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** (contrast 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.
Established Researchers & Teams

▶ PROSE Team, Microsoft: Sumit Gulwani, Microsoft, Obtained Ph.D. at UC Berkeley.  
https://www.linkedin.com/in/sumit-gulwani/

(Solar-Lezama + J.B. Tenenbaum = Creativity!)

Task: Semantic Parsing

  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.

```
def run():
    if rightIsClear():
        turnRight()
        move()
        putMarker()
        turnLeft()
        turnLeft()
        move()
        turnRight()
    while frontIsClear():
        move()
        if rightIsClear():
            turnRight()
        move()
        putMarker()
        turnLeft()
        turnLeft()
        move()
        turnRight()
```

- **Abstracting and Reasoning Challenge:** Imitation Learning.
PBE vs. PBD

- Programming by Example: A single input-output example
  \[\text{factorial}(6) = 720.\]

- Programming by Demonstration: An example with trace
  \[\text{factorial}(6) = 6 \times (5 \times (4 \times (3 \times (2 \times 1)))) = 720.\]
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 Expressive.
Abstract Syntax Tree

- The most common representation of a program.
- \( \text{expr} := \text{term} \mid \text{term} + \text{expr} \)
  \( \text{term} := (\text{expr}) \mid \text{term} \ast \text{term} \mid \text{N} \)
- \( \text{data AST} = \text{Num Int} \mid \text{Plus AST AST} \mid \text{Times AST AST} \)
Context-free Grammar

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

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

- $\tau_G$ 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 $\tau_G$ containing rules $\alpha_1 \rightarrow \beta_1, \ldots, \alpha_n \rightarrow \beta_n$,

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

(1)
An Example

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

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

- 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)

- No longer looking for a program that 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.

STUN at a glance

- (1,2) -> 1
- (2,3) -> 2
- (4,3) -> 3
- (8,1) -> 8
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

- 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 $\Phi$ 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 unsatisfiability 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

- 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 Monte Carlo.
- Genetic Programming.
- Machine Learning.
- Neural-Guided Synthesis.
Algorithm: MCMC-MH (Stochastic SyGus Solver)

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

▶ 5 kinds of probability.

\[ \pi(Prog) = \exp(-\beta(Crct(Prog, Prog')) + perf(Prog, Prog')) \]  \hspace{1cm} (5)

▶ Correct measures the Hamming Distance between outputs; Performance serves as cost functions. First ignore the Performance term to obtain large steps.

The proposal Distribution
Search Process with an Interpreter

- 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

- 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

- 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.

\[
\begin{array}{|c|c|}
\hline
\text{Production} & \text{Probability} \\
\hline
P \rightarrow \text{join}(\text{LIST}, \text{DELIM}) & 1 \\
\text{LIST} \rightarrow \text{split}(x, \text{DELIM}) & 0.3 \\
\text{LIST} \rightarrow \text{concatList}(\text{CAT,CAT,CAT}) & 0.1 \\
\text{LIST} \rightarrow \text{concatList}(\text{"\",CAT,"\"}) & 0.2 \\
\text{LIST} \rightarrow \text{dedup}(\text{LIST}) & 0.2 \\
\text{LIST} \rightarrow \text{count}(\text{LIST}, \text{LIST}) & 0.2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Production} & \text{Probability} \\
\hline
\text{CAT} \rightarrow \text{LIST} & 0.7 \\
\text{CAT} \rightarrow \text{DELIM} & 0.3 \\
\text{DELIM} \rightarrow \text{\"\"} & 0.5 \\
\text{DELIM} \rightarrow \text{"} & 0.3 \\
\text{DELIM} \rightarrow \text{"} & 0.1 \\
\text{DELIM} \rightarrow \text{\"} & 0.1 \\
\hline
\end{array}
\]
Bayesian Program Synthesis

- Form our belief in the relative likelihood 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

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

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

- Both the inputs and the functions are unknown!
- Objective of Unsupervised Learning:

\[
\min_{f, I_i \in E} - \log P_f(f) - \sum_{i=1}^{N} (\log P_{x_i|z}(x_i|f(I_i)) + \log P_{I_i}(I_i))
\]  

(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 $P(f, [in])$ that maximizes $P(f, [in]|[out])$.
- Not Marginalize: maximize
  $$P(f|[out]) = \sum_{[in]} P(f, [in]|[out])P([in])$$
- Optimize the joint distribution!
Algorithm: Length Minimization

\[ P(f) = \begin{cases} 
\frac{1}{Z} e^{-\text{len}(f)}, & f \in \mathcal{F} \\
0, & \text{otherwise}
\end{cases} \]  \hspace{1cm} (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

- Form the synthesis problem into SAT Solving problem. Instead of search for one program, approximately sample the program space and incrementally upgrade 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.

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

(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

- Discovering 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

- Learns a circuit composed with a given set of modules.
- Obtain continuous representation of all modules, learn a controller.
Neural RAM
Deep Coder

- 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.
Learn from Noisy Example

▶ 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

- 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

- 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

- Learn to convert hand drawings into \LaTeX{} programs.
- 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

- Feser et al, *Synthesizing data structure transformations from input-output examples*, SIGPLAN’15.
References

References