**ProgramMe**

How to make program interpretation clearer after generation from text? How to prove that program satisfy textual requirements? We know how to make code ideal!

**Challenge**

Train LLM to better understand program **interpretation** based on run-time (RT) **execution feedback** and **algorithmic nature** of program by using reinforcement learning (RL) and huge amount of syntactic code generated program.

Run-Time are Python, RDBMS (DDL, SQL), SMT solver (SMT-LIB), Mathworks Matlab, Wolfram Mathematica, Lean (potentially). Program’s interpretation is considered as new representations of code which should help programmer to understand specification and complex code which realize it to refine it:

* **Decomposition of programs** on **functionally independent atomic** units.
* **Formal high-level abstractions** (e.g. Algebraic Data Type).
* **Model of programs** (e.g. Bounded Model with constraints, Finite-State Machine).
* Maximizing **code coverage** with finding **counter-examples** to test program.

Why is needed additional layer as PSQL:

* **Intuition**: Many modern developers use .Net LINQ and Java Stream where core logic is based on SQL-oriented style of iteration through collection of entities. That’s why the reason is balance between popular simplicity and declarative power to declare general algorithms.
* **Ambiguity Reduction**: Natural language is inherently ambiguous. Formal PSQL can constrain interpretations, ensuring the translation to code is deterministic and unambiguous.
* **Specification Layer**: The PSQL can serve as a structured intermediary that captures the intent of the user in a machine-readable way. This layer is easier to validate and verify than natural language.
* **Training Efficiency**: Training ML models directly on code from natural language requires bridging a wide semantic gap. Introducing PSQL narrows this gap, simplifying the mapping and potentially improving model performance.
* **Error Isolation**: If the final code has issues, debugging is more manageable since the PSQL provides an intermediate layer to check whether the intent was captured correctly.

What are disadvantages:

* **Increased Complexity**. Two-step process adds an extra layer, requiring both a Text-to-PSQL translation and a PSQL-to-code translation. This increases overall system complexity and potential failure points.
* **Semantic Drift.** The PSQL might not capture all the nuances of the natural language input, leading to misinterpretations or incomplete representations of user intent.
* **Domain Limitation**. PSQL designed for a specific domain may struggle to adapt to broader or unforeseen use cases without significant reengineering.

How unit-tests execution feedback is helped?

* **Dynamic Symbolic Execution**: SMT-probes analyze program paths and suggest hard-to-reach branches to target.
* **Dynamic Exploration**: penalizing repetitive or trivial code should be used to explore diverse unit-tests.
* **Curriculum Learning**: training starts with simple unit tests for basic correctness and gradually introduces more complex tests to guide the LLM toward generating advanced code structures.

# Overview

**Ideal-code**

**Unit-tests code**

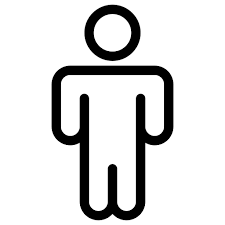
**SMT-LIB**

**code**

**Shema #1 (Dual to #2)**

**EvalLoop**

**Run-time**



**LLM**

**LLM**

**PSQL**

**Spec**

**Text**

**(NL)**

**Ideal-code**

**Text**

**Unit-tests code**

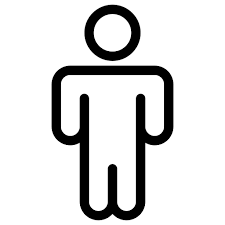
**SMT-LIB**

**code**

**Schema #2 (Dual to #1)**

**EvalLoop**

**Run-time**



**LLM**

**LLM**

**PSQL**

**Spec**

**Real-code**

# Concept

LLM inference can be split on the following phases:

1. LLM(Text) → PSQL1..n
2. LLM(PSQL(1..k), …, (n..m)) → Code(1..k…l), …, (n..m..p); RT(Code(1..k…l), …, (N..m..p)) → Reward(1..k…l), …, (n..m..p)
3. LLM(PSQL(1..k), …, (n..m), Code(1..k…l), …, (n..m..p)) → SMT(1..k…l), …, (n..m..p); RT(SMT(1..k…l), …, (n..m..p)) → Reward(1..k…l), …, (n..m..p)

# Interpretation

|  |  |  |
| --- | --- | --- |
| **#** | **Representation** | **Details** |
|  | Ideal-code | Optimized code with interpretation artifacts |
|  | PSQL | Functional Independencies based on Atoms |
|  | PSQL | Relational Decomposition by: Join, Where, GroupBy, OrderBy, Min/Max/Count aggregations |
|  | SMT-LIB | * Abstractions: Arithmetic, Induction, ADT, Bit-vector, etc. * Models: FSM, bounded model, invariants * Counter-examples as part of probing. |
|  | Unit-test | Cover program as much as possible by maximizing variance of explored program paths and guide execution into bugs like smart white-box fuzzing |

# Training

|  |  |  |
| --- | --- | --- |
| **#** | **Training** | **Realization** |
|  | RT feedback | Code results sampling / Unit-tests variance / SMT-LIB probes |
|  | Supervised with RL | Generator reward is based on correspondence with Classical algorithms sampling and performance |
|  | Supervised with RL | Critic’s reward is based on Unit-test variance and SMT-LIB probes |
|  | Artificial Game | In AlphaGo manner where PSQL can be used as combinator to synthetically cover enough space of programs during the game. Reasoning is happened by using Monte-Carlo Tree Search (MCTS) where state is current program and action is range of next generated programs |

