Guiding Enumerative Function Synthesis with Machine Learning

Julian Parsert June 15, 2024

Syntax-Guided Synthesis (SyGuS)

SyGuS is a problem of synthesising

- a function F within
- a theory τ that satisfies
- a semantic specification ϕ
- with a syntactic restriction G.

$\rightarrow$ SyGuS-IF closely follows SMTLIB

SyGuS Tools/competitions

$$\exists f.\ \forall xy.f(x,y) \geq x \wedge f(x,y) \geq y \wedge (f(x,y) = x \vee f(x,y) = y)$$
$$f \in G$$

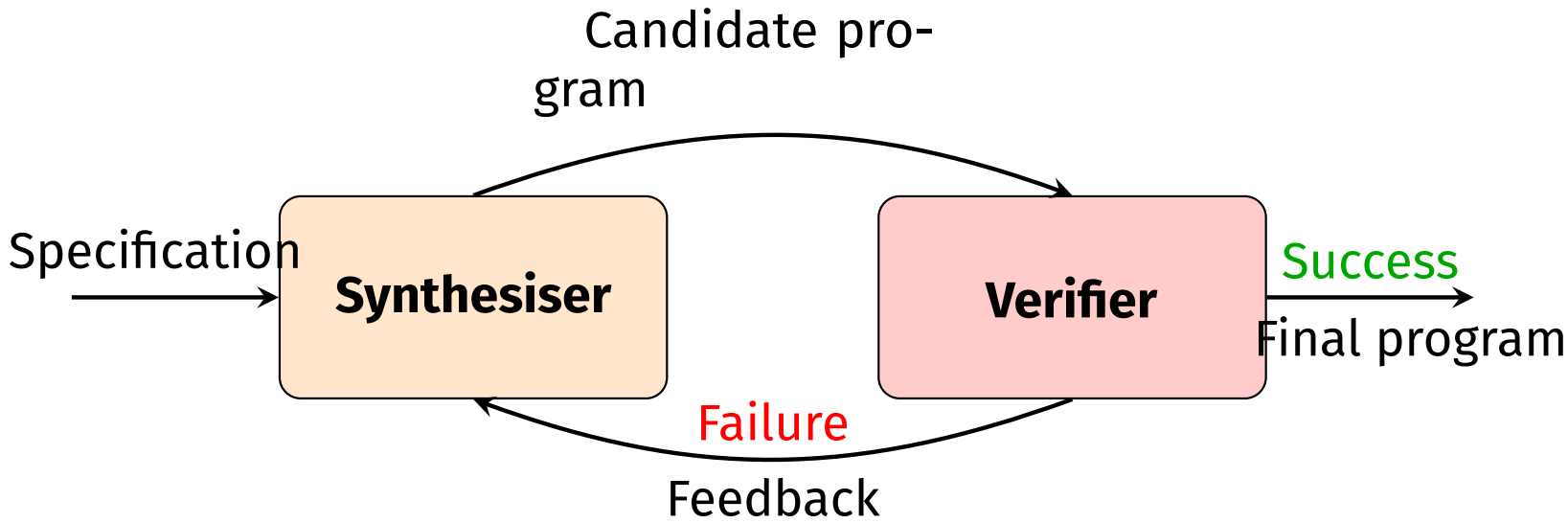
```
1  (set-logic LIA)
2  (synth-fun max2 ((x Int) (y Int)) Int
3      ((I Int) (B Bool))
4      ((I Int (x y 0 1 (+ I I) (- I I) (ite B I I)))
5       (B Bool ((and B B) (or B B) (not B)
6                  (= I I) (<= I I) (>= I I)))))
7  (declare-var x Int)
8  (declare-var y Int)
9  (constraint (>= (max2 x y) x))
10 (constraint (>= (max2 x y) y))
11 (constraint (or (= x (max2 x y)) (= y (max2 x y))))
12 (check-synth)
```

```
1  (define-fun max2 ((x Int) (y Int)) Int   (ite (>= x z) x z))
```

Enumerative Synthesis

In Syntax-Guided Synthesis:
Use grammar to systematically enumerate space of possible programs

Counter-Example Guided Inductive Synthesis (CEGIS)



AI in SyGuS

Many AI based approaches to Synthesis (and SyGuS):

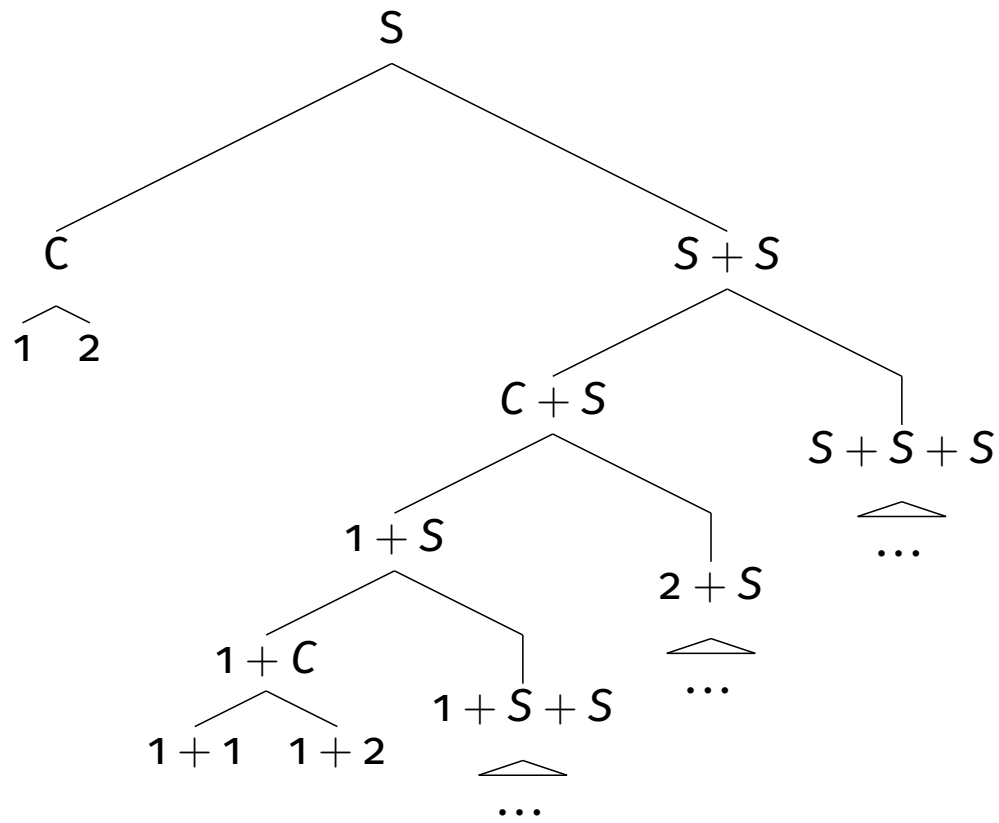
- DeepCoder (Deep Learning for I/O examples)
- Neuro-Symbolic Program Synthesis (Neural Embedding of I/O examples, R3NN syntheses a function from Embedding)
- DreamCoder
- Flash Fill
- ...

