# Large Language Model Enabled Program Synthesis

*Yixuan Li*

李艺暄



Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

The University of Edinburgh

2025

# Abstract

Program synthesis aims to automate the construction of programs that satisfy user-defined specifications, yet existing approaches face trade-offs between correctness, scalability, and efficiency. One established approach is syntax-guided synthesis (SyGuS), in which the user provides both a formal specification of the desired program behaviour and a syntactic template, typically expressed as a context-free grammar. However, it becomes computationally expensive as the grammar grows and the number of possible programs increases. In contrast, machine learning methods, including large language models (LLMs), can generate code quickly by learning patterns from large datasets, but they often produce incorrect or unverifiable programs.

This thesis presents two complementary lines of research for integrating large language models (LLMs) into program synthesis. The first develops prompt engineering techniques to automatically select the most effective prompt-LLM configuration for a given synthesis task. The second investigates hybrid synthesis methods that combine the correctness guarantees of classical SyGuS techniques with the speed and generalisation capabilities of LLMs. This thesis further demonstrates the applicability of hybrid synthesis methods to domain-specific program lifting, where translating low-level code into high-level representations is essential for machine learning workloads. Together, these contributions advance the development of program synthesis frameworks that are both principled and practical.

# Lay Summary

Software engineers increasingly rely on large language models (LLMs) to draft code quickly, yet these systems still hallucinate bugs and offer no formal guarantees. Conversely, classical program synthesis tools explore programs with formal guarantees but become increasingly slow as the search space expands. This thesis presents two complementary lines of work that enhance the reliability and efficiency of program synthesis using LLMs.

First, we address the problem of choosing the most effective solver for a given synthesis task. Different LLMs and prompt styles vary widely in their performance, and symbolic solvers are still often better for some tasks. We develop CYANEA, an online prompt selection framework that uses a contextual multi-armed bandit to predict the best LLM–prompt pair or symbolic solver based on features of the input query.

Second, we focus on scenarios where LLMs generate incorrect solutions. Instead of discarding these failed attempts, we extract probabilistic grammars from them, capturing patterns in their structure. These grammars are then used to guide enumerative search, effectively turning LLM outputs into heuristics. This hybrid method, by combining statistical guidance from LLMs with formal symbolic search, boosts synthesis success by up to 80.1% compared to using the LLM alone, and outperforms state-of-the-art solvers on SyGuS benchmarks.

We also apply this hybrid synthesis framework to domain-specific program lifting, where low-level tensor kernels are translated into high-level representations in TACO, a tensor algebra DSL. Our system, STAGG, integrates LLM sketching with guided search and achieves 99% verified correctness on real-world benchmarks, outperforming existing tools in both speed and coverage.

Together, these contributions show how large language models can be used more effectively, either through careful prompt selection or as heuristic generators, to enable scalable and correct program synthesis.

# Acknowledgements

To the blessed memory of my maternal grandfather.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

This thesis is based on the following three peer-reviewed papers:

1. Chapter 3 is based on the paper *Online Prompt Selection for Program Synthesis* [60], published at *AAAI 2025*.

2. Chapter 4 is based on the paper *Guiding Enumerative Program Synthesis with Large Language Models* [63], published at *CAV 2024*.

3. Chapter 5 is based on the paper *Guided Tensor Lifting* [61], published at *PLDI 2025*.

The following peer-reviewed papers, also produced during my Ph.D., are not included in the thesis:

1. *Genetic Algorithms for Searching a Matrix of Metagrammars for Synthesis* [62] accepted at *SYNT 2023* workshop.

2. *HyGenar: An LLM-Driven Hybrid Genetic Algorithm for Few-Shot Grammar Generation* [104] published at *ACL 2025* findings.

3. *Unlocking Hardware Verification with Oracle Guided Synthesis* [112] published at *FMCAD 2025*.

*Yixuan Li*

李艺暄

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Program synthesis aims to automatically generate correct programs from high-level specifications, such as logical constraints, input-output examples, or natural language descriptions. Classical approaches to synthesis, particularly syntax-guided synthesis (SyGuS) and counterexample-guided inductive synthesis (CEGIS), provide rigorous correctness guarantees by exhaustively exploring and verifying the program space. However, these methods often suffer from a combinatorial explosion as the search space grows, making them difficult to scale to complex domains and large grammars.

In contrast, large language models (LLMs) have demonstrated remarkable proficiency in generating code snippets quickly and flexibly. Their outputs are often impressively fluent and creative, capturing a broad spectrum of programming patterns. However, LLM-generated code frequently lacks the formal correctness guarantees required for many synthesis tasks. Hallucinated logic or subtle semantic errors can undermine the reliability of these outputs, limiting their applicability in high-assurance settings.

