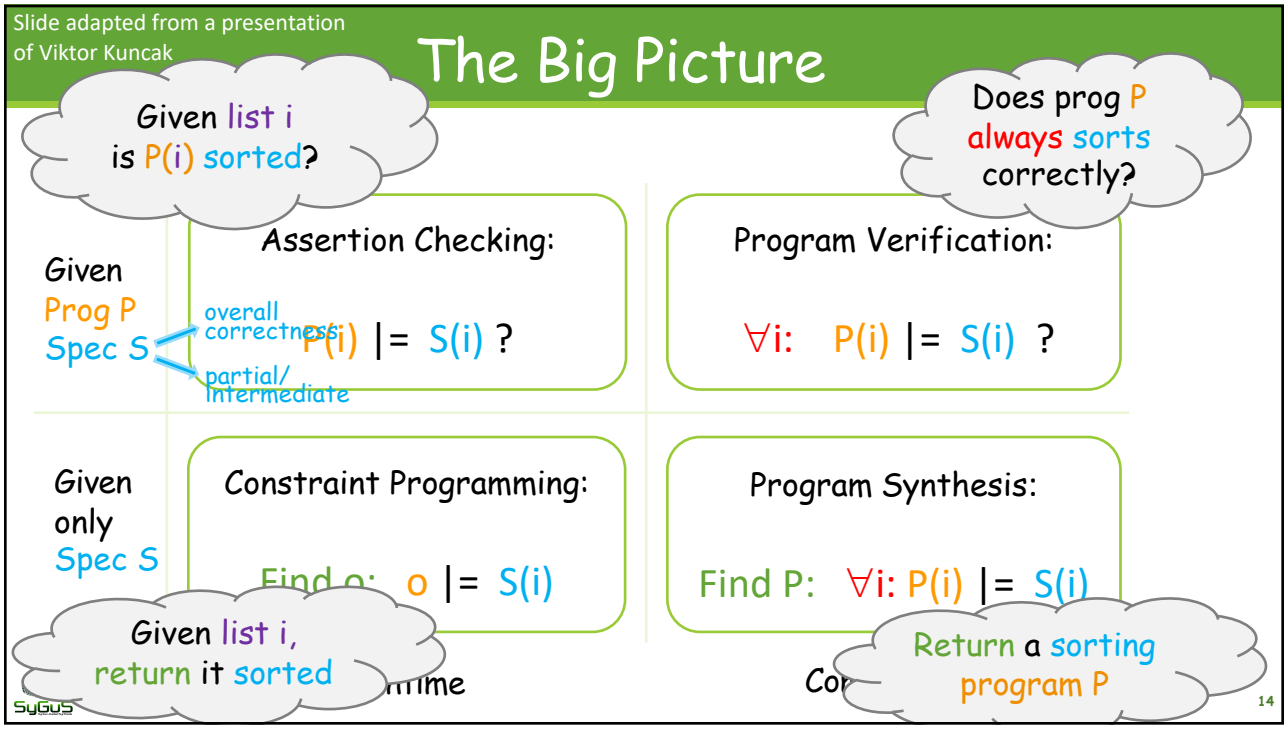
13

# SyGuS – The Idea [Alur et al.]




Slide adapted from a presentation of Viktor Kuncak

# The Big Picture



# The Big Picture

Given Prog P Spec S

**Assertion Checking:** $P(i) \models S(i)$ ?

**Program Verification:** $\forall i: \; P(i) \models S(i)$ ?

Given only Spec S

Return a program P implementing turnoff rightmost 1's

**Program Synthesis:** $\exists P: \; \forall i: P(i) \models S(i)$

Return a program P implementing turnoff rightmost 1's using only so and so operators

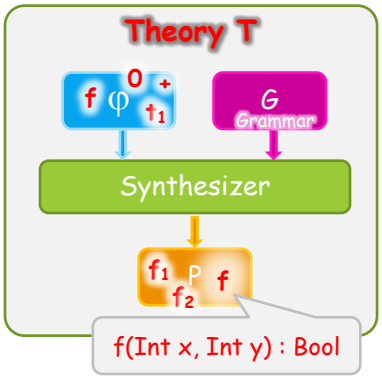
**Syntax-Guided Synthesis:** $\exists P \in [\![R]\!]: \; \forall i: P(i) \models S(i)$

SyGuS

15

# Syntax-Guided Synthesis (SyGuS) Problem

- Fix a background theory T: fixes types and operations
- Function to be synthesized: name f along with its type
  - General case: multiple functions to be synthesized
- Inputs to SyGuS problem:
  - Specification φ
    Typed formula using symbols in T + symbol f
  - Context-free grammar G
    Characterizing the set of allowed expressions $[\![G]\!]$ (in theory T)
- Computational problem:
  Find expression e in $[\![G]\!]$ such that φ[f/e] is valid (in theory T)




SyGuS

17

# SyGuS – formalization example

- Theory QF-LIA
  Types: Integers and Booleans
  Logical connectives, Conditionals, and Linear arithmetic
  Quantifier-free formulas
- Function to be synthesized f (int x, int y) : int
- Specification: $f(x,y) \geq x$ &
  $f(x,y) \geq y$ &
  ( $f(x,y)=x$ | $f(x,y)=y$ )
- Candidate Implementations: Linear expressions
  LinExp := x | y | Const | LinExp + LinExp | LinExp - LinExp
- No solution exists



# SyGuS - formalization example

- Theory QF-LIA
  Types: Integers and Booleans
  Logical connectives, Conditionals, and Linear arithmetic
  Quantifier-free formulas
- Function to be synthesized f (int x, int y) : int
- Specification: $f(x,y) \geq x$ &
  $f(x,y) \geq y$ &
  ( $f(x,y)=x$ | $f(x,y)=y$ )
- Candidate Implementations: Conditional expressions with comparisons

  Term := x | y | Const | If-Then-Else (Cond, Term, Term)
  Cond := Term $\leqslant$ Term | Cond & Cond | ~Cond | (Cond)
- Possible solution:
  If-Then-Else ($x \leq y$, y, x)