$\rightarrow$ All based on I/O or PBE domains

Abstract Domains/Theories with logical specifications?

Enumerative Search as a Tree Search

$$S \to S + S \mid C$$
$$C \to 1 \mid 2$$



Reinforcement Learning and Data-Generation for Syntax-Guided Synthesis

with Elizabeth Polgreen, AAAI 2024

- AlphaZero-style Monte-Carlo Tree Search
- train Value/Policy models to “evaluate” intermediate nodes
- Value/Policy models trained in reinforcement learning

Guiding Enumerative Program Synthesis with LLMs

Yixuan Li, Julian Parsert, and Elizabeth Polgreen, CAV 2024

Synthesis with LLMs: Idea

Let's ask ~~blockchains~~ LLMs to solve ~~everything~~ synthesis.

Synthesis with LLMs: Idea

Let's ask ~~blockchains~~ LLMs to solve ~~everything~~ synthesis.

If the LLM fails, try again ($\times n$ where $n = 6$, trust me)

Synthesis with LLMs: Idea

Let's ask ~~blockchains~~ LLMs to solve ~~everything~~ synthesis.

If the LLM fails, try again ($\times n$ where $n = 6$, trust me)

If it still fails?
Hypothesis:
Correct solutions "syntactically close" to LLM suggestions

Synthesis with LLMs: Idea

Let's ask ~~blockchains~~ LLMs to solve ~~everything~~ synthesis.

If the LLM fails, try again ($\times n$ where $n = 6$, trust me)

If it still fails?
Hypothesis:
Correct solutions "syntactically close" to LLM suggestions

Technique

- Use symbolic enumerator,
- to narrow synthesis search space
- by prioritising functions "near" LLM suggestions.

Rule Weights, probabilistic CFG

Before diving into the specifics of our approach...

- a function F within
- a theory τ that satisfies
- a semantic specification ϕ
- with a syntactic restriction G.

Rule Weights

LLM-Generated program for the SyGuS example

```
(define-fun Prog((x1 Int) (x2 Int) (x3 Int)) Int
    (ite (>= x1 x2) (ite (>= x1 x3) x1 x3) (ite (>= x2 x3) x2 x3)))
```

Rules appeared are:

$$\texttt{Start} \longrightarrow \texttt{(ite StartBool Start Start)}$$

$$\texttt{Start} \longrightarrow \texttt{x1}$$

$$\texttt{Start} \longrightarrow \texttt{x2}$$

$$\texttt{Start} \longrightarrow \texttt{x3}$$

$$\texttt{StartBool} \longrightarrow \texttt{(>= Start Start)}$$

Rule Weights

LLM-Generated program for the SyGuS example

```
(define-fun Prog((x1 Int) (x2 Int) (x3 Int)) Int
    (ite (>= x1 x2) (ite (>= x1 x3) x1 x3) (ite (>= x2 x3) x2 x3)))
```

We calculate a weight for each rule as the number of times that rule appears in the programs.

$$\mathit{weight}[\texttt{Start} \rightarrow \texttt{(ite StartBool Start Start)}] = 3$$
$$\mathit{weight}[\texttt{Start} \rightarrow \texttt{x1}] = 3$$
$$\mathit{weight}[\texttt{Start} \rightarrow \texttt{x2}] = 3$$
$$\mathit{weight}[\texttt{Start} \rightarrow \texttt{x3}] = 4$$
$$\mathit{weight}[\texttt{StartBool} \rightarrow \texttt{(>= Start Start)}] = 3$$

pCFG

A context-free grammar with probabilities attached to the rules.

The probability associated with a rule $\alpha \to \beta$

$$[\texttt{Start} \to \texttt{(ite StartBool Start Start)}] \mapsto 3/19$$
$$[\texttt{Start} \to \texttt{x1}] \mapsto 3/19$$
$$[\texttt{Start} \to \texttt{x2}] \mapsto 3/19$$
$$[\texttt{Start} \to \texttt{x3}] \mapsto 4/19$$
$$[\texttt{StartBool} \to \texttt{(>= Start Start)}] \mapsto 3/19$$