This thesis proposes two complementary research topics. First, we study how to choose among many possible LLM–prompt combinations (or a tra-

ditional enumerator) for a given query, balancing accuracy, latency, and cost. Second, we study how to exploit even incorrect LLM solutions by converting them into guidance that accelerates enumerative search.

Chapter 3 introduces CYANEA, a framework designed to address a key practical challenge in program synthesis: how to dynamically select the most effective configuration for each new query. Today's synthesis workflows offer many options, symbolic solvers with formal guarantees, multiple large language models (LLMs), and a growing variety of prompt engineering techniques, but performance varies wildly across tasks. Even among LLMs, the effectiveness of a prompt can depend heavily on subtle features of the synthesis problem, such as logical complexity, grammar size, or constraint type. As a result, no single LLM–prompt pair consistently performs best across queries. Selecting the wrong pair not only reduces accuracy but also wastes time and money, especially when using commercial APIs that charge per token. We consider three kinds of existing prompting techniques: *manual*, *continuous*, and *discrete*. All three techniques attempt to improve the performance of LLMs without modifying the LLM itself. CYANEA learns to make these choices automatically. It observes features of each incoming synthesis query, such as the keywords, length, and logic type, and predicts which solver or LLM–prompt pair is most likely to succeed. Over time, it improves its decisions by learning from past outcomes. This learning process balances exploration (trying less-tested options) with exploitation (choosing proven ones), a strategy known in machine learning as a contextual multi-armed bandit. CYANEA also allocates time and token budgets across solvers in a cost-aware way, rather than using them uniformly. Together, this adaptive selection and resource allocation lead to a 37.2% increase in success rate compared to the best static solver, approaching oracle-level performance under fixed

computational budgets.

Chapter 4 focuses on mining the solution with LLM heuristics for SyGuS benchmarks. While LLMs are increasingly effective at generating code, direct prompting alone remains unreliable for such tasks. In initial experiments, one-shot prompting, where the model is asked to generate a complete program given the specification, solves only about half of the benchmarks. These failures often stem from subtle logical errors. However, even when the answers generated by LLMs are wrong, they tend to contain useful fragments, which we call probabilistic context-free grammars (pCFG), which we will introduce in Chapter 2. To capitalise on this partial knowledge, we first introduce pCFG-synth, which distils LLM-generated candidates into a probabilistic context-free grammar. This grammar encodes the LLM's biases as rule probabilities, allowing symbolic enumerators to prioritise candidates that are more likely to resemble a correct solution. To go further, we present iLLM-synth, which tightly integrates the LLM into the enumerative search loop. As the search proceeds and uncovers counterexamples or promising partial programs, these are incorporated into new prompts that elicit "helper" fragments from the LLMs. These helpers are then added back into the probabilistic grammar, continuously refining the heuristic as the search evolves. This creates a dynamic feedback loop in which symbolic reasoning and neural guidance iteratively reinforce each other.

Finally, Chapter 5 validates the hybrid synthesis approach in a real-world setting, lifting dense tensor kernels from C into the TACO DSL. This problem is representative of a growing class of code migration and optimisation tasks where legacy code must be ported to specialised DSLs to exploit modern compiler infrastructures and hardware accelerators. Manual translation is tedious, error-prone, and requires domain expertise, while fully

automated lifting is difficult due to the semantic gap between low-level imperative code and high-level tensor algebra constructs. To address this, we present STAGG, a synthesis framework that applies our hybrid methodology end-to-end. STAGG begins by querying an LLM for multiple solutions of the target program in the TACO DSL. These solutions, even when incorrect, reveal heuristics that are distilled into a probabilistic grammar. The symbolic search engine then explores candidates guided by this learned grammar. To ensure correctness, each candidate is subjected to an I/O checking phase and a bounded model checking phase that verifies functional equivalence with the original C kernel. This real-world application highlights the practical value of combining LLMs with program synthesis, where the LLM contributes rich, domain-specific heuristics without requiring manual engineering, and the synthesis technique ensures precision and correctness. On benchmarks drawn from existing literature, STAGG achieves 99% correctness, while delivering order-of-magnitude runtime improvements over previous lifting tools.

The remainder of this dissertation is structured to follow the progression from foundational concepts to applied systems. It begins with essential background (Chapter 2) and a review of related work (Chapter 6) to frame the contributions in context. The core chapters develop increasingly integrated hybrid synthesis techniques, first addressing selection among solvers, then combining LLMs with program synthesis, and finally applying these ideas to a real-world code lifting task. Each chapter is accompanied by an empirical evaluation that highlights its practical benefits. The dissertation concludes in Chapter 7 with reflections on limitations, broader implications, and future directions for research in LLM-guided synthesis.