# Invariant Synthesis Example – SyGuS IF

A simple program

```
Pre: i >= 0 and j=j0 and i=i0;
while (i > 0) {
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

SyGuS



# Invariant Synthesis Example – SyGuS IF

Function definition

```
(set-logic LIA)

(synth-fun inv-f ((i Int) (j Int) (i0 Int) (j0 Int)) Bool
                ((Start Bool (StartBool))
                 (StartBool Bool (true false
                                  (and StartBool StartBool)
                                  (not StartBool)
                                  (<= StartInt StartInt)))
                 (StartInt Int (i j i0 j0 0 1
                                (+ StartInt StartInt)
                                (- StartInt StartInt)))))
```

SyGuS

# Invariant Synthesis Example – SyGuS IF

Variables declaration

```
(declare-var i0 Int)
(declare-var j0 Int)
(declare-var i Int)
(declare-var j Int)

(declare-var i0! Int)
(declare-var j0! Int)
(declare-var i! Int)
(declare-var j! Int)
```

SyGuS

22

# Invariant Synthesis Example

Program definition

```
(define-fun pre-f ...)

(define-fun trans-f ...)

(define-fun post-f ...)

(constraints ...)
```

SyGuS

23

# Invariant Synthesis Example – SyGuS IF

Constraints

```
(constraint (=> (pre-f i j i0 j0)
                (inv-f i j i0 j0)))

(constraint (=> (and (inv-f i j i0 j0)
                     (trans-f i j i0 j0 i! j! i0! j0!))
                (inv-f i! j! i0! j0!)))

(constraint (=> (inv-f i j i0 j0)
                (post-f i j i0 j0)))

(check-synth)
```

SyGuS



# Invariant Synthesis Example – SyGuS IF

```
Pre: i >= 0 and j=j0 and i=i0;
while (i > 0) {
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

Precondition definition

```
(define-fun pre-f ((i Int) (j Int) (i0 Int) (j0 Int))
                  Bool
                 (and (>= i 0)
                      (and (= i i0) (= j j0))))
```

SyGuS

# Invariant Synthesis Example

```
Pre: i >= 0 and j=j0 and i=i0;
while (i > 0) {
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

Transition definition

```
(define-fun preserve ((x Int) (y Int) (x! Int) (y! Int)) Bool
                     (and (= x! x) (= y! y)))

(define-fun dec-inc ((x Int) (y Int) (x! Int) (y! Int)) Bool
                    (and (= x! (- x 1)) (= y! (+ y 1))))

(define-fun trans-f ((i Int) (j Int) (i0 Int) (j0 Int)
                     (i! Int) (j! Int) (i0! Int) (j0! Int)) Bool
                   (and (preserve i0 j0 i0! j0!)
                        (ite (> i 0)
                             (dec-inc i j i! j!)
                             (preserve i j i! j!))))
```



# Invariant Synthesis Example – SyGuS IF

```
Pre: i >= 0 and j=j0 and i=i0;
while (i > 0) {
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

Post-condition definition

```
(define-fun post-f ((i Int) (j Int) (i0 Int) (j0 Int))
                   Bool
                   (= j (+ j0 i0)))
```

# Optimizing Multiplications – SyGuS IF

## Superoptimizing Compiler

Given a program P, find a "better" equivalent program P'.

```
multiply (x[1,n], y[1,n]) {
   x1 = x[1,n/2];
   x2 = x[n/2+1, n];
   y1 = y[1, n/2];
   y2 = y[n/2+1, n];
   a = x1 * y1;
   b = shift( x1 * y2, n/2);
   c = shift( x2 * y1, n/2);
   d = shift( x2 * y2, n);
   return ( a + b + c + d)
}
```

Replace with equivalent code with only 3 multiplications



# Optimizing Multiplications – SyGuS IF

```
(set-logic BV)

(define-fun multiply ((x (BitVec 32)) (y (BitVec 32))) (BitVec 32) ...)

(synth-fun opt-multiply ((x (BitVec 32)) (y (BitVec 32))) (BitVec 32)

                    ... SYNTACTIC RESTRICTIONS ...

(declare-var x (BitVec 32))
(declare-var y (BitVec 32))

(constraint (= (multiply x y) (opt-multiply x y)))

(check-synth)
```

# Let Expressions

- Synthesized expression maps directly to a straight-line program
- Grammar derivations correspond to expression parse-trees
- Problem:
  How to capture common sub-expressions (which map to aux vars) ?
- Solution:
  Allow "let" expressions
- Candidate-expressions for a function f(int x, int y) : int

T specifies the some of two **identical** sub-expressions

```
T := (let ((z U)) z+z)
U := x | y | Const | U+U | U*U | T
```

SyGuS

30

# Outline

- **SyGuS-The Problem**
  - Motivation for Syntax-Guided Synthesis
  - Recent trends in program synthesis
  - SyGuS within the big picture
  - Formalization of Syntax-Guided Synthesis
- **Solving SyGuS**
  - CEGIS – Counterexample Guided Inductive Synthesis
  - Search by enumeration
  - Symbolic search
  - Stochastic search
  - ICE Learning
  - A glance at competition results

SyGuS

31

# Solving SyGuS

P(i) |= S(i) ? | ∀i: P(i) |= S(i) ?
∃o: o |= S(i) | ∃P: ∀i: P(i) |= S(i)
∃P∈ ⟦G⟧:∀i: P(i)|= S(i)

- Is SyGuS same as solving SMT formulas with quantifier alternation?

$$\exists P \in [\![G]\!]: \forall i: P(i) \models S(i)$$

- SyGuS can sometimes be reduced to Quantified-SMT, but not always
  - Set ⟦G⟧ is all linear expressions over input vars x, y
    SyGuS reduces to ∃a,b,c. ∀x,y. φ [ f / ax+by+c ]
  - Set ⟦G⟧ is all conditional expressions
    SyGuS cannot be reduced to deciding a formula in LIA