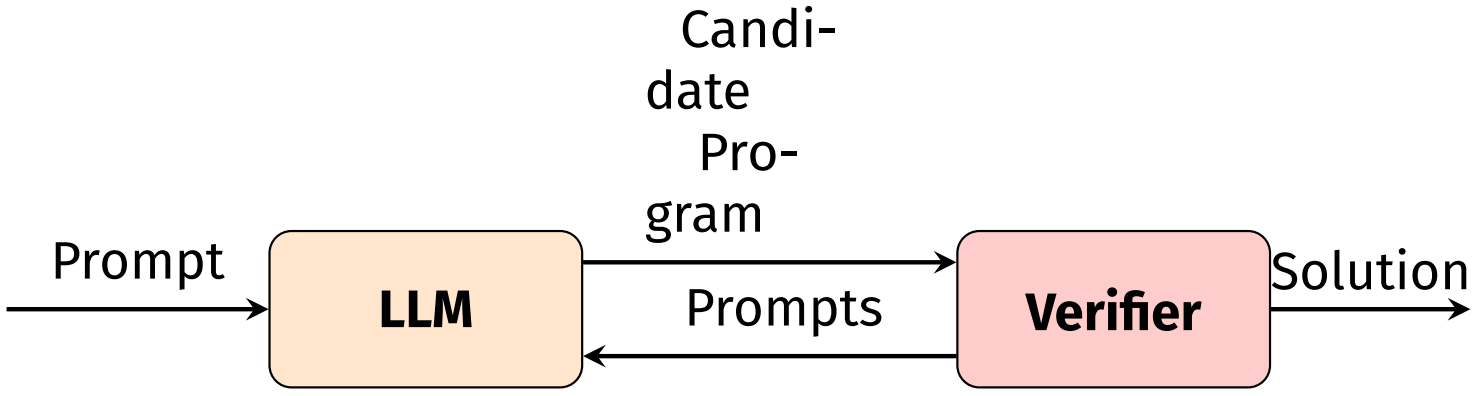
3 Synthesis Methods with LLMs

- Stand-alone LLM
- pCFG-synth (pCFG = probabilistic context-free grammar)
- iLLM-synth (iLLM = integrated Large Language Model)

Stand-alone LLM

- Tailored prompts assess LLM's formal synthesis skills.
- Inspired by Chain-of-Thought: Prompt LLM, verify, re-prompt if failed.

Up to 6 synthesis attempts per benchmark



A SyGuS Example: the `max3` Problem

A SyGuS specification that asks for a program synthesizing the maximum of 3 inputs.

```
(set-logic LIA)
(synth-fun Prog ((x1 Int) (x2 Int) (x3 Int)) Int
    ((Start Int) (StartBool Bool) (Const Int)) (
    (Start Int(
        Const (- Start) (+ Start Start) (- Start Start)
        (* Start Const) (ite StartBool Start Start) ...
        x1 x2 x3))
    (StartBool Bool(
        (> Start Start) (= Start Start) (>= Start Start)
        (and StartBool StartBool) (or StartBool StartBool)
        (not StartBool) true ...))
    (Const Int (0 1))))
(declare-var x1 Int)
(declare-var x2 Int)
(declare-var x3 Int)
(constraint (>= (Prog x1 x2 x3) x1))
(constraint (>= (Prog x1 x2 x3) x2))
(constraint (>= (Prog x1 x2 x3) x3))
(constraint (or (= x1 (Prog x1 x2 x3))
                (or (= x2 (Prog x1 x2 x3)) (= x3 (Prog x1 x2 x3)))))
(check-synth)
```

How to Prompt the LLM

Prompt for SyGuS example

```
(set-logic LIA)
(declare-var x1 Int)
(declare-var x2 Int)
(declare-var x3 Int)
(constraint (>= (Prog x1 x2 x3) x1))
(constraint (>= (Prog x1 x2 x3) x2))
(constraint (>= (Prog x1 x2 x3) x3))
(constraint (or (= x1 (Prog x1 x2 x3)) (or (= x2 (Prog x1 x2 x3)) (= x3 (Prog x1 x2 x3)))))
(check-synth)
You are a good synthesizer. Do you know what "(define-fun Prog ((x1 Int) (x2 Int) (x3 Int)) Int" is doing?
Write only one Lisp-like method "defun Prog" without any built-in methods or arrays.
Requirements:
1. No built-in functions.
2. Never violate the SMT-LIB constraints above.
3. Pay attention to the define functions.
4. Ensure the response contains one and only one function.
5. Do not include any iterations, BitVec, or Int notations in the function body.
Write it correctly, or I will lose my job and 100 grandmothers will die. Don’t disappoint me.
Write only one Lisp-like method "defun Prog" that never violates the SMT-LIB constraints above.
```

How to Prompt the LLM

Response from LLM

```
(defun Prog (x1 x2 x3)
    (ite (>= x1 x2) (ite (>= x1 x3) x1 x3) (ite (>= x2 x3) x2 x3)))
```

Request for converting Lisp to SMT-LIB code for the last response

```
You are a good programming language converter. Convert the Lisp function to SMT-LIB:
Based on the Lisp code provided above, convert the 'defun' Lisp-like code to a corresponding SMT-LIB function. Use SMT-
       LIB syntax starting with (define-fun
Follow these guidelines:
1. Only give me the function definition starting with '(define-fun'.
2. Pay attention to types. If there are bit-vector terms, they need to be of the same width.
3. Ensure the SMT-LIB function contains one and only one function definition starting with '(define-fun'.
4. Do not include any iterations, BitVec, or Int notations in the function body.
5. Use the assigned values from the Lisp code during translation.
6. Do not introduce any variables that do not exist in the Lisp function.
Rules for SMT-LIB: +, -, *, ite, >, =, <, >=, <=, and, or, not, true, false.
```

How to Prompt the LLM

LLM-Generated program (Stand-alone LLM is done)

```
(define-fun Prog ((x1 Int) (x2 Int) (x3 Int)) Int
    (ite (>= x1 x2) (ite (>= x1 x3) x1 x3) (ite (>= x2 x3) x2 x3)))
```

Prompt requesting a revised solution

```
You are close to the right answer. Take another guess. You have to try something different, think harder. Write a
       different Lisp method that never violates the SMT-LIB constraints above again.
```

