Program Synthesis

An Introduction

Yu-Zhe Shi

July 25th, 2020

Overview

- Concepts of program synthesis.
- Domain Specific Language.
- Enumerative Search.
- Constraint Solving.
- Stochastic Search.

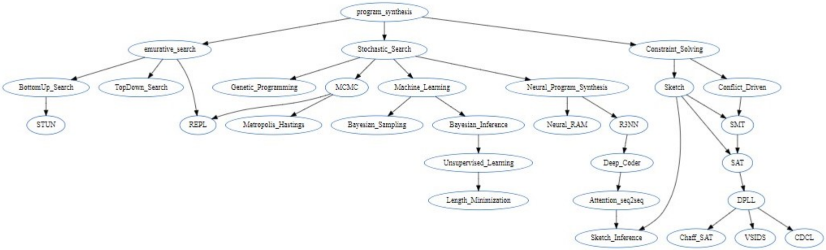
What is Program Synthesis?

- **Automatically**.
- **Find** programs from underlying programming language.
- **Satisfy** user intent explained by constraints.
- **Second-Order**.
- **Domain-Specific Language** (constrast to General Purpose Language).

Dimensions

- User intent:
  - Logical Specification between inputs and outputs.
  - Input-output Examples.
  - Step-by-step description (Trace).
  - Partial program, relative programs.
- Search Space:
  - Operators.
  - Control Structure.
- Search Technique:
  - Enumerative Search (bottom-up).
  - Deduction (top-down).
  - Constraint Solving.
  - Statistical Techniques.

Road Map


program_synthesis
emurative_search
Stochastic_Search
Constraint_Solving
BottomUp_Search
TopDown_Search
Genetic_Programming
MCMC
Machine_Learning
Neural_Program_Synthesis
Sketch
Conflict_Driven
STUN
REPL
Metropolis_Hastings
Bayesian_Sampling
Bayesian_Inference
Neural_RAM
R3NN
SMT
Unsupervised_Learning
Deep_Coder
SAT
Length_Minimization
Attention_seq2seq
DPLL
Sketch_Inference
Chaff_SAT
VSIDS
CDCL

Established Researchers & Teams

- PROSE Team, Microsoft: Sumit Gulwani, Microsoft, Obtained Ph.D. at UC Berkeley. `https://www.linkedin.com/in/sumit-gulwani/`
- Sketch, MIT: Armando Solar-Lezama, CSAIL, MIT, Obtained Ph.D. at MIT. `https://people.csail.mit.edu/asolar/` (Solar-Lezama + J.B.Tenenbaum = Creativity!)
- STOKE, Stanford: Alex Aiken, CS, Stanford, Obtained Ph.D. at Cornell. `http://theory.stanford.edu/~aiken/`

Task: Semantic Parsing

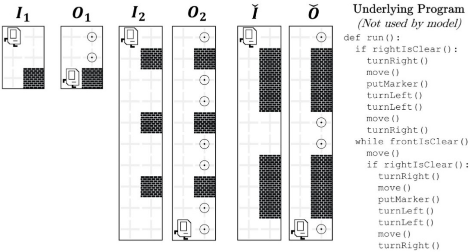
- StackOverflow Question Code Dataset (SQCD): Semantic Parsing, English to Python.
- CoNaLa: The Code/Natural Language Challenge: Semantic Parsing, English to Python.
  e. g. {
  ```
  "intent": "How do I check if all elements in a
  list are the same?",
  "rewritten_intent": "check if all elements in
  list `mylist` are the same",
  "snippet": "len(set(mylist)) == 1",
  "question_id": 22240602
  ```
  }
- WikiSQL: Semantic Parsing, English to SQL.

Task: Algorithmic Synthesis

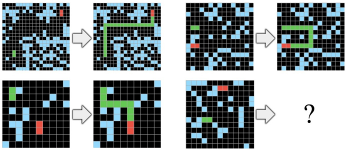
- NAPS: Dataset containing preprocessed problems from algorithmic competitions along with imperative descriptions and examples.
  e. g. [
  `input = [1, 2, 5, 4, 6, 3],`
  `output = [1, 4, 9, 16, 25, 36]`
  ]

Task: Planning

- Karel Language and Benchmark: Robot planning.