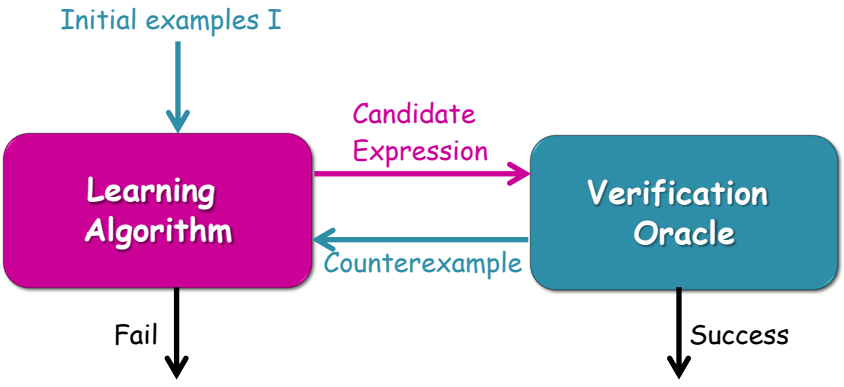
No bound/structure for all possible programs
- ITE(x>=y,x,y)
- y & (ITE (x & !y, y | x, ITE())

- Solving approaches build on
  - structure of the set ⟦G⟧ of candidate implementations
  - solution strategies for Quantified-SMT formulas

SyGuS

32

# SyGuS as Active Learning




Concept class: Set ⟦G⟧ of expressions

Examples: Concrete input values

SyGuS

36

# SyGuS Solutions

- CEGIS - Counter-Example Guided Inductive Synthesis approach (Solar-Lezama, Seshia et al)
- Related work: Similar strategies for solving quantified formulas and invariant generation
- Learning strategies based on:
  - **Enumerative** (search with pruning): Udupa et al (PLDI'13)
  - **Symbolic** (solving constraints): Library components - Gulwani et al (PLDI'11); Sketch-based – Solar-Lezama et al (PhD thesis, UCB, 2008)
  - **Stochastic** (probabilistic walk): Schkufza et al (ASPLOS'13)
  - **Quantified Instantiation** (treat SMT as whitebox): Reynolds et al (CAV'15)
  - **ICE** (learning + implications): Garg et al (CAV'14)



# Running Example

- Specification:

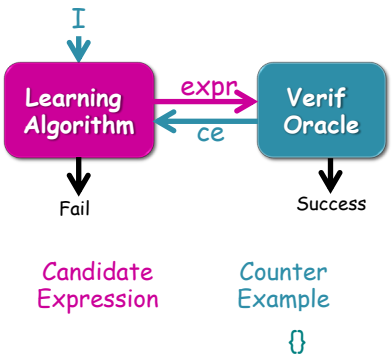
$$(x \le f(x,y)) \;\&$$
$$(y \le f(x,y)) \;\&$$
$$(f(x,y) = x \mid f(x,y) = y)$$

- Syntactic Restrictions:

  all expressions built from x,y,0,1,
  <=, =, =>, +,
  If-Then-Else

# Enumerative

- Find an expression consistent with a given set of concrete examples
- Enumerate expressions in increasing size, and evaluate each expression on all concrete inputs to check consistency
- Key optimization for efficient pruning of search space:
  - Expressions $e_1$ and $e_2$ are equivalent if $e_1(a,b)=e_2(a,b)$ on all concrete values (x=a,y=b) in Examples
  - E.g. If-Then-Else $(0 \le x, e_1, e_2)$ considered equivalent to $e_1$ if in current set of examples x has only non-negative values
  - Only one representative among equivalent sub-expressions needs to be considered for building larger expressions






# Symbolic CEGIS

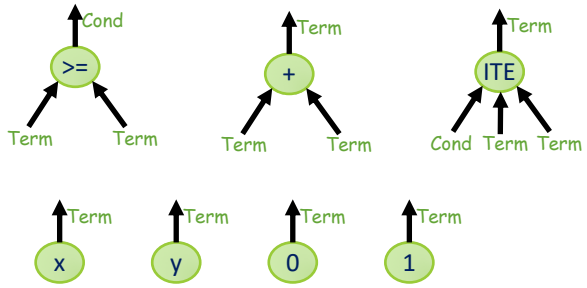
- Recall, in general we cannot use SMT solvers to solve SyGuS

$$\exists P \in [\![ G ]\!] : \forall i: P(i) \models S(i)$$

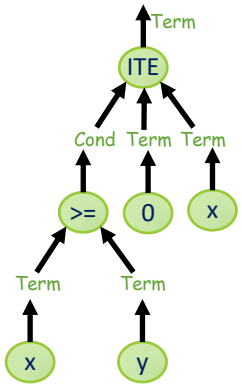
- Idea:
  We can use the constraint solver also in the learning algorithm by restricting the size/depth of the expressions
- We need a way to encode the restricted set of expressions, and if no such expression exists, increase the size
- Two variations:
  - Symbolic [Jha et al.]
  - Sketch-Based [Solar-Lezama et al.]

# Symbolic [Jha et al.]

- Each production in the grammar is thought of as a library component




- A well-typed loop-free program comprising these component corresponds to an expression DAG from the grammar






# Symbolic [Jha et al.]

- Start with a library consisting of some number of occurrences of each component.