Stand-alone LLM

OpenAI's GPT-3.5

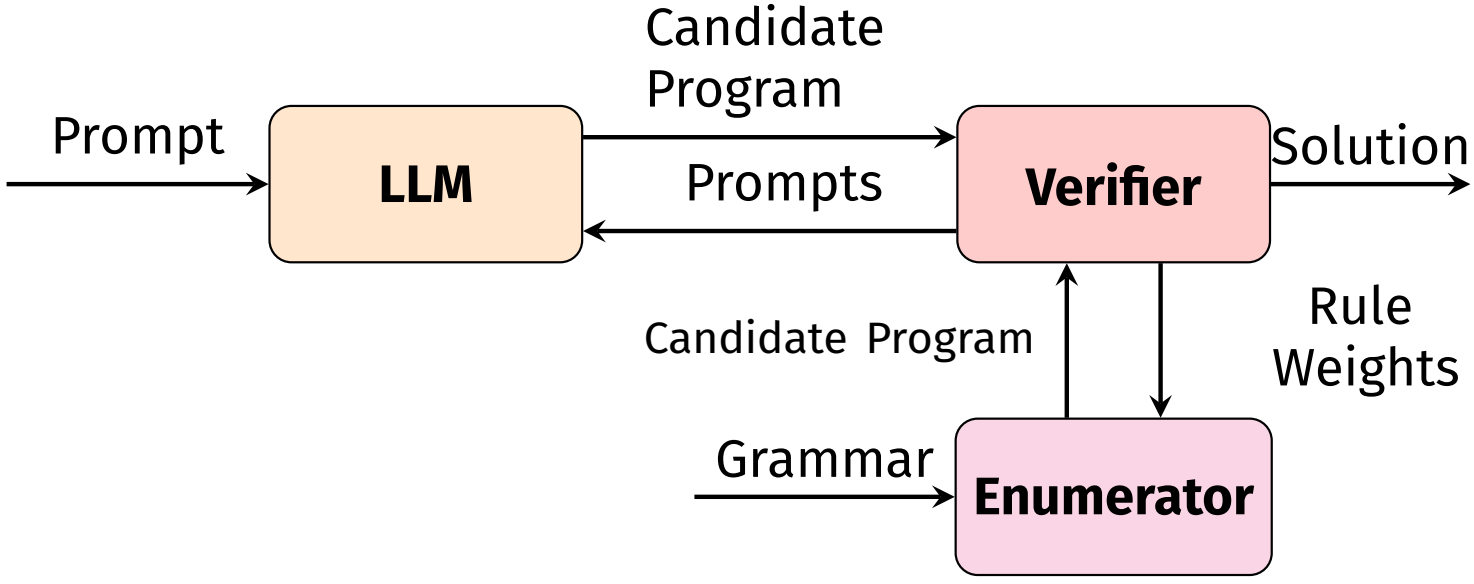
- Solves 49% of benchmarks.
- 4 attempts on average for a correct solution.
- Average generation time: 5 seconds.

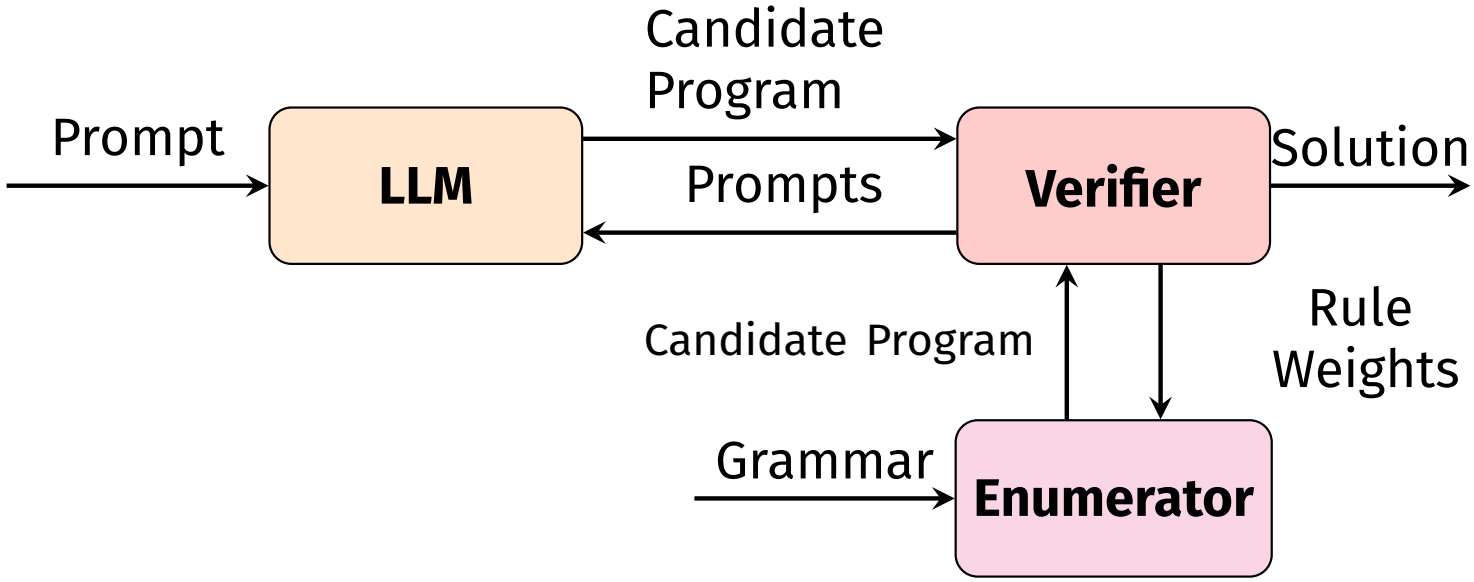
What to do after ~~3~~ ~~4~~ 6 LLM attempts, are we lost?

Let's apply hypothesis:
Correct solutions "syntactically close" to LLM suggestions:

pCFG-synth

- prompt LLM for solutions to the benchmark (as before)
- generate pCFG from LLM candidates.
- use enumerative synthesizer to enumerate according to pCFG