- Abstracting and Reasoning Challenge: Imitation Learning.



PBE vs. PBD

- Programming by Example: A single input-output example `factorial(6) = 720`.
- Programming by Demonstration: An example with trace `factorial(6) = 6*(5*(4*(3*(2*1))))=720`.

Challenges

- *How do you find a program that matches the observation?*
- *How do you know the program you found is the one you were actually looking for?*
- Intractability of Programming Space: Exponential growth of non-trivial search space.
- Diversity of User Intent: Specification is as sophisticated as programming; User intent is ambiguous.

Domain Specific Language

- Subsets of general-proposed language.
- No side effects(Pure functions).
- Concise and Experissive.

Abstract Syntax Tree

- The most common representation of a program.
- `expr:=term | term+expr`
  `term:=(expr) | term*term | N`
- `data AST = Num Int | Plus AST AST | Times AST AST`

Context-free Grammar

Definition

Context-free Grammar $G = (V, \Sigma, R, S)$

- V is a finite set of non-terminal symbols.
- Σ is a finite set of terminal symbols.
- R is a finite set of rules of the form $X \rightarrow Y_1 Y_2 \dots Y_n$, $X \in V$, $n \geq 0$, $Y_i \in (V \cup \Sigma)$
- S is a distinguished start symbol.

CFG: Left-most Derivations

Definition

Derivations $s_1 s_2 \ldots s_n$

- $s_1 = S$
- $s_n \in \Sigma^*(\Sigma^* \subseteq \Sigma)$
- s_i is derived from s_{i-1} by picking the left-most non-terminal X in s_{i-1} and replace X by the rule in $\{X \rightarrow \beta\} \in R$

Probabilistic CFG

- τ_G is the set of all possible derivations under grammar G.

Definition

PCFG

- $G = (V, \Sigma, R, S)$
- Parameter q, $\forall X \in V, \sum_{\alpha \rightarrow \beta \in R: \alpha = X} q(\alpha \rightarrow \beta) = 1$ where $q(\alpha \rightarrow \beta)$ denotes the conditional probability of choosing rule $\alpha \rightarrow \beta$ in a derivation.
- For derivation t in τ_G containing rules $\alpha_1 \rightarrow \beta_1, \ldots, \alpha_n \rightarrow \beta_n$,

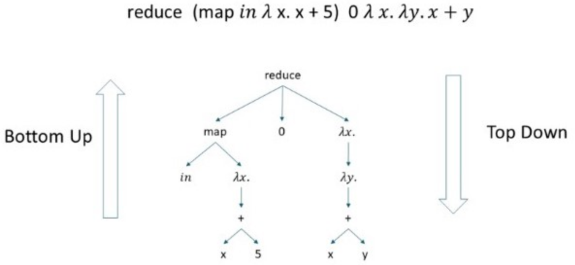
$$p(t) = \prod_{i=1}^{n} q(\alpha_i \rightarrow \beta_i) \tag{1}$$

An Example

- $V = \{Init, Op, Dest, Num, Equal, Predecess, Success\}$
- $\Sigma = \{0, 1\}$
- $R, q = \{S \rightarrow Init : 1, Init \rightarrow Num : 0.5, Init \rightarrow Op : 0.5, Op \rightarrow Equal : 0.5, Op \rightarrow Predecess : 0.25, Op \rightarrow Success : 0.25, \}$
- S

Enumerative Search

- Top-Down Tree Search: From root to input specification.
- Bottom-Up Tree Search: From leaf to output speciication.
- Bidirectional Search: Combination of top-down and bottom-up search.
- Offline Exhaustive Enumeration and Composition: retrive the program mapping to input-output pair.