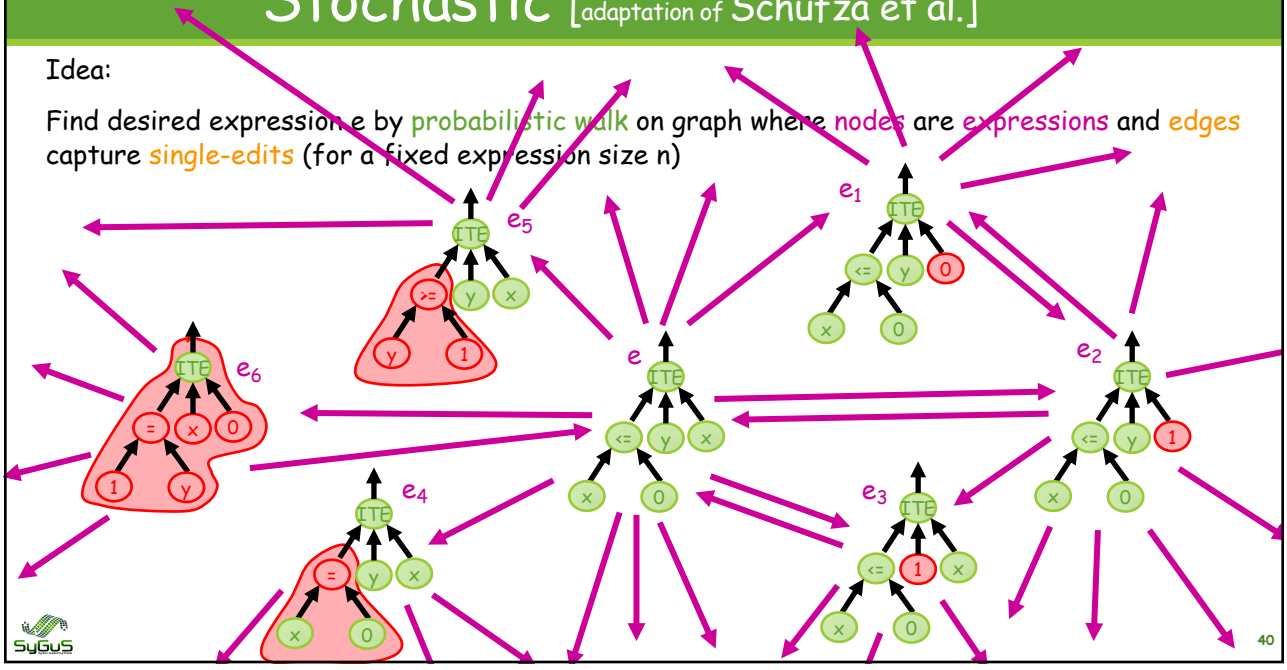
| x | y | 0 | 1 | + | ITE | >= | = |
|---|---|---|---|---|---|---|---|
| 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 |

- Synthesis Constraints:
  - Program composed of library's component, Shape is a DAG, Types are consistent
  - Spec $\varphi[f/e]$ is satisfied on every concrete input values in Examples
- Use an SMT solver to find a satisfying solution.
  - If synthesis fails, try increasing the number of occurrences of components in the library in an outer loop.

# Stochastic [adaptation of Schufza et al.]

Idea:

Find desired expression e by probabilistic walk on graph where nodes are expressions and edges capture single-edits (for a fixed expression size n)




SyGuS

40

# Stochastic [Metropolis-Hastings Algorithm]

- Because the graph is strongly connected, we can reach each node with some probability
- Let Score(e) be the "Extent to which e meets the spec φ"
  Having P(e) ∝ Score(e) we increase the chances of getting to expressions with better score.
  To escape "local minima" we allow with some probability moving to expressions with lower score.
- Specific choice of score:
  For a given set I of concrete inputs, Score(e) = $\exp(-\frac{1}{2}\text{Wrong}(e))$
  where Wrong(e) = No of examples in I for which ~ φ [f/e]
- Score(e) is large when Wrong(e) is small
  => Expressions e with Wrong(e) = 0 more likely to be chosen in the limit than any other expr

SyGuS

41

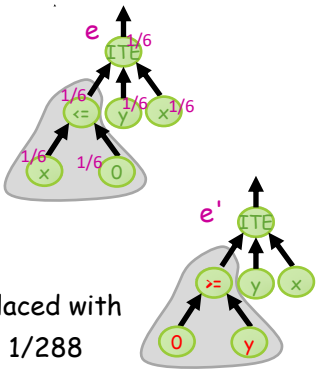
# Stochastic

- Initial candidate expression e sampled uniformly from $E_n$
- When Score(e) = 1, return e
- Pick node v in parse tree of e uniformly at random.
  Replace subtree rooted at v with subtree of same size, sampled uniformly
- With probability min{ 1, Score(e')/Score(e) }, replace e with e'
- Outer loop responsible for updating expression size n



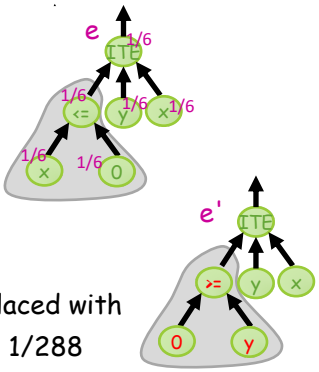
# Stochastic - Example

- Expr size = 6
- All expressions of this size have the following form
- One chosen at random, with probability 1/(4×4×4×4×3) = 1/768
- Suppose it is e = ITE(x <= 0, y, x)
- Next a sub-expression (node) to mutate is chosen at random
- Probability to mutate node 2 is 1/6
- There are 1/(4×4×3) = 1/48 conditional expression node 2 can be replaced with
- Choice e' = ITE(0 <= y, y, x) is considered with probability 1/(48×6) = 1/288
- If concrete examples are (-1 , 4) (-3 , -1) (-1 , -2) (1 , 2) (3, 1) (6, 2)
- With p



# Stochastic - Example

- Expr size = 6
- All expressions of this size have the following form
- One chosen at random, with probability $1/(4\times4\times4\times4\times3) = 1/768$
- Suppose it is e = ITE(x <= 0, y, x)
- Next a sub-expression (node) to mutate is chosen at random
- Probability to mutate node 2 is 1/6
- There are $1/(4\times4\times3) = 1/48$ conditional expression node 2 can be replaced with
- Choice e' = ITE(0 <= y, y, x) is considered with probability $1/(48\times6) = 1/288$
- If concrete examples are




| | (-1 , 4) | (-3 , -1) | (-1 , -2) | (1 , 2) | (3, 1) | (6, 2) |
|---|---|---|---|---|---|---|
| e = ITE(x <= 0, y, x) | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| e' = ITE(0 <= y, y, x) | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |

- With probability min{ 1, $exp^{-3/2}/exp^{-1}$ }, replace e with e'