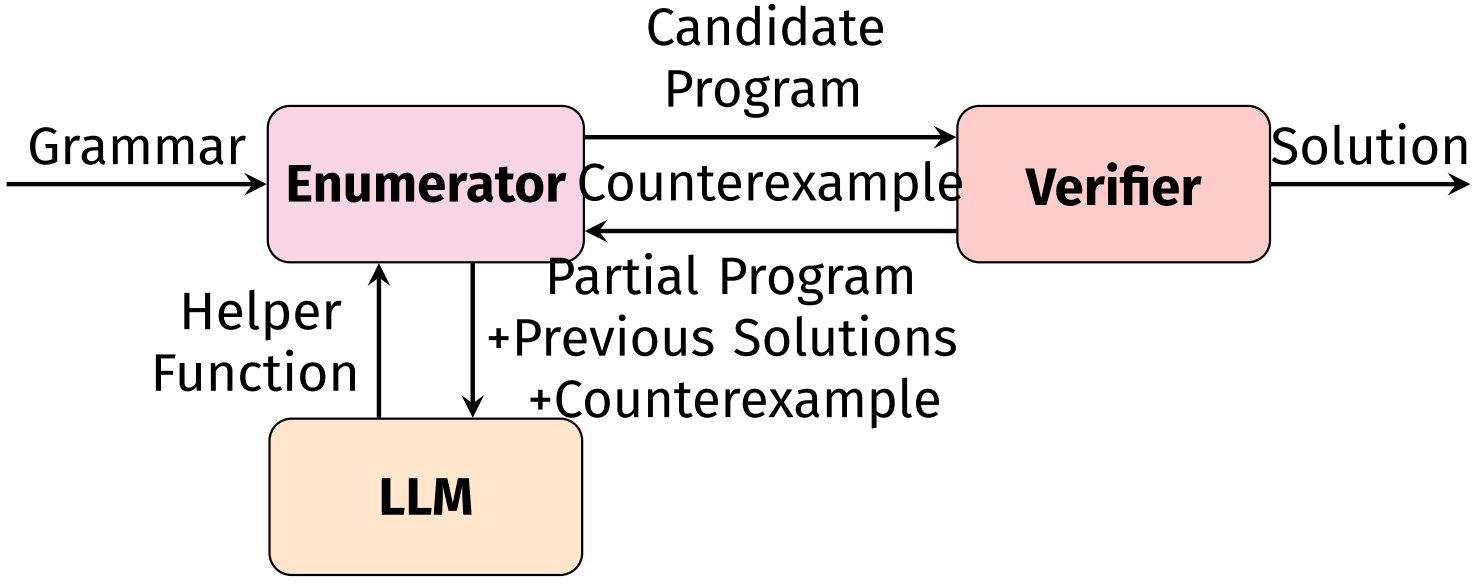
But we gain information during enumeration phase

- Partial Programs
- Previous candiates
- CounterExamples (CEGIS)
- …

Let’s use this information

iLLM-synth

- integrates LLM prompts for dynamic information use.
- LLM suggests helper functions for partial programs.
- Uses LLM responses to expand grammar's production rules and update rule weights.



How does iLLM-synth Work

Prompt for SyGuS example

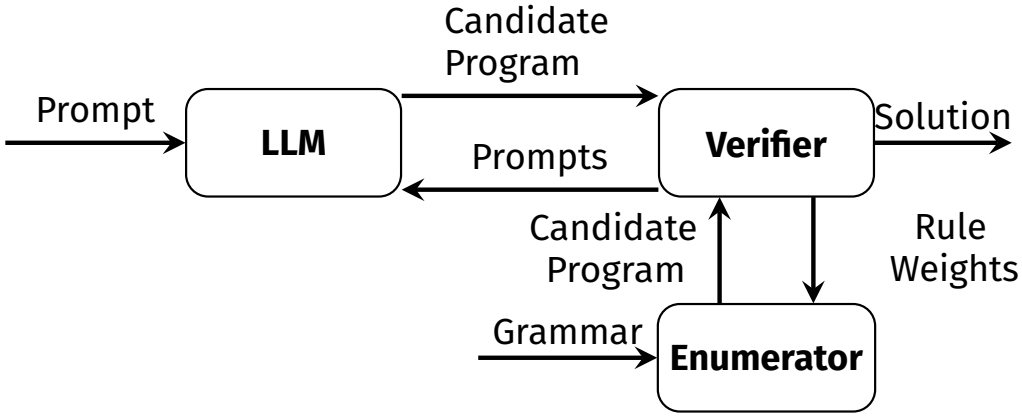
```
You are teaching a student to write SMT-LIB. The student must write a function that satisfies the following constraints:
(constraint ...
...
So far, the student has written this code:
(define-fun Prog ((x1 Int) (x2 Int) (x3 Int)) Int
    (ite ?? ?? ??)
Can you suggest some helper functions for the student to use to complete this code and replace the ??
You must print only the code and nothing else.
```

```
You are teaching a student to write SMT-LIB. The student may find the following functions useful:
(define-fun Prog ...
...
The student must write a function that satisfies the following constraints:
(constraint ...
...
The last solution the student tried was this, but the teacher marked this solution incorrect:
(define-fun Prog ...
This solution was incorrect because it did not work for the following inputs:
x3  =  (- 3)
x2  =  (- 2)
x1  =  (- 4)
```

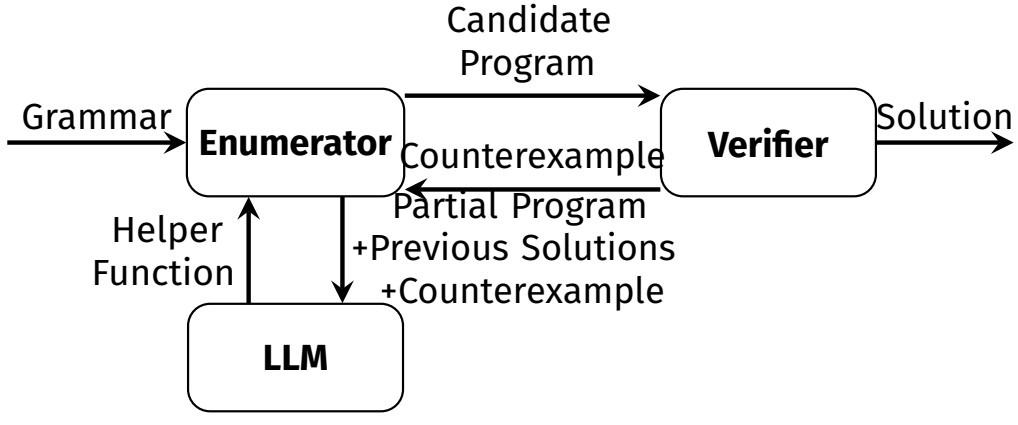
Weight Update

- we add helper function as a standalone rule (of according type)
- we update all weights by considering the syntax of helper function (as previously)