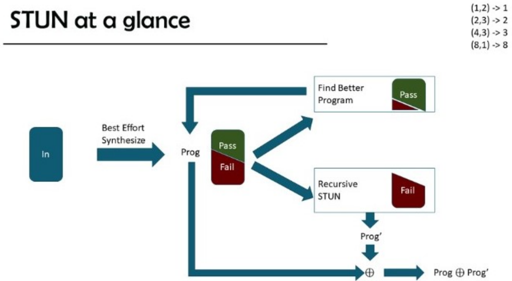


Algorithm: Bottom-Up Search

- Guiwani et al, *Recursive Program Synthesis*, CAV'13.
- Start with terminals!
- Prune the set of primitives at every step by eliminating those that are deemed to be *observationally equivalent*.
- Observationally Equivalent: Expressions that have the same output given same input.
- Drawbacks: Scalability.

Algorithm: Synthesis through Unification (STUN)

- Alur et al, *Synthesis through Unification*, CAV'15.
- No longer looking for a program thats works for all inputs in one shot.
- Search for multiple programs that work for different situations.
- An initial best-effort search to produce a program that works correct on some inputs.
- Input fails: improve on current program OR reconstruct a new program.
- Searching heuristic: When fail on an input, search for a better solution with that input.



Algorithm: Top-Down Search

- Feser et al, *Synthesizing data structure transformations from input-output examples*, SIGPLAN'15.
- Using the production rule of the grammar to generate candidate programs.
- Expand the expressions. First prune the expressions with the undesired types.
- Further pruning with additional deduction rules: Derive rules from known functions to unknown subexpressions:
  - Rules tell you that a candidate is not going to work.
  - Rules tell you that how to propagate input/outputs to subexpressions.

  e.g. `map x lambda y.expr`, if the input-output doesn't have same length...

Constraint Solving

Encoding the specification and syntactic program restrictions into a single formula.

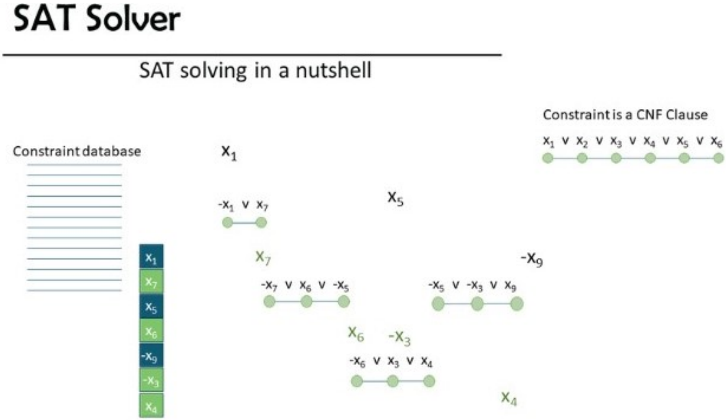
- Component-Based Synthesis:
  - End-to-end SAT encoding.
  - Sketch generation and completion: Program with holes.
- Solver-aided Programming: high level program argumented with constructs.
- Inductive Logic Programming.

Algorithm: Sketch

- Armando Solar-Lezama, *The Sketching Approach to Program Synthesis*, APLAS'08; Armando Solar-Lezama, *Program sketching*, IJSTTT'13.
- Parametric Program: different values of the parameters correspond to different programs in the space.
- Unknown Constants: `??`
- Generator Function: `generator int gen(int i){if(??) return i*?? + ??;}`
- Symbolic Execution: Run a program and produce symbolic values and constraints.
- Structural Hashing: Identify common sub-expressions and represent them in the same node.
- Representation of sets: Represent set Φ as predicate $P_\Phi(\phi)$ iff $\phi \in \Phi$

Algorithm: Sketch

- Transform constraints to Conjunctive Normal Form.
- One-hot encoding indicating the true value.
- Solving SAT Problems: SAT Solvers based on DPLL.