# Chapter 2

# Background

The foundational background for the techniques and systems developed in this thesis is provided in this chapter, including the formal problem definition, the role of grammars in constraining the search space, and the syntax-guided synthesis (SyGuS) framework.

## 2.1  Program Synthesis

Program synthesis focuses on automated program creation that satisfies a high-level specification, which can be comprehensive, such as a basic, unrefined program, or incomplete, like a logical formula or a set of test cases. It has applications in planning [23], program analysis [27], data wrangling [31], and more.

## 2.2  Grammar

**Context-Free Grammar.** A context-free grammar is a 4-tuple

$$G = (V, \Sigma, R, S).$$

*V* is a finite set of variables, also known as non-terminal symbols. $\Sigma$ with $\Sigma \cap V = \varnothing$ is called the set of terminal symbols or alphabet. $R \subseteq V \times (V \cup \Sigma)^*$ is a finite relation describing the production rules of the grammar. We define $R_\Sigma = R \cap V \times \Sigma^*$, i.e. the set of rules restricted to those whose right-hand side only consists of terminal symbols. Elements of $(V \cup \Sigma)^*$ are known as words in sentential form. $S \in V$ is the start symbol of the grammar *G*.

Given a context-free grammar $G = (V, \Sigma, R, S)$ with $x, y \in (V \cup \Sigma)^*$ and $(\alpha, \beta) \in R$ we say that $x\alpha y$ yields $x\beta y$, written $x\alpha y \rightarrow x\beta y$. We say that *x* derives *y* written $x \rightarrow^* y$ if either $x = y$ or $x \rightarrow x_1 \rightarrow \ldots x_n \rightarrow y$ for $n \geq 0$. Finally, we define the *language* of a grammar

$$\mathscr{L}(G) = \{s \in \Sigma^* \mid S \rightarrow^* s\}.$$

We now introduce two extensions of context-free grammars:

**Weighted Context-Free Grammar, wCFG.** A weighted context-free grammar(wCFG) [65, 73] is a 5-tuple

$$W_G = (V, \Sigma, R, S, W)$$

such that $(V, \Sigma, R, S)$ is a context-free grammar and *W* is a function assigning a numeric value to each rule $r \in R$.

**Probabilistic Context-Free Grammar, pCFG.** A probabilistic context-free grammar [65, 73] is a 5-tuple

$$P_G = (V, \Sigma, R, S, \mathbb{P})$$

such that $(V, \Sigma, R, S)$ is a context-free grammar and $\mathbb{P}$ is a probability mass function assigning a probability $\mathbb{P}[r]$ to each rule $r \in R$. $\mathbb{P}_\Sigma$ is the probability mass function that assigns a probability to $\mathbb{P}_\Sigma[r]$ to each rule $r \in R_\Sigma$. A pCFG is a specific instance of a wCFG, where the weights are normalised to represent probabilities, satisfying the fundamental principle of probability theory that the sum of probabilities for all possible rules must equal one.

## 2.3   Satisfiability Modulo Theories

Boolean satisfiability (SAT) is the decision problem of determining whether a given Boolean formula can be made true by some assignment of truth values to its variables. Satisfiability Modulo Theories (SMT) extends SAT solving to richer logical contexts. Instead of only Boolean variables, SMT deals with formulas that include predicates over structured domains like numbers, bit-vectors, or data structures. It asks whether a first-order formula is satisfiable with respect to some background theory, generalising the SAT problem to constraints modulo theories such as arithmetic, arrays, or equality with uninterpreted functions. A theory $T$ is essentially a set of assumptions or axioms that constrain the behaviour of certain functions and predicates, usually corresponding to a well-defined domain like integers, reals, or bit-vectors.

## 2.4   Syntax-Guided Synthesis

In general, program synthesis is concerned with the generation (i.e., synthesis) of a program that satisfies a certain specification. Syntax-guided synthesis (SyGuS) describes a standardised function synthesis format that precisely defines a synthesis problem within first-order theories [14]. We will use the notation $\phi[F \mapsto f]$ to denote the replacement of all occurrences of $F$ in $\phi$ with the concrete implementation $f$, while substituting all arguments to $f$ by the arguments of $F$ in the same order.

A SyGuS problem is a 4-tuple $\langle T, G, \phi, F \rangle$ such that $T$ is a first-order theory, $G$ is a context-free grammar, $\phi$ is a first-order formula, and $F$ is a function symbol that may occur in $\phi$. A solution to a SyGuS problem $\langle T, G, \phi, F \rangle$ is either a function $f$ such that $T \vDash \phi[F \mapsto f]$ and $f \in \mathscr{L}(G)$, or proof that no

such function can exist.