SyGuS



# ICE - Invariant Synthesis [Garg. et al.]

- ICE is a solver only for invariant synthesis problems
- An acronym for Implications, Counterexamples & Examples
- Method adapting traditional learning techniques to learning invariants

SyGuS

# Machine Learning

- Traditional Algorithms:
  - Task that we need the program to do is clear, precise.
  - We build algorithms that perform the task, efficiently.
- Machine Learning Algorithms:
  - Algorithms that automatically improve on (possibly ill-defined) tasks through experience.
  - Improves with experience at some task T
    - with respect to some performance measure P
    - based on experience E

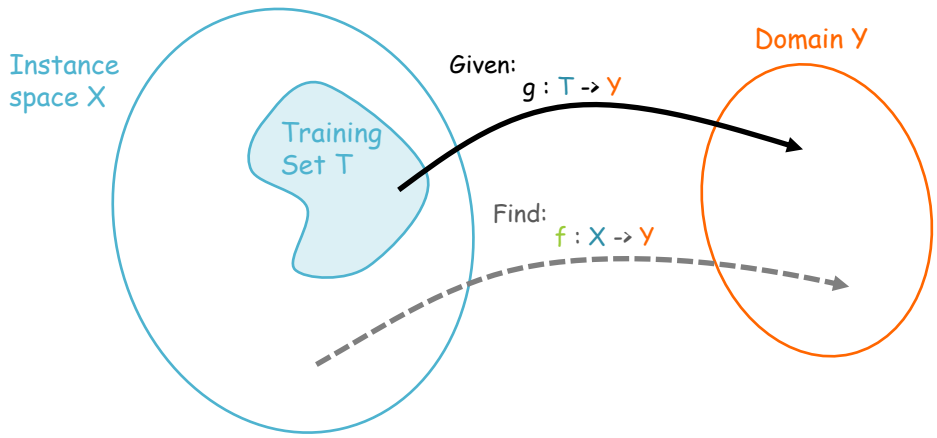
PAC etc.

- Examples:
  - Spam filters, Recognizing faces in a crowd, Credit fraud, Netflix recommendations

SyGuS

56

# Machine Learning

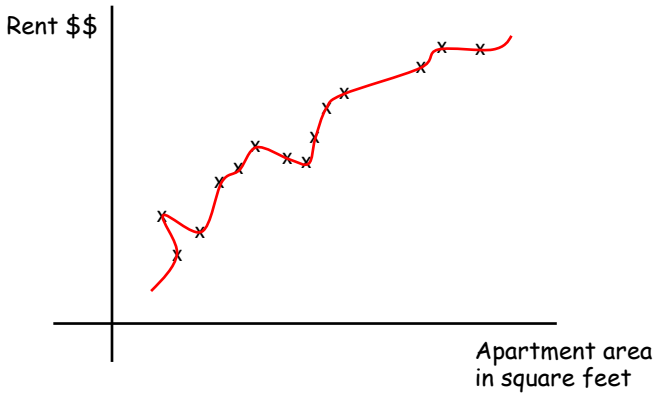



f needs to predict the result for unseen instances in X\T.
f needs to generalize g.

SyGuS

57

# Machine Learning

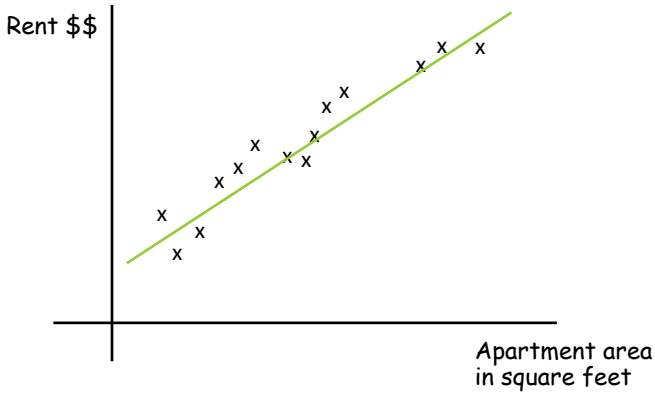



An f that maps T correctly but maps other points to 0 overfits!

SyGuS

58

# Machine Learning




Concept class:
A definition of the set of possible functions the learning algorithm can output e.g. only linear functions.

The concept class states what type of functions to generalize to, or with what type of functions we can approximate?

An f that maps T correctly but maps other points to 0 overfits!
In order to generalize, f may not even map T correctly.

SyGuS

59

## Overfitting and Generalizing

- Consider a sample { x1 -> r2, x2 -> r2, …, xn -> rn }
  providing valuations from $\{0,1\}^n$ to {0,1}
- Let hypotheses space be all Boolean functions.
  Then the function f that says
  if (x=x1) return r1
  else if (x=x2) return r2
  else if (x=x3) return r3
  . . . else return True;
  overfits the sample.
- How do we avoid overfitting?
  - Choose a more constrained hypothesis space (e.g., conjuncts)
  - Occam's razor: learner must find a "small" function.



## Overfitting and Generalizing

- Consider a sample { x1 -> r2, x2 -> r2, …, xn -> rn }
  providing valuations from $\{0,1\}^n$ to {0,1}
- Possible choices for the concept class:
  - Conjuncts (Elimination Algorithm)
  - Linear threshold functions (Winnow)
  - Arbitrary Boolean formulas (Decision Tree Learning)
- Every choice of concept class introduces a bias:
  - Conjuncts – bias towards large conjuncts
  - Linear threshold functions – margin bias
  - Arbitrary Boolean formulas – bias towards small trees