Improvements on SAT Solver

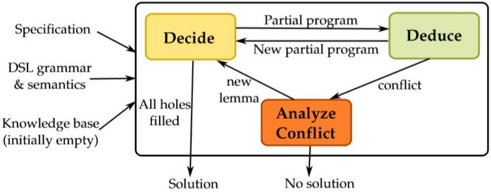
- Conflict Driven Clause Learning(CDCL), GRASP SAT Solver:
  - When contradict, trace back a small set of assignments that lead to the contradiction.
  - Define a conflict graph that shows the possible conflict clauses.
- Two Literal Watching, Chaff SAT Solver:
  - There is no need to keep track of all unassigned literals because only the last two unassigned literals determines the 'action' of the clause.
  - For every clause, we keep track of two literals that haven't been set.
- Heuristic on selecting variable, Variable State Independent Decaying Sum (VSIDS):
  - Keep a score for every variable that is additively dumped based on how much it is used.
  - Decayed over time. (Expontional Moving Average)

SMT Solver

- Satisfiability Modulo Theory:
  - Goal: Either Find an assignment to satisfy a logical formula or prove the unsatisibililty of a logical formula.
- Leverage SAT Solver.
  - Initially take all predicates and replace them with boolean variables.
  - Eager Approach: Explicitly generate boolean constraints.
  - Lazy Approach: Get a solver that interacts with the SAT solver and incrementally add constraints to the boolean abstraction.

NEO: Conflict-Driven Learning

- Feng et al, *Program Synthesis using Conflict-Driven Learning*, PLDI'18.
- In SAT/SMT solving, NEO learns a root reason for the failure of branch search (conflict) and add it to the constraints to avoid similar mistakes.
- e.g. `[1,2,3]`$\rightarrow$`[2,4]`, eliminates functions like `map`, `sort`, `reverse`, which are called *equivalent modulo conflict*.
- Key Procedures:
  - Decide: which hole to fill and how to fill it with DSL.
  - Deduce: Keep Track of use Lemmas.
  - Conflict Analyze: Find the root cause (minimal unsatisfiable) of the failure and learn new lemmas.



Stochastic Search

- Markov Chain Monto Carlo.
- Genetic Programming.
- Machine Learning.
- Neural-Guided Synthesis.

Algorithm: MCMC-MH (Stochastic SyGus Solver)

- Alur et al, *Syntax-guided synthesis*, FMCAD'13.
- Score function of expressions: Distribution over the domain of programs.

$$\pi = e^{-0.5C(e)} \tag{2}$$

  where $C(e)$ denotes the number of examples for which e is correct.
- The probability of acception:

$$P_A(\mathbf{x}^*|\mathbf{x}^{t-1}) = \min\left(1, \frac{p(\mathbf{x}^*)P(\mathbf{x}^{t-1}|\mathbf{x}^*)}{p(\mathbf{x}^{t-1})P(\mathbf{x}^*|\mathbf{x}^{t-1})}\right)) \tag{3}$$

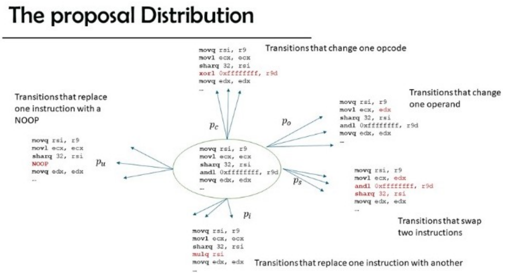
  , in this case

$$P_A(e, e') = \min\left(1, \frac{\pi(e)}{\pi(e')}\right) \tag{4}$$

- Shortcomings: Scoring Function isn't precise enough; The proposal distribution only make big changes to the program.

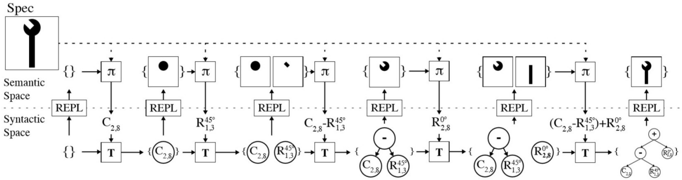
Algorithm: More Specified AST Synthesis

- Schkufza et al, *Stochastic superoptimization*, ASPLOS'13.
- 5 kinds of probability.
- $$\pi(Prog) = \exp(-\beta(Crct(Prog, Prog')) + perf(Prog, Prog')) \tag{5}$$
- Correct measures the Hamming Distance between outputs; Performance serves as cost functions. First ignore the Performance term to obtain large steps.