SyGuS closely follows the syntax and semantics of SMT, and hence *T* usually refers to theories that are also common in SMT. Usually, SMT solvers are queried in the background of SyGuS solvers to verify solution candidates. This connection is made explicit in *Counter-Example Guided Inductive Synthesis* (CEGIS) [101]. CEGIS is a family of algorithms that alternate between a synthesis phase, which searches for a candidate solution that works for a subset of inputs, and a verification phase, where the candidate is checked against all possible inputs. If the verification fails, a counter-example is passed back to the synthesis phase and appended to the subset of inputs used to guide the search. The synthesis phase is often implemented as an enumerative search [5, 95, 3].

# Chapter 3

# Online Prompt Selection for Program Synthesis

The material in this chapter comes from paper *Online Prompt Selection for Program Synthesis* [60], published at *AAAI 2025*. I designed and implemented the algorithms, frameworks presented in this chapter, and also carried out the experimental evaluation.

## 3.1   Introduction

Large Language Models (LLMs) are beginning to dominate the discourse around program synthesis and code generation. So much so, that one might suppose they are the de facto answer to all code-generation questions. However, this is not the case. There are many synthesis problems in which LLMs still fall far short of the basic enumerative techniques and symbolic solvers [63, 13]. In addition, even when an LLM is the best choice, they still hold a significant barrier to entry for the inexperienced user: first, not all LLMs perform uniformly well across all problem sets, and it is often unclear which LLM a user should choose; second, the performance

of LLMs is often dependent on careful prompt engineering by expert researchers, with the literature reporting performance gains from many different prompting styles. Finally, compounding the challenge of these choices, calling LLMs is often expensive (in terms of computational cost, or the financial cost of using commercial APIs), and so making the wrong choice for a large set of synthesis tasks is highly undesirable. This chapter addresses these gaps through an online learning method that, given a synthesis task, will predict whether a symbolic solver or LLM, from a portfolio of LLMs, is most likely to solve the problem, with a corresponding prompting style.

We collate a portfolio of prompting styles and language models, which we combine into LLM-prompt pairs that we refer to as "solvers". We formulate the task of ranking the solvers in order of most likely to solve the problem as a multi-armed bandit problem [8]. The multi-armed bandit sequentially selects between choices (in our case, solvers) with unknown rewards (in our case, rewards are given for solving problems correctly and fast or with low computational cost). It trades off exploration, i.e., trying new solvers, with exploitation, i.e., using solvers that are known to be good. We also present a second variation of this formulation, with multiple layers of bandits. The top multi-armed bandit selects between the symbolic solver and the LLMs, and then the bandits in the lower layer predict the best prompt style for the chosen LLM.

We implement an instance of our approach, CYANEA, and evaluate it on synthesis tasks from the syntax-guided synthesis competition [4], from the literature on ranking function synthesis [33, 34], and generated from the SMT competition [85]. CYANEA solves 37.2% more synthesis queries than the best single LLM or solver, and gets within 4% of the virtual best solver.

## 3.2  Overview

### 3.2.1  Problem Statement

We hypothesise that program synthesis users will frequently have not just one but a series of synthesis problems to solve.  For instance, when synthesising invariants, one may be synthesising invariants for multiple different loops within the same code base or system under verification.  Users may also have different needs when it comes to performance (e.g., fastest, solves most queries, cheapest).

An example program synthesis problem, written in SyGuS-IF [83], is shown in Benchmark 1.  Given a candidate solution, we can validate whether this solution is correct or not using a Satisfiability Modulo Theories (SMT) solver, by checking if the formula

$$\exists x. \neg \phi(f)$$

is satisfiable (in which case the candidate $f$ is incorrect) or not.

```
1  (set-logic LIA)
2
3  (synth-fun fn0 ((vr0 Int) (vr1 Int) (vr2 Int)) Int
4      ((Start Int) (StartBool Bool) (Const Int)) (
5      (Start Int
6          (Start
7          Const
8          (- Start)
9          (+ Start Start)
10         (- Start Start)
11         (* Start Const)
12         (div Start Const)
13         (mod Start Const)
```

```
14          (abs Start)
15          (ite StartBool Start Start)
16          vr0 vr1 vr2))
17      (StartBool Bool
18          (StartBool
19          (> Start Start)
20          (= Start Start)
21          (>= Start Start)
22          (and StartBool StartBool)
23          (or StartBool StartBool)
24          (not StartBool)
25          true))
26      (Const Int (0 1)))) 
27 (declare-var vr0 Int)
28 (declare-var vr1 Int)
29 (declare-var vr2 Int)
30 (constraint (>= (fn0 vr0 vr1 vr2) vr0))
31 (constraint (>= (fn0 vr0 vr1 vr2) vr1))
32 (constraint (>= (fn0 vr0 vr1 vr2) vr2))
33 (constraint (or (= vr0 (fn0 vr0 vr1 vr2)) (or (= vr1 (
      fn0 vr0 vr1 vr2)) (= vr2 (fn0 vr0 vr1 vr2)))))
34
35 (check-synth)
```

Benchmark 1:  A SyGuS specification that asks for a program that synthesizes the maximum of 3 inputs.