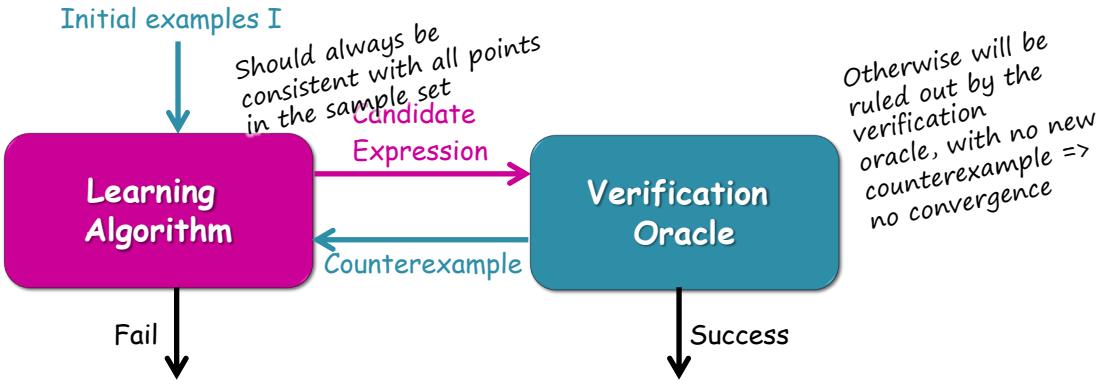
# Adapting ML for invariant synthesis

Issue #1: The function needs to be correct on all instances!



# Adapting ML for invariant synthesis

Issue #1: The function needs to be correct on all instances!



# Adapting ML for invariant synthesis

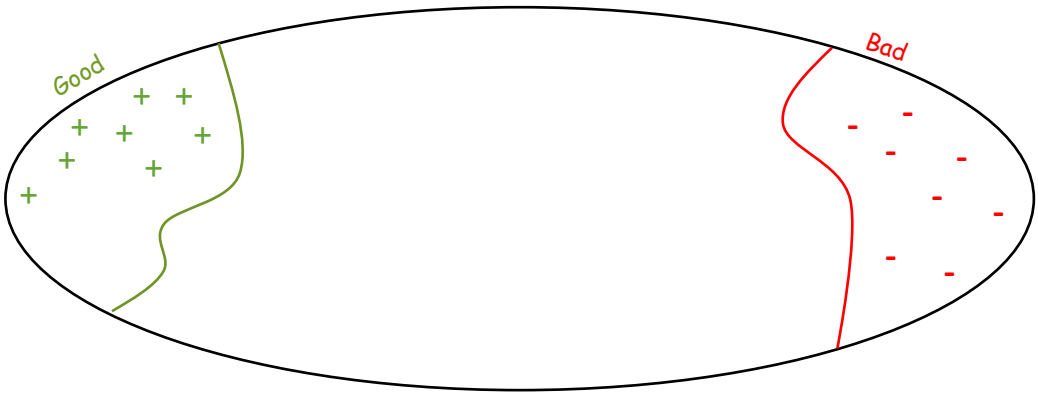
Issue #2:

- The oracle does not know the invariant
- It only knows the premises it should satisfy
  - Includes init
  - Excludes bad
  - Is inductive
- Using only negative/positive examples does not suffice