Search Process with an Interpreter

- Ellis, Solar-Lezama and Tenenbaum, *Write, Execute, Assess: Program Synthesis with a REPL*, NIPS'19.
- Challenge: Tiny changes in syntax lead to huge changes in semantic.
- Read-Evalutaion-Print-Loop: propose new code to write, assess the prospects of codes written-so-far.
- REPL serves as a bridge to apply Markov Decision Process jointly on both syntax space and semantic space.
- Sequential Monte Carlo Method: Maintaining the policy-desired programs.



Stochastic Search: Genetic Programming

- Katz et al, *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*, ATVA'08.
- 4 operations: crossover, mutation, duplication, deletion.
  - Mutation: Random change.
  - Crossover: Useful subprograms from other programs.
- Hierarchical programs vary on different sizes and shapes.
  - A set of terminal and function symbols.
  - Fitness measure.
  - Search parameters: population, number of expressions, probability of the 4 operations.
  - Termination criterion.

Crossover

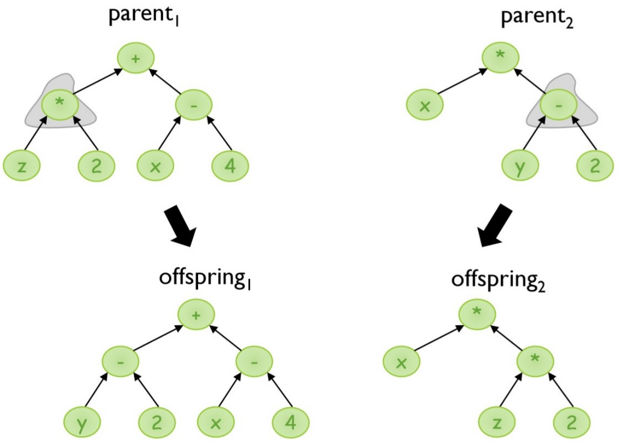


Figure: Crossover

Mutation

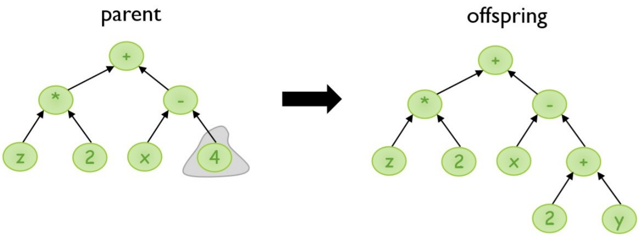


Figure: Mutation

Stochastic Search: Machine Learning

- Menon et al, *A Machine Learning Framework for Programming by Example*, ICML'13.
- Learn the weights for the rules R in PCFG G.
- The weights conditioned on the input-output examples are trained offline.
- Hand-crafted features. e.g. `sort_cue` whether the output strings are sorted.

```
Anthony Hopkins
Al Pacino
Tom Hanks
Tom Hanks
Nicolas Cage
```

$\rightarrow$

```
Anthony Hopkins (1)
Al Pacino (1)
Tom Hanks (2)
Nicolas Cage (1)
```

Production	Probability	Production	Probability
P→join(LIST,DELIM)	1	CAT→LIST	0.7
LIST→split(x,DELIM)	0.3	CAT→DELIM	0.3
LIST→concatList(CAT,CAT,CAT)	0.1	DELIM→"\n"	0.5
LIST→concatList("(",CAT,")")	0.2	DELIM→" "	0.3
LIST→dedup(LIST)	0.2	DELIM→"("	0.1
LIST→count(LIST,LIST)	0.2	DELIM→")"	0.1