Given a list of synthesis queries $Q = \{q_1, \dots q_m\}$, and a set of solvers $S = \{s_1, \dots s_n\}$, where each solver is either a symbolic solver, or an LLM paired with a prompting style (and LLM-prompt pair), we wish to use the solvers to generate a list of synthesis functions $f_1, \dots f_m$ such that each $f_i$ is a valid

solution to $q_i$, using as few computational resources as possible. We define computational resources to be both the time spent solving a query, and an estimate of the financial cost of running it (which is based on tokens used for the LLMs, or runtime for the symbolic solver).

## 3.2.2   Approach

We capture the computational resources we care about as reward functions. Our approach takes in $Q, S$, and a time budget, and cost budget per query, $T$, and $C$, respectively. For each synthesis query, our approach predicts an order of solvers that are most likely to solve the synthesis problem, and the time and cost that each is likely to take. We then distribute the total time and cost budget across the solvers accordingly, and deploy the solvers in sequence until the problem is solved.
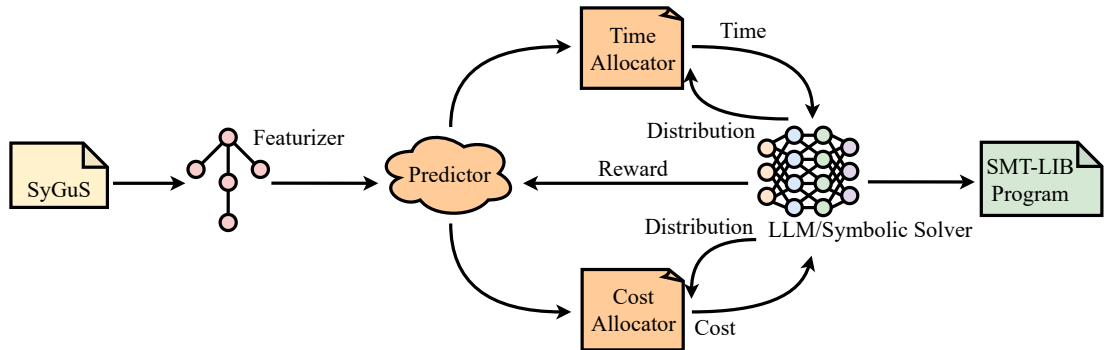


Figure 3.1: Single-layer Multi-Armed Bandit prediction
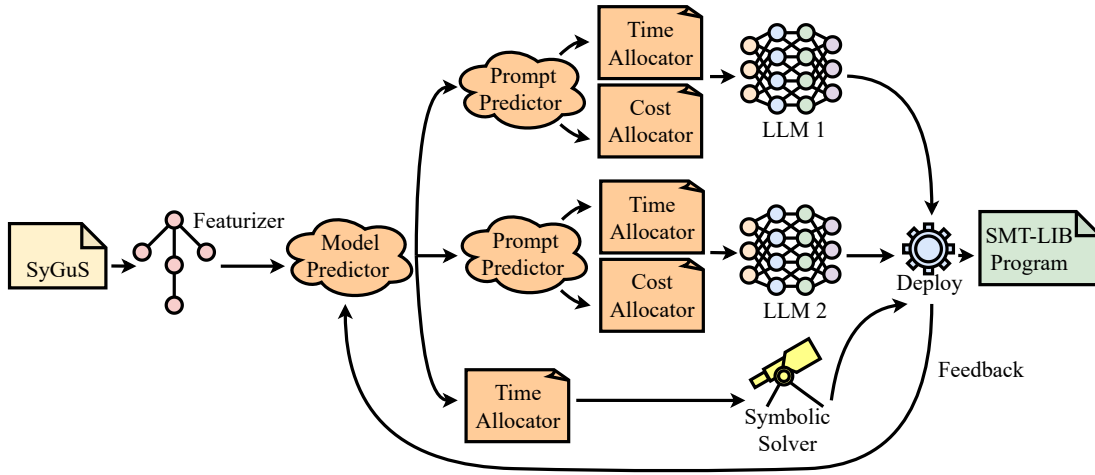
Figure 3.2: Multi-layer Multi-Armed Bandit prediction

After a problem is solved, data is passed back to update the predictors, in the form of any rewards obtained by the solvers called, and the solving time and cost.