SyGuS

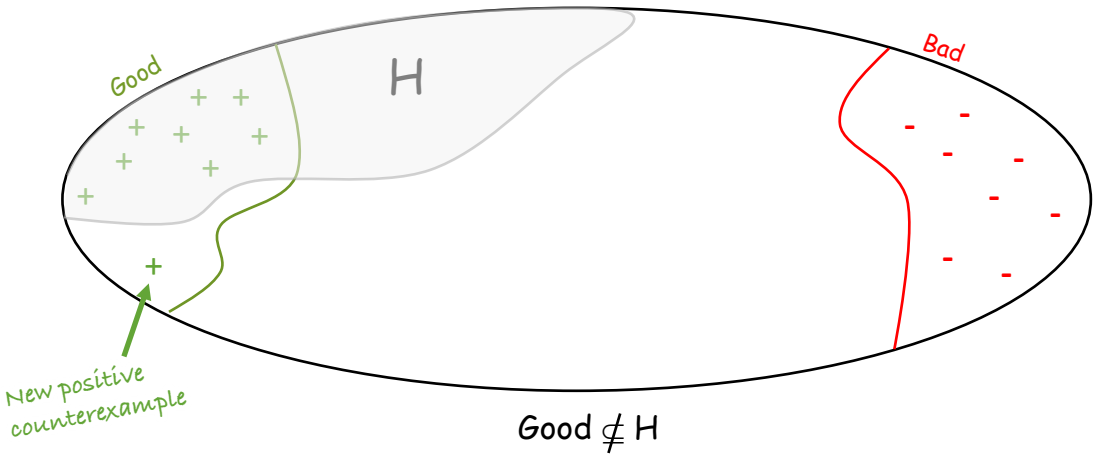


# Positive counterexample




SyGuS

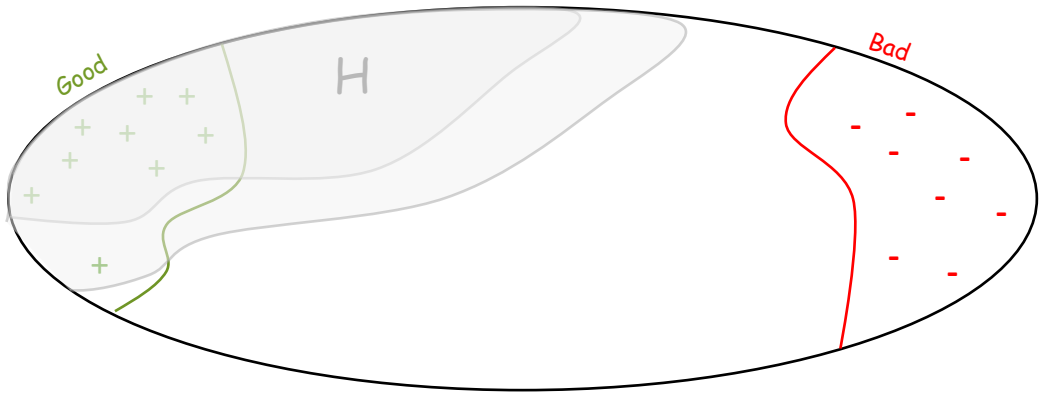
# Positive counterexample




SyGuS

66

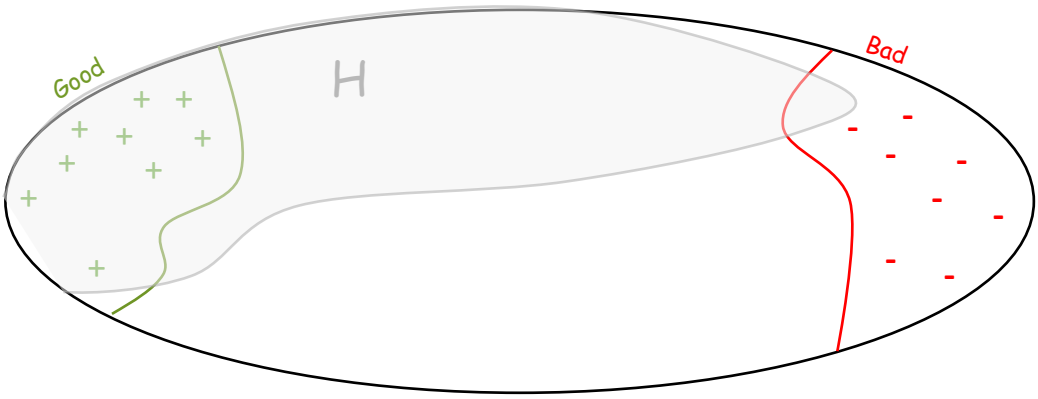
# Positive counterexample




SyGuS

67

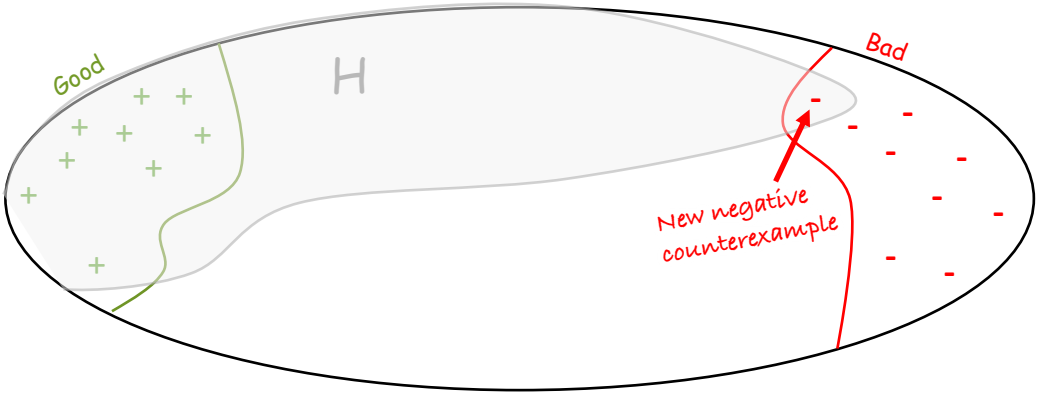
# Negative counterexample




SyGuS

68

# Negative counterexample

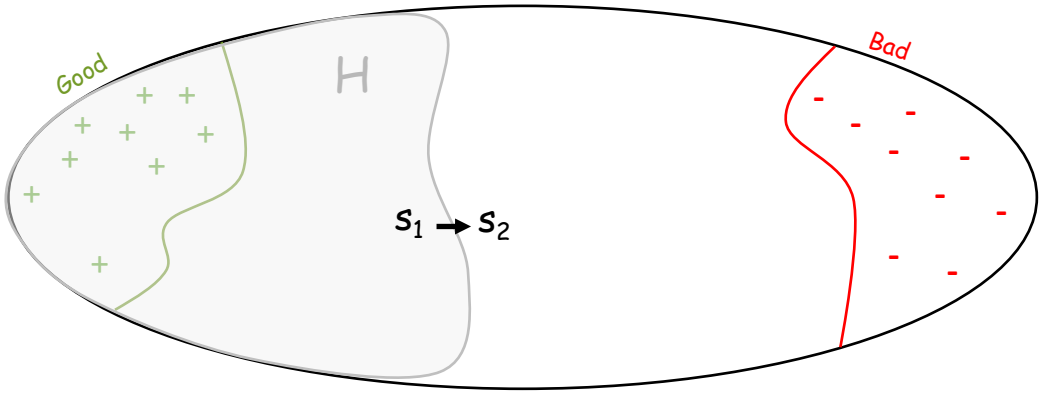



$H \cap Bad \neq \emptyset$

SyGuS

69

# ?? counterexample

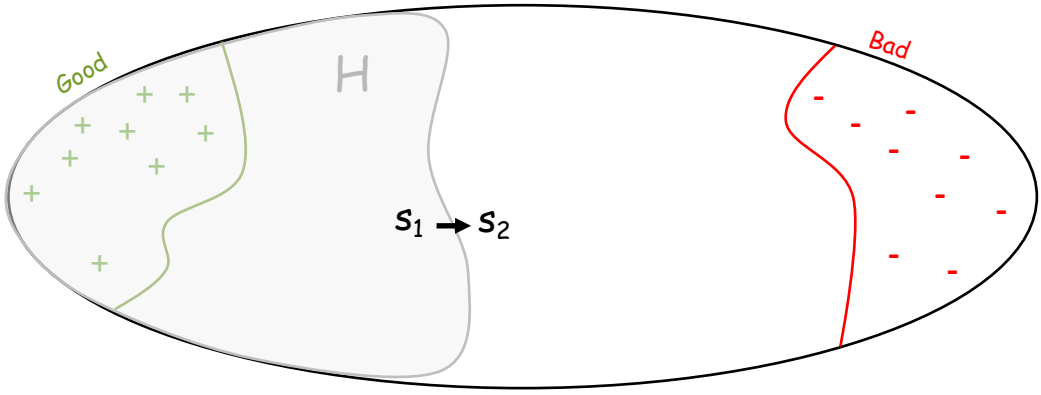



$s_1 \in H$ and $s_2 \notin H$
We don't know whether these should be included in the invariant or not.
All we know is that if $s_1$ is included so is $s_2$.