Bayesian Program Synthesis

- Form our belief in the relative likelyhood desired by the user (priori) and update our belief with new evidence (I/O examples).
- A strict generation of the original program synthesis formulation. Let O be Observation Evidence, f denote desired program

$$P(O|f) = \begin{cases} U(e), & \forall e \in O, Con(O \cup f) \\ 0, & \exists e \in O, \neg Con(O \cup f) \end{cases} \tag{6}$$

- $$P(f|[in_i, out_i]) \approx P(f) \prod_{[in_i, out_i] \in E} P(out_i|f, in_i) \tag{7}$$

Unsupervised Learning

- Ellis, Solar-Lezama and Tenenbaum, *Unsupervised Learning by Program Synthesis*, NIPS'15.
- Both the inputs and the functions are unknown!
- Learning noisy Visual Concepts.
- Objective of Unsupervised Learning:

$$\min_{f, I_i \in E} -\log P_f(f) - \sum_{i=1}^{N} \left(\log P_{x|z}(x_i | f(I_i)) + \log P_I(I_i) \right) \tag{8}$$

  where the three terms are length of generated program, data reconstruction error and input encoding length respectively.
- Generating SMT Formulae that computes description length of program and the output given an input.
- Additional Constraint on SMT Solver: Generating description as short as possible.

Unsupervised Learning: To Marginalize or Not to Marginalize?

- Should we marginalize over the inputs or not?
- Marginalize: find the $P(f, [in_i])$ that maximizes $P(f, [in_i]|[out_i])$.
- Not Marginalize: maximize $P(f|[out_i]) = \sum_{[in_i]} P(f, [in_i]|[out_i])P([in_i])$
- Optimize the joint distribution!

Algorithm: Length Minimization

- $$P(f) = \begin{cases} \frac{1}{Z} e^{-len(f)}, & f \in \mathcal{F} \\ 0, & otherwise \end{cases} \tag{9}$$
- Conventional Bottom-Up Search guarantees the minimization of height of the search tree.
- However, the improvements of Bottom-Up Search and Top-Down Search no longer guarantees the minimization.

Algorithm: Bayesian Sampling

- Ellis, Solar-Lezama and Tenenbaum, *Sampling for Bayesian Program Learning*, NIPS'16.
- Form the synthesis problem into SAT Solving problem. Instead of search for one program, approximately sample the program space and incrementally upgrate the SAT Solver.
- The example follows p-distribution, we aim to sample a $q(\cdot)$ in program space that has low KL-Divergence from $p(\cdot)$.
- d serves as the threshold of description length of the program.

$$q(x) \propto \begin{cases} 2^{-|x|}, & |x| \leq d \\ 2^{-d}, & otherwise \end{cases}, A(x) \propto \begin{cases} 1, & |x| \leq d \\ 2^{-|x|+d}, & otherwise \end{cases} \tag{10}$$

  where $A(x)$ is the acception ratio of an expression.
- y denotes the auxiliary assignments of program space where $y_i = 1$ if $|x_i| \leq d$, $r(x) = \sum_y r(x, y)$, $q(x) = A(x)r(x)$

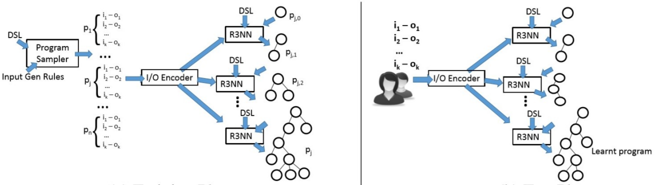
Stochastic Search: Neural Program Synthesis

- Key idea: Developing a continuous representation of the atomitic operations of the network.
- End-to-end training/Reinforcement Learning.
- Shortcomings: Weak Interpretibility, Resource Consuming.