# Estimation

|  |  |  |
| --- | --- | --- |
| **#** | **Activity** | **Cost (person/month)** |
|  | Dataset development for supervised learning based on popular dataset like HumanEval | 3 |
|  | Dataset development for supervised learning based on classical algorithms (Graph, Linear Programming, etc.) are as samples (about 1000 implementations). Format: Text + PSQL + Unit-tests + SMT-LIB + Data | 20 |
|  | Calculation variance of explored program paths and guide execution into bugs | 8 |
|  | Development of Gym environment for reward calculation:   * PSQL compile-time validation * Unit-tests and SMT-LIB processing | 4 |
|  | Open-source LLM development to add special layers for RL training | 2 |
|  | Artificial Game development | 6 |
|  | Testing and Training | 3 |
|  | **Total** | **46** |

# Questions

|  |  |  |
| --- | --- | --- |
| **#** | **Question** | **Answer** |
|  | Could Text represent a law and a request for automation of its application? |  |
|  | Is possible Text as entire data from database and request to create application? |  |
|  | PSQL is approach to go away from human’s natural language and figure more formal specification out. Is it right approach? |  |

# Knapsack example

// Atom of scalar product  
int dot(int[] a, int[] b) {  
 int s = 0;  
 for (int i = 0; i < Math.*min*(a.length, b.length); i++)  
 s += a[i] \* b[i];  
 return s;  
}  
// Atom of vector [x1, .., xn] where xi ∈ {0, 1} & n = length  
boolean isZeroOrOne(int[] vector, int length) {  
 if (length <= 0 || vector == null || vector.length != length)  
 return false;  
  
 for (int e : vector)  
 if (!(e == 0 || e == 1))  
 return false;  
 return true;  
}

|  |  |  |
| --- | --- | --- |
| **#** | **Input as Atoms and Operator (PSQL)** | **Output as Ideal-Code** |
|  | SELECT MAX(dot(:v, N.vector)) FROM ZeroOrOne AS N WHERE N.length = :n AND dot(:w, N.vector) <= :W | int maxDotVector(int[] w, int[] v, int n, int W) {  int[] m = new int[W + 1];   for (int ni = 0; ni < n; ni++) {  for (int wi = W; wi >= 0; wi--) {  if (w[ni] <= wi) {  var prevMax = m[wi - w[ni]];  if (m[wi] < prevMax + v[ni]) {  m[wi] = prevMax + v[ni];  }  }  }  }  return m[W]; } |

# SMT-LIB example

|  |  |  |
| --- | --- | --- |
| **Text** | **SMT-LIB** | **Python** |
| *Accordion represented as a concatenation of: an opening bracket [, a colon :, some (possibly zero) vertical line characters |, another colon, and a closing bracket ]. The length of the accordion is the number of characters in it. For example, [::], [:||:] and [:|||:] are accordions having length $4$, $6$ and $7$. (:|:), {:||:}, [:], ]:||:[ are not accordions. You are given a string $s$. You want to transform it into an accordion by removing some (possibly zero) characters from it. Note that you may not insert new characters or reorder existing ones. Is it possible to obtain an accordion by removing characters from $s$, and if so, what is the maximum possible length of the result?* | (declare-fun ys () String) (declare-fun result () Int) (declare-fun fi () Int) (declare-fun li () Int) (declare-fun s () String) (define-funs-rec ( ( filter ((s!1 String) (s!2 String)) String))  ( (let ((a!1 (>= (str.indexof s!2 (seq.unit (seq.nth s!1 0)) 0) 0))  (a!2 ((\_ filter 0)  (str.substr s!1 1 (- (str.len s!1) 1)) s!2)))  (let ((a!3 (ite a!1 (str.++ (seq.unit (seq.nth s!1 0)) a!2) a!2)))  (ite (= (str.len s!1) 0) "" a!3))))) (assert (let ((a!1 (ite (or (< fi 0) (< li 0) (> fi li)) ""  ((\_ filter 0) (str.substr s (+ fi 1) (- li fi)) "|:")))  (a!2 (distinct (seq.nth ys (- (str.len ys) 1)) (\_ Char 58)))  (a!3 ((\_ filter 0) (str.substr ys 1 (- (str.len ys) 1)) "|"))) (let ((a!4 (ite (or (= (str.len ys) 0)  (distinct (seq.nth ys 0) (\_ Char 58)) a!2)  (- 1) (+ (str.len a!3) 4))))  (and (= fi (str.indexof s "[" 0))  (= li (seq.last\_indexof s "]"))  (= ys a!1)  (= result a!4))))) | def main(s):  n = len(s)  ind = -1  f = False  for i in range(n):  if s[i] == '[':  f = True  elif s[i] == ':':  if f:  ind = i  break  bind = -1  f = False  for i in range(n - 1, -1, -1):  if s[i] == ']':  f = True  elif s[i] == ':':  if f:  bind = i  break  # print(ind,bind)  if ind == -1 or bind == -1:  return -1  elif ind >= bind:  return -1  else:  ans = 4  for i in range(ind + 1, bind):  if s[i] == '|':  ans += 1  return ans |