SyGuS



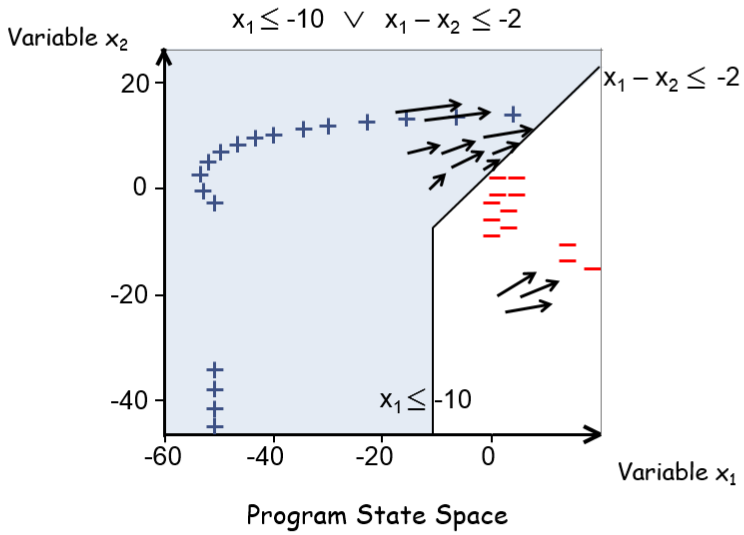
# Implication counterexample




$s_1 \in H$ and $s_2 \in H$
We don't know whether these should be included in the invariant or not.
All we know is that if $s_1$ is included so is $s_2$.

SyGuS

# Learning an invariant given an ICE sample




SyGuS

72

# Other Methods for Solving SyGuS

- Quantified Instantiation
- Unification
- …

SyGuS

# SyGuS Competition

- Six competitions were held annually: 2014, 2015, 2016, 2017, 2018, 2019
- Competition was conducted **offline**, using **StarExec.**
- First competition had a **Single track**, Second competition had **3 tracks**, third **4 tracks**, fourth and onward **5 tracks:**
  1. General,
  2. Conditional Linear Integer Arithmetic (CLIA)
  3. Invariant Generation (INV, subset of #2 for invariant synthesis )
  4. Programming by examples (PBE-BV)
  5. Programming by examples (PBE-Strings)



# SyGuS Competition

- **Benchmarks**:
  - Bit-manipulation programs from Hacker's delight
  - Integer arithmetic: Find max, search in sorted array
  - Challenge problems such as computing Morton's number
  - Invariant generation problem
  - Compiler Optimizations
  - Motion Planning
  - Let Benchmarks
  - Multiple Functions
  - ICFP
  - LIA
  - ….

# SyGuS for cyber security [Eldib, Wu, Wang 2016]

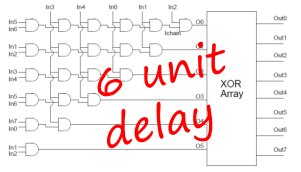



Fig. 1. PPRM1 AES S-box that is vulnerable to FSA.

***Original circuit***
*[Morioka & Satoh, CHES 2002]*

***SyGuS-Synthesized countermeasure***
*[Eldib, Wu, Wang CAV2016]*

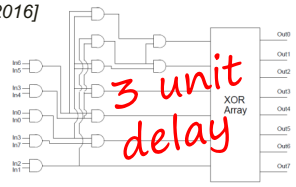



Fig. 3. S-box with our new countermeasure.

***Buffered circuit***
*[Ghalaty et al, DATE 2014]*
*[Endo et al, IEEE TVLSI, 2014]*

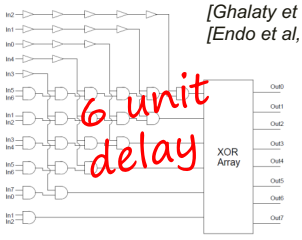



Fig. 2. S-box with buffered countermeasure [22].

# Timeline View – Tracks

| 1st Comp. | 2nd Comp. | 3rd Comp. | 4th Comp | 5th Comp |
|---|---|---|---|---|
| General | General | General | General | General |
| | CLIA | CLIA | CLIA | CLIA |
| | Inv | Inv | Inv | Inv |
| | | PBE Bitvectors | PBE Bitvectors | PBE Bitvectors |
| | | PBE Strings | PBE Strings | PBE Strings |

# Timeline View – Solvers

| 1st Comp. | 2nd Comp. | 3rd Comp. | 4th Comp | 5th Comp |
|---|---|---|---|---|
| Enumerative | Enumerative, | Enumerative | CVC4$_{2017}$ | CVC4$_{2018}$ |
| Symbolic, | SosyToast | Sketch$_{AC}$ | EUSolver$_{2017}$ | DryadSynth$_{2018}$ |
| Sketch | Sketch AC | Stochastic | Euphony | LoopInvGen$_{2018}$ |
| Stochastic | Stochastic | Alchemist$_{CS,CSDT}$ (2) | DryadSynth | Hondini |
| Alchemist | Alchemist$_{CS,CSDT}$ (2) | ICE | LoopInvGen | EUSolver$_{2017}$ |
| | ICE | CVC4 1.5.1 | E3Solver | |
| | CVC4 1.5 | EUSolver | | |

# Timeline View – Successes & Challenges

| | 1st Comp. | 2nd Comp. | 3rd Comp. | 4th Comp. | 5th Comp. |
|---|---|---|---|---|---|
| Successes: | Hackers' Delight Invariant Gen | Arrays Max n <= 15 | More benchs ICFP Faster | PBE-Stings Cryp. Circ. Prog. Rep. | More bench across most cat. Expr Size Improv. |
| Challenges: | Arrays Max n>4 Let MultFunc ICFP | Let MultFunc ICFP CompOpt | Let MultFunc CompOpt PBE-Strings Expr Sizes | Let MultFunc CompOpt Inst. Selec. More Cryp. Circ. | Let MultFunc CompOpt Inst. SelecMore More Cryp. Circ. |

## Discussion

# Comments or questions?

Acknowledgments:
- ICE slides – adapted from a presentation of Garg & Madhusudan.


## Syntax-Guided Synthesis


www.sygus.org