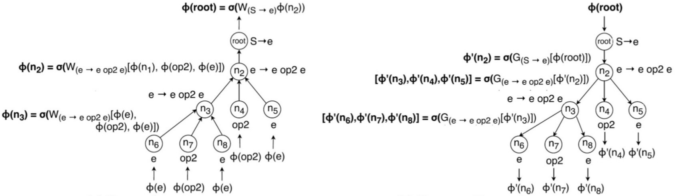
Neural FlashFill

- Parisotto et al, *Neruo-Symbolic Program Synthesis*, ICLR'17.
- Discoverying input substrings copied to output: Cross-Correlation based encoder presenting a continuous representation between I/O.
- Recursive-Reverse-Recursive Neural Network (R3NN): Constructing programs incrementally.

Neural FlashFill



(a) Training Phase (b) Test Phase

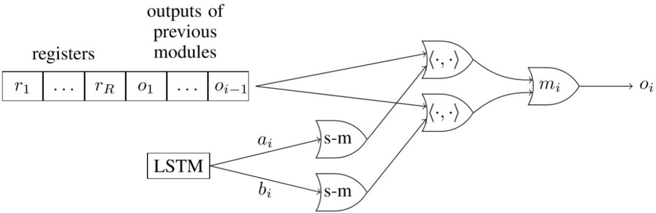


(a) Recursive pass (b) Reverse-Recursive pass

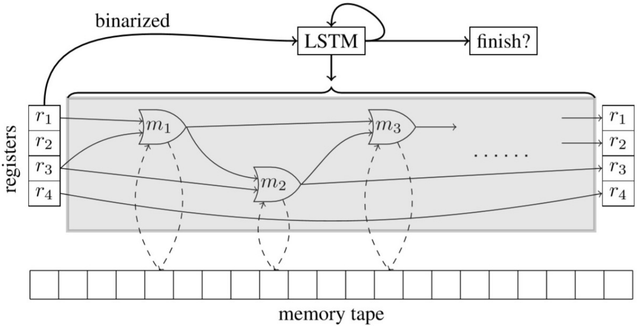
Neural RAM

- Kurach and Andrychowicz et al, *Neural Random-Access Machines*, ICLR'16.
- Learns a circuit composed with a given set of modules.
- Obtain continuous representation of all modules, learn a controller.

Neural RAM



outputs of previous modules
registers
r_1
...
r_R
o_1
...
o_{i-1}
$\langle\cdot,\cdot\rangle$
$\langle\cdot,\cdot\rangle$
m_i
o_i
a_i
b_i
LSTM
s-m
s-m



binarized
LSTM
finish?
registers
r_1
r_2
r_3
r_4
m_1
m_2
m_3
......
r_1
r_2
r_3
r_4
memory tape

Deep Coder

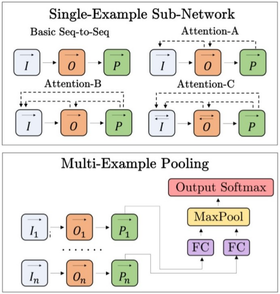
- Balog et al, *Deep Coder: Learning to Write Programs*, ICLR'17.
- Encode the features of specification, then decodes it to a vector, where every dimension corresponds to the probability of an element of the grammar.
- Learns a distribution over the candidate functions.
- Use the distribution to guide a depth-first top-down enumerative search.

(+1)	(-1)	(*2)	(/2)	(*-1)	(**2)	(*3)	(/3)	(*4)	(/4)	(>0)	(>0)	(%2==1)	(%2==0)	HEAD	LAST	MAP
.0	.0	.1	.0	.0	.0	.0	.0	1.0	.0	.0	1.0	.0	.2	.0	.0	1.0

FILTER	SORT	REVERSE	TAKE	DROP	ACCESS	ZIPWITH	SCANL1	+	-	*	MIN	MAX	COUNT	MINIMUM	MAXIMUM	SUM
1.0	1.0	.7	.0	.1	.0	.4	.0	.0	.1	.0	.2	.1	.0	.0	.0	.0

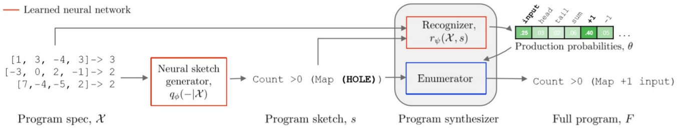
Learn from Noisy Example

- Devlin et al, *RobustFill: Neural Program Learning under Noisy I/O*, ICML'17.
- An end-to-end differentiable version of FlashFill that's trained on a large volume of synthetically generated tasks.
- Attention RNN Representation of I/O examples.