We break down the task of predicting LLM-prompt pairs in two different ways. In the first, we implement a single multi-armed bandit that predicts between all LLM-prompt pairs at once, shown in Figure 3.1. In the second, we implement a multi-stage prediction, where we first predict which solver is most likely to succeed, and then predict which prompt strategy is most likely to succeed if an LLM is chosen, shown in Figure 3.2. The components are outlined as follows:

### *Featurize:*

The featurize block takes in a synthesis query $q$ in SyGuS-IF and generates a vector of features $\vec{f}$ representing that problem. We use a set of custom-designed features, which are outlined in Section 3.4.

### Multi-armed bandit solver and prompt predictor:

The multi-armed bandit component of our workflow takes in a feature vector that represents the synthesis problem, and predicts the order in which its library of solvers will perform, according to a cost function. In our implementation, the library of solvers consists of two LLMs, with 6 prompting styles, and an enumerative solver, detailed in Section 3.3.

We frame the problem as a contextual multi-armed bandit problem, where the actions that the agent is choosing are the solvers. We implement two variations of this: the first, shown in Figure 3.1, uses a single multi-armed bandit agent to rank a set of LLM/prompt combinations and a symbolic solver. The second, layered approach, uses several layered multi-armed bandit agents; one to choose between the base LLMs or symbolic solvers, and a further agent for each LLM, which predicts which prompt to deploy.

We give details of the contextual multi-armed bandit algorithms in Section 3.4. After a solver is deployed, the reward obtained by that solver is passed back to the solver performance predictor, which stores a list of rewards obtained so far by each solver.

### Time and token budgeting:

The final two phases of our pipeline allocate a certain amount of tokens to each LLM, followed by a number of time. Both are described in Section 3.4.

### Deploy solvers:

Finally, given a ranked list of solvers, and a time and token budget for each, the deploy phase sequentially calls each solver on the synthesis problem until it either returns an answer or exceeds the time or token budget. If a

solver returns the answer, we check if the answer satisfies the specification using an SMT-solver. We also return all information about which solvers successfully solved or did not solve the problem, the reward they obtained, and how long they took to the prediction phases of the pipeline.

## 3.3   Prompting Styles and Solvers

In this section, we give an overview of the library of solvers *S* that our approach is equipped with. First, we discuss the prompting styles:

### 3.3.1   Prompting Styles

We develop a library of prompt templates based on the prompting styles reported to be successful in the literature. We detail the styles here, and illustrate them on our running example. Once we have chosen a style, we give the LLM up to 16 attempts to produce a correct synthesis result. If an answer produced is incorrect, we report the error information obtained from the SMT-solver used to check correctness back to the LLM:

***Natural language prompts:***

LLMs are primarily trained on natural language inputs, and so we implement a simple syntactic transformation procedure that translates a set of logical constraints into natural language.

***Few-shot prompting:***

Few-shot prompting is prompting whereby the LLM is provided with a number of examples of the task, with satisfying solutions, before asking it to

solve a new, similar task. We use 3 examples, taken from the previously solved synthesis problems.

**Higher resource programming language prompts:**

Our synthesis queries are in SyGuS-IF, a relatively uncommon language in the training data for LLMs. Thus, we use a prompting style that asks for the solutions in a higher-resource language and then asks for the translation into SyGuS-IF. We choose Lisp as the higher resource language rather than a more common language like Python because we find that translation from Python to SyGuS-IF is more error-prone than translation from Lisp, which is a fully parenthesised prefix notation similar to SyGuS-IF. This is a multi-stage prompting approach, and the prompts are shown below. When asking for the translation into Lisp, we also provide 3 examples of previous translations:

```
Solve the following function ``function" with Lisp.
Only return one function, do not use recursion or
iterations. Do not return any text that isn't code.
Minimise token use. It's important you keep the
variables and function names the same as the original
function. The following is the problem that you are
meant to solve:
You need to synthesise: (synth-fun function ((x Int) (y
    Int) (z Int)) Int). The function is called ``solution
    " and takes arguments x, y, and z. These arguments
    are Int, Int, and Int.
Write only one Lisp-like method ``defun function" that
    never violates the SMT-LIB constraints.
No built-in functions in code.
Universally quantified variables: x, y, and z. The types
```

```
     of universally quantified variables are Int, Int,
       and Int.
The function must follow the constraints:
[constraints]
```

Prompt 1: Prompt for Benchmark 1 to generate Lisp response.

```
Please convert the Lisp function you generated into SMT-
    LIB format. Follow these guidelines:
Start the function with "(define-fun".
Provide only the function definition, starting with "(
    define-fun".
Ensure the SMT-LIB function contains exactly one
    function definition.
Avoid using iterations, bitvec, or int notations inside
    the body.
Check the function description in the first message to
    ensure variable and function names are consistent.
Use the assigned values from the Lisp code during
    translation.
Do not introduce any new variables that do not exist in
    the Lisp function.
Pay attention to types. If there are bit-vector terms,
    ensure they are of the same width.
Rules for SMT-LIB: +, -, *, ite, >, =, <, >=, <=, and,
    or, not, true, false.
```

Prompt 2: Prompt for Benchmark 1 to convert Lisp to SMT-LIB.