pCFG-synth



iLLM-synth



Enumerators

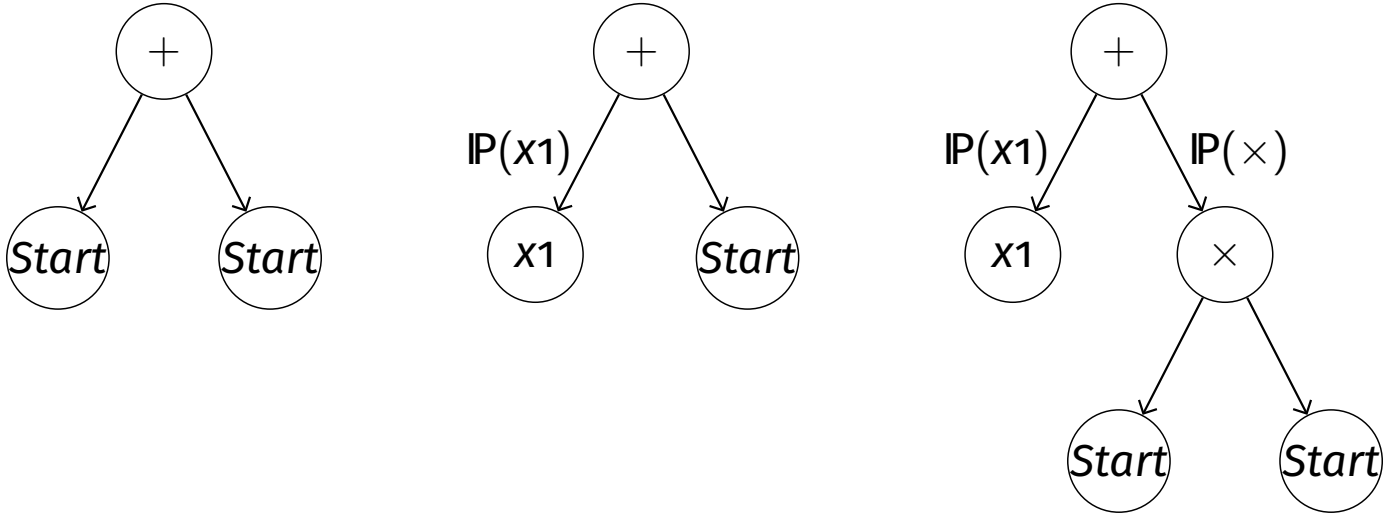
Different search methods:

- Top-down enumerator.
- A^* enumerator.

Top-down Enumerator

Top-down enumerator

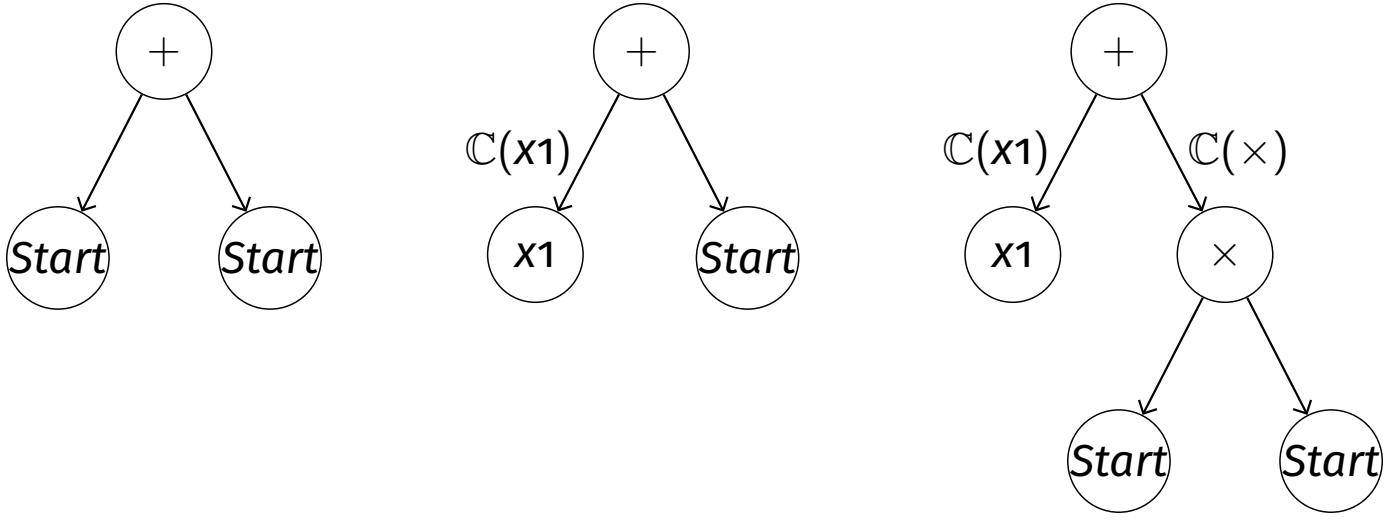
- Uses probabilistic rule to navigate the grammar tree.
- Generates unique programs, discarding duplicates and respecting a depth limit.
- Prioritizes new and complete programs to improve search productivity.