Infer Sketch

- Nye, Hewitt, Tenenbaum and Solar-Lezama, *Learning to Infer Sketch*, ICML'19.
- Specifications that human can most easily provide.
- Generating Sketch from example or nature language: seq-to-seq-RNN with Attention.
- Enumerative search guided by a recognizer that predicts the likelihood of the program filling in the hole.

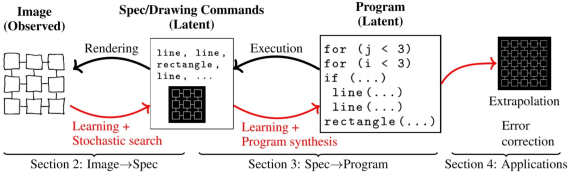


Reinforcement Learning

- Verma et al, *Programmatically Interpretable Reinforcement Learning*, ICML'18.
- Represent policy using domain specific language.
- Firstly learn a neural network by DRL to represent the policies.
- Then produce local search over programmatic policies that minimize the L2 distance from neural oracle (or most closely imitates the behavior of its neural counterpart).

Graphics Program

- Ellis, Solar-Lezama and Tenenbaum, *Learning to Infer Graphics Programs from Hand-Drawn Images*, NIPS'18.
- Learn to convert hand drawings into LaTeXprograms.
- CNN learning hand drawings as 'primitives', which serves as specification.
- Bottom-up Search Program Synthesis by learning a search policy that obtains a trade-off between search space and cost minimization.



Conclusion

- The Three Methods (Enumerative Search, Constraint Solving, Stochastic Search) are Combining!
- Cooperate with ABL!
- Program Invention?

References

- Guiwani et al, *Recursive Program Synthesis*, CAV'13.
- Alur et al, *Synthesis through Unification*, CAV'15.
- Feser et al, *Synthesizing data structure transformations from input-output examples*, SIGPLAN'15.
- Armando Solar-Lezama, *The Sketching Approach to Program Synthesis*, APLAS'08.
- Armando Solar-Lezama, *Program sketching*, IJSTTT'13.
- Feng et al, *Program Synthesis using Conflict-Driven Learning*, PLDI'18.

References

- Alur et al, *Syntax-guided synthesis*, FMCAD'13.
- Schkufza et al, *Stochastic superoptimization*, ASPLOS'13.
- Ellis, Solar-Lezama and Tenenbaum, *Write, Execute, Assess: Program Synthesis with a REPL*, NIPS'19.
- Katz et al, *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*, ATVA'08.
- Menon et al, *A Machine Learning Framework for Programming by Example*, ICML'13.
- Ellis, Solar-Lezama and Tenenbaum, *Unsupervised Learning by Program Synthesis*, NIPS'15.
- Ellis, Solar-Lezama and Tenenbaum, *Sampling for Bayesian Program Learning*, NIPS'16.

References

- Parisotto et al, *Neruo-Symbolic Program Synthesis*, ICLR'17.
- Kurach and Andrychowicz et al, *Neural Random-Access Machines*, ICLR'16.
- Balog et al, *Deep Coder: Learning to Write Programs*, ICLR'17.
- Devlin et al, *RobustFill: Neural Program Learning under Noisy I/O*, ICML'17.
- Nye, Hewitt, Tenenbaum and Solar-Lezama, *Learning to Infer Sketch*, ICML'19.
- Verma et al, *Programmatically Interpretable Reinforcement Learning*, ICML'18.
- Ellis, Solar-Lezama and Tenenbaum, *Learning to Infer Graphics Programs from Hand-Drawn Images*, NIPS'18.