***Prompting with roles:***

Prefixing a prompt with an appropriate role description for the LLM can improve the performance of the LLM [115]. We append the sentence "You are a good program synthesiser" to the beginning of each prompt, if using "prompting with roles".

***Emotional stimuli:***

It has been shown in the literature that adding emotional stimuli to prompts can improve the performance of LLMs [58]. We append the following emotional stimuli to the prompt.

```
You are excited to help, and you are ready to provide
the best answer possible. You understand that if you
fail to provide the best answer, your client will be
extremely upset. Please don't fail me.
```

Prompt 3: Prompt for Benchmark 1 to add emotional stimuli.

***Matrix of prompts:***

In order to reduce the search space of prompts, we choose a fixed combination of prompting styles, shown in Table 3.1.

| | Natural Language | Higher Resource PL | Roles | Emotional Stimuli | Few-shot |
|---|---|---|---|---|---|
| Prompt Style 1 | ✓ | ✓ | ✗ | ✗ | ✗ |
| Prompt Style 2 | ✓ | ✓ | ✗ | ✗ | ✓ |
| Prompt Style 3 | ✗ | ✓ | ✗ | ✗ | ✗ |
| Prompt Style 4 | ✗ | ✗ | ✗ | ✗ | ✗ |
| Prompt Style 5 | ✗ | ✓ | ✓ | ✗ | ✗ |
| Prompt Style 6 | ✓ | ✓ | ✓ | ✓ | ✗ |

Table 3.1: Prompt styles

### 3.3.2 Enumerative Solver

The final solver in the library is an enumerative solver, based on Counter-Example Guided Inductive Synthesis (CEGIS) [101], with an $A^*$ based search phase. Details about $A^*$ search will be discussed in Chapter 4. CEGIS alternates between a synthesis phase, which searches for a candidate solution that works for a subset of inputs, and a verification phase, where the candidate is checked against all possible inputs. If the verification fails, a counterexample is passed back to the synthesis phase and appended to the subset of inputs used to guide the search. In our case, the synthesis phase is implemented as an $A^*$ search, similar to that used by Euphony [57].

    $A^*$ is a graph search algorithm that uses two functions to guide its search: $f$: the sum of the costs on the edges used to reach the current state, and $g$: the estimated sum of the costs on the edges that will be used to reach a target state from the current state. In our setting, each state is an expression

(a partial or complete program) that can be generated from the grammar for the full logic, the initial state is start symbol of the grammar, the target states are any complete program, and each edge between states $s_i$ and $s_k$ corresponds to a production rule that can transform the partial program at $s_i$ into the partial/complete program at $s_j$. The cost on any edge is proportional to the number of possible choices (so the more edges there are leaving from one state, the higher the cost of each edge).

To give some intuition, a partial program that contains few non-terminals and where each non-terminal symbol can only be replaced by production rules that lead immediately to a complete program has a low estimated cost to reach the target. We refer the reader to the detailed descriptions in the related work [57, 63] for the full details. We chose this implementation of CEGIS as the enumerative solver to use because, in our experiments, it excels at finding short solutions that the LLMs often struggle with, without running into the memory issues that often plague bottom-up search methods in synthesis.

## 3.4   Online Solver Selection

The aim of the multi-armed bandit is to predict a ranking of which LLM and prompt combinations are most likely to solve the synthesis problem, and obtain the maximum reward while doing so. In our setting, the agent must trade off the exploration of using LLMs and prompts that it has not tried before, vs deploying LLM and prompt combinations that are known to have given high rewards in the past. In fact, we use an extension of the standard MAB problem, and ask the agent to predict a sequence of solvers to deploy rather than a single solver.

### 3.4.1　$k$-**Nearest Neighbor**

We choose $k$-Nearest Neighbour as our contextual multi-armed bandit. Other contextual multi-armed bandits are available, but many of the common ones, for instance LinUCB, make assumptions that the performance of solvers is correlated linearly with the feature vector, which is unlikely to be the case in our application.

$k$-NN is a simple supervised learning classifier. In our context, given a synthesis query $q$, it identifies the nearest $k$ previously solved queries to $q$ by calculating the Cartesian distance between the feature vectors. Each of the $k$ queries $q_1, ... q_k$ is labelled with the solver that it was solved by and the reward that was obtained, $r_1, ... r_k$ respectively. The score for a solver $s_i$ is given by the sum of the rewards for all queries solved by $s_i$. We rank the solvers based on this score (the highest score is best). For any solvers that do not appear in this ranking, we randomly shuffle them and append them to the end of the list of solvers. If an LLM-prompt pair solves a query, we add a query with that feature vector to our database of queries with the corresponding reward.