A^* Enumerator

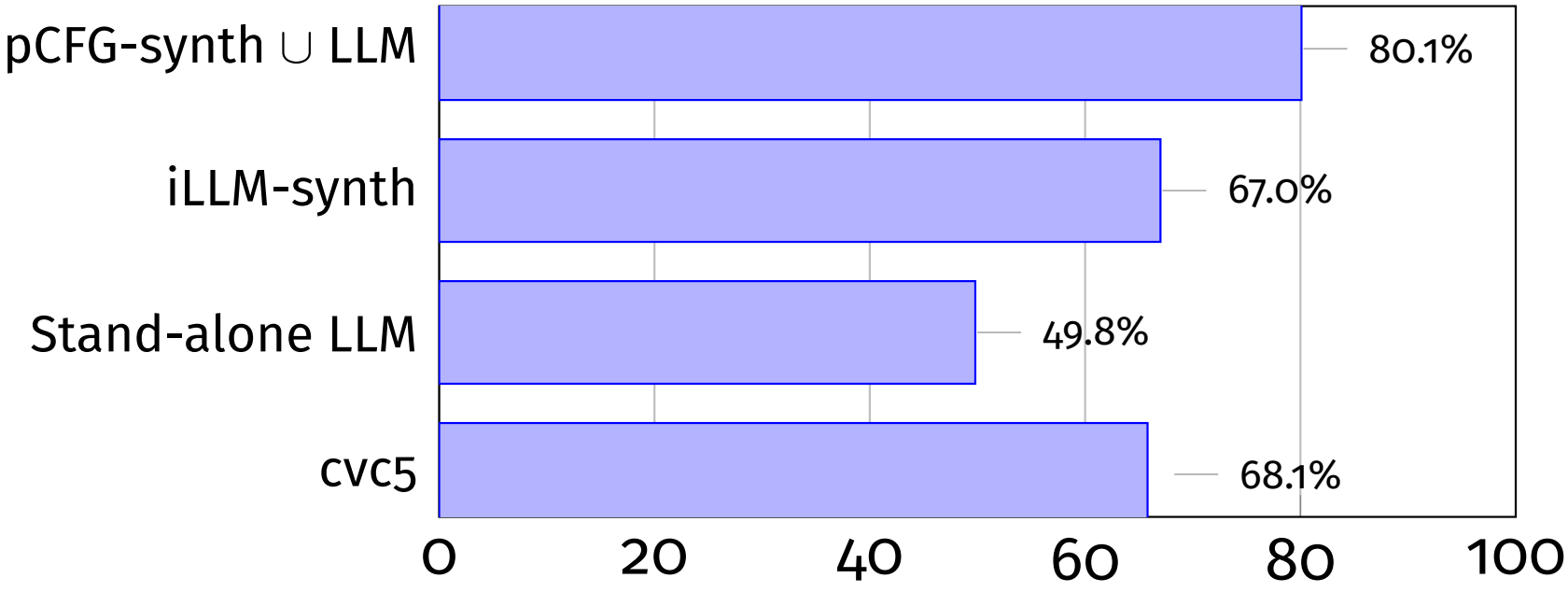
A^* enumerator

- Chooses paths based on minimizing current path cost plus estimated cost to goal.
- Focuses on paths with lower combined actual and predicted costs.



Some Results (600s Timeout)

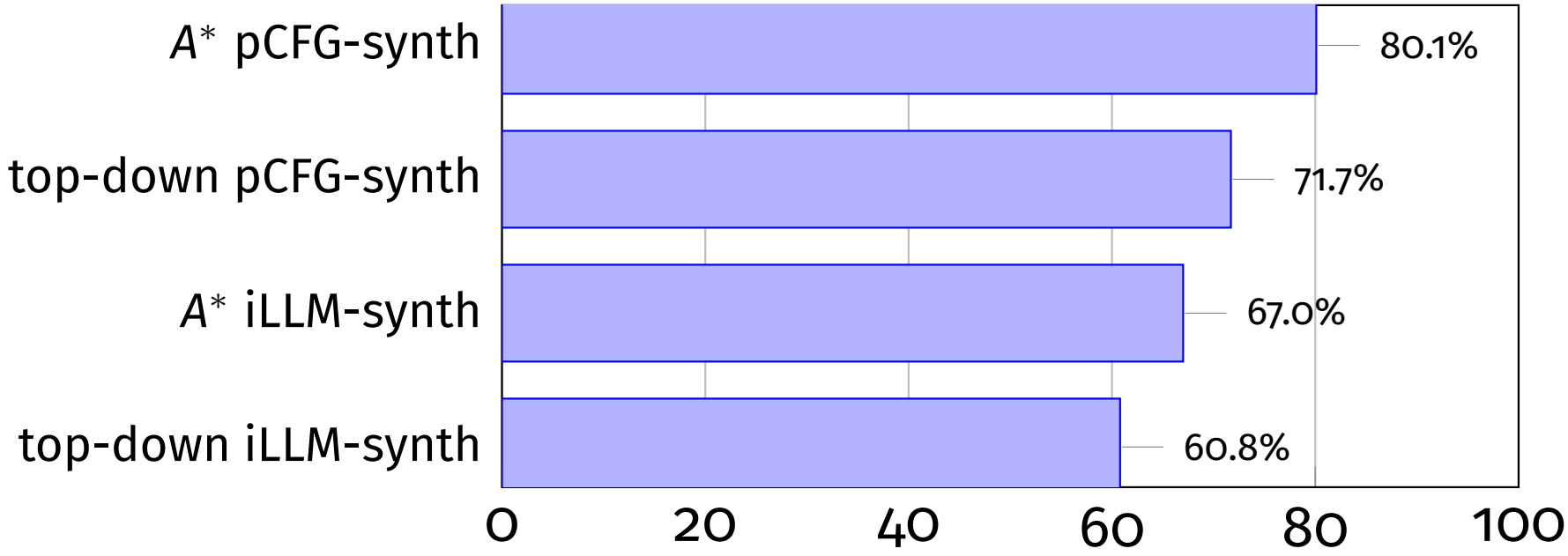
Effectiveness of various synthesis methods



- The average length of a solution: LLM is 4.7x than cvc5.

Summary of Results

Effectiveness of various enumerators



- A^* $>$ top-down.

Big Ugly Table

	BV (384)		LIA (87)		INV (138)		Total (609)	
Methods	#	time(s)	#	time(s)	#	time(s)	#	%
LLM only	137	*13.5*	54	*7.10*	112	*29.2*	303	49.8%
e-pCFG-synth ◇	196.0	48.3	24.0	40.0	25.4	100.5	245.4	40.3%
A^*-pCFG-synth	262	60.1	35	72.7	25	99.7	322	52.9%
LLM ∪ *e*-pCFG-synth	255.0	*37.0*	64.0	*17.20*	117.7	*40.4*	436.7	71.7%
LLM ∪ A^*-pCFG-synth	**305.0**	*35.0*	65.0	*18.1*	**118.0**	*33.6*	**488.0**	80.1%
e-iLLM-synth ◇	241.0	*88.2*	63.4	*9.3*	65.3	*25.4*	370.0	60.8%
A^*-iLLM-synth ◇	272.3	*24.6*	**68.3**	*20.8*	67.3	*43.6*	408.0	67.0%
enumerator◇	142.7	7.2	25.0	1.53	21.0	3.2	188.7	31.0%
A^*	253.0	25.4	34.0	73.19	22.0	31.1	309.0	50.7%
cvc5	292.0	17.1	43.0	19.53	80.0	23.6	415.0	68.1%

Failure of standalone LLM

- Constraints too long/complex
- simple syntactic errors (wrong place for operators)
- wrong nesting (e.g. if-then-else)
- ...

Neuro-symbolic approach can help with most problems.

Thank you

Feel free to contact me for questions, ideas, collaboration, ...

julian.parsert@gmail.com

Interests, Ideas, Etc.

Interests

Computer-Aided Verification is important!

I want to make it better, stronger, and easier for everyone to use!

(Computational) logic/formal methods are just interesting by themselves.

Research Ideas/Plans

From my thesis:

- Learning Non-Termination?
- (Un)decidable fragments of synthesis in SMT
- More complex predictors for MCTS

In general:

- Unrealizability: When/How can we prove that a function *cannot* be synthesized? Following LPAR paper on counter-models.
- Differentiable software monitoring
- Self-supervised learning on embeddings (to find fragments?)
- ...

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

2. Anti-Unification on Terms

$$(2 * 1) + 8 \sqcup (1 * 3) + 5$$

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

2. Anti-Unification on Terms

$$(2 * 1) + 8 \sqcup (1 * 3) + 5$$
$$\rightsquigarrow (2 * 1) \sqcup (1 * 3) + 8 \sqcup 5$$

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

2. Anti-Unification on Terms

$$(2 * 1) + 8 \sqcup (1 * 3) + 5$$
$$\rightsquigarrow (2 * 1) \sqcup (1 * 3) + 8 \sqcup 5$$
$$\rightsquigarrow (2 \sqcup 1) * (1 \sqcup 3) + w$$

1. Choose a set of Terms

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

2. Anti-Unification on Terms

$$\begin{aligned}
&(2 * 1) + 8 \sqcup (1 * 3) + 5 \\
&\rightsquigarrow (2 * 1) \sqcup (1 * 3) + 8 \sqcup 5 \\
&\rightsquigarrow (2 \sqcup 1) * (1 \sqcup 3) + w \\
&\rightsquigarrow u * v + w
\end{aligned}$$

lgg $u * v + w$ with 3 new variables

3. Replace terms with $F(u, v, w)$

$$10 * 1 = (2 * 1) + 8$$
$$(1 * 3) + 5 = 8$$

3. Replace terms with $F(u, v, w)$

$$10 * 1 = F(u, v, w)$$
$$F(u, v, w) = 8$$

3. Replace terms with $F(u, v, w)$

$$10 * 1 = F(u, v, w)$$
$$F(u, v, w) = 8$$

4. Unification of $u * v + w$ with terms for arguments

$$(2 * 1) + 8 \stackrel{?}{=} u * v + w \qquad u \mapsto 2, v \mapsto 1, w \mapsto 8$$
$$(1 * 3) + 5 \stackrel{?}{=} u * v + w \qquad u \mapsto 1, v \mapsto 3, w \mapsto 5$$

$$10 * 1 = F(2, 1, 8)$$
$$F(1, 3, 8) = 8$$

What is Second-Order Unification?

$$F(g(b, g(a, a)))\{F \mapsto \lambda x.g(b, g(x, x))\}) =$$
$$(\lambda x.g(b, g(x, x)))g(b, g(a, a)) \rightarrow_\beta$$
$$g(b, g(g(b, g(a, a)), g(b, g(a, a))))$$

$$g(b, g(F(a), F(a)))\{F \mapsto \lambda x.g(b, g(x, x))\}) =$$
$$g(b, g((\lambda x.g(b, g(x, x)))a, (\lambda x.g(b, g(x, x)))a)) \rightarrow_\beta$$
$$g(b, g(g(b, g(a, a)), g(b, g(a, a))))$$

Decidable fragments of Second-Order (SO) Unification

- **Monadic Second-Order Unification** (W. Farmer, 1988). Signature only contains monadic functions and constants.
- **Linear Second-Order Unification** (J. Levy, 1996). Substitutions may use the bound variables at most once and Second-Order variables have arbitrary, fixed arity.
- **Bounded Second-Order Unification** (M. Schmidt-Schauss, 2004). Substitutions may use the bound variables at most a fixed number of times and Second-Order variables have arbitrary, fixed arity.
- **Context Unification** (A. Jez, 2014). Substitutions may use the bound variables precisely once and Second-Order variables have arity 1.

Undecidability of Second-Order (SO) Unification

- Undecidability of Higher-order unification (G. Huet, 1972). Reduction from Post Correspondence Problem.
- Undecidability of second-order unification (W. Goldfarb, 1981). Reduction from Hilbert's 10^{th} problem.
- Undecidability of second-order in the following cases (W. Farmer, 1991). Reduction from Simultaneous Rigid Reachability.
  - Each SO variable occurs at most twice and there are only two SO variables.
  - One Unary SO variable.
  - For some fixed $n \geq 0$, **(i)** arguments of all SO variables are ground terms of size $< n$, **(ii)** the arity of all SO variables is $< n$, **(iii)** Occurances of SO variables ≤ 5, and **(iv)** there is either a single unary SO variable or there are two SO variables and no FO variables.

Undecidability of Second-Order (SO) Unification

- Undecidability of Second-order unification in the following case (H. Ganzinger *et al.*, 1998). Reduction from universal turing machine.
  - One second-order variable and at least 8 first-order variables.
- Undecidability of Second-order unification in the following cases (J. Levy and M. Veanes, 2000). Reduction from Simultaneous Rigid Reachability.
  - Two SO variables and no FO variables.
  - One SO variable and only ground arguments.

No Further Generalization for 24 Years.

What's Left??

Second-Order Ground Unification

- So far undecidability requires one of the following:
  A) Multiple Second-order variables.
  B) Occurances of first-order variables.
- We Ask the following:

  Are first-order variables Important for undecidability?

- we show that **One SO variable** and **No** FO variables is enough for undecidability.
- Our reduction exploits Hilbert's 10^{th} problem, also known as:

Theorem (Matiyasevich–Robinson–Davis–Putnam theorem)

Given a polynomial $p(\overline{x})$ with integer coefficients, finding solutions in $\mathbb{N}$ to $p(\overline{x}) = 0$ is undecidable.

Second-Order Ground Unification

Why is SOGU Interesting?

- Observe that such unification programs are closely related to function synthesis task such as
  A) Programming-By-Example (PBE)
  B) Syntax-Guided Synthesis (SyGuS)
- Often the contraints for such tasks are Input-Output examples using ground terms and a single function variable.
- While uncommon, considering nested variables could be of interest depending on the application domain.
- Additionally, it is often the case that a task requires checking equavalence modulo ground constraints on the solution.