For the double-layered multi-armed bandit, the first layer contains one $k$-NN multi-armed bandit which selects only between the LLMs, and the second layer contains a $k$-NN multi-armed bandit for each LLM, which selects between prompts. The second layer $k$-NN predictors are independent.

***Reward functions:***

Our approach is customizable to different reward functions We use three reward functions: the first simply aims to solve the queries as fast as possible, regardless of computational cost; and the second takes computa-

tional cost into account. The first reward function is given as follows:

$$
r^t = \begin{cases} 0 & \text{if query } q \text{ is unsolved,} \\ \left(1 - \frac{t}{T}\right)^4 & \text{if query } q \text{ is solved} \end{cases}
$$

where $t$ is the time taken to solve query $q$, and $T$ is the total time budget for solving query $q$.

The second reward function aims to prioritise cheaper solving, and so accounts for the number of tokens in the prompt and response.

$$
r^c = \begin{cases} 0 & \text{if query } q \text{ is unsolved,} \\ \left(1 - \frac{c}{C}\right)^4 & \text{if query } q \text{ is solved} \end{cases}
$$

where $c$ is a cost estimate proportional to the number of tokens used in solving query $q$ and $C$ is the total cost budget for solving query $q$. The cost estimate is defined as

$$
c = \text{input tokens} + 3 \times \text{output tokens}
$$

for LLMs, which accounts for the higher cost of output tokens from commercial language model APIs. The actual cost of deploying an enumerative solver is proportional to the runtime and would be negligible for all queries in comparison to the cost of calling a commercial language model, so we fix the cost for the enumerative solver to be a small constant (0.4) for all queries.

The final reward is a simple binary reward, $r^b$, which evaluates to 1 if a query is solved and 0 if it is not solved.

### *Features:*

A key component of the contextual multi-armed bandit is the featurization. We propose a feature extraction method to analyse SyGuS queries, capturing key syntactic attributes and query types. The extracted features we use are:

**Keywords:** Frequencies of specific SMT-LIB keywords (e.g., `+`, `−`, `*`, `div`, etc).

**Query length:** The total number of tokens in the file.

**Constants:** Number of constants of each type.

**Query logic:** e.g., `BV, LIA, PBE, INV`, etc.

## 3.4.2 Time and Token Budget Allocation

The final stage of the dynamic solver selection predicts the time that should be allocated to solver, and the cost. The goal is to allocate a sufficient proportion of our time and cost budget to each solver in the series that we are reasonably confident that it was unlikely to solve the query past this point. That is, for a solver $s_i$, we wish to find a minimum time allocation $t_i$ and cost allocation $c_i$ such that $P(t_i < u_i < T) \leq \delta_1$ and $P(c_i < v_i < C) \leq \delta_2$, where $u_i$ is the true runtime, $v_i$ is the true cost, and $\delta_1$ and $\delta_2$ are some small error thresholds. $\delta_1$ is the probability that we failed to solve a query because we allocated too little time to solving it, and $\delta_2$ is the probability that we failed to solve a query because we allocated too few tokens to it.

Let us consider the cost allocation first: we model each prompt-pair's cost per query as an exponential distribution (that is, most queries are solved with a small number of tokens, only a few queries are solved with an excessively large number of tokens). We use maximum likelihood estimation (MLE) [75] to estimate the parameters of the underlying exponential distribution, given the costs we have observed so far. Suppose we observe $u_1 \ldots u_n$ costs, which we assume are drawn from an exponential distribution $Exponential(\lambda)$. To find the exponential distribution which fits our observations best, we aim to solve

$$\min_{\lambda} n \ln \lambda - \lambda (\Sigma_i u_i),$$

where $\Sigma_i u_i$ is the sum of all costs observed so far. This gives us the minimiser

$$\lambda^* = \frac{n}{\sum_i u_i}.$$

We can apply the cumulative distribution function and calculate $c_i$ as:

$$c_i = \frac{-\ln(\delta + e^{-\lambda^* c})}{\lambda^*}.$$

To make this contextual, we use only the costs from the $k$ nearest samples, according to the feature vectors. We divide the total token budget $C$ greedily between the solvers, calculate $c_i$ for each solver starting from the beginning of our ranking, and, once we have reached the total budget $C$, all following solvers are allocated zero tokens. If we reach the end of the list and have remaining tokens, they are given to the final solver.

We repeat all of the above for a time. It is worth noting that the time budget is not independent from the cost budget, because if a solver is allocated zero tokens by the cost budget allocator, the time budget allocator will also not allocate it any time.

## 3.5  Evaluation

We implement an instance of our approach, called CYANEA, using two LLMs: GPT(gpt-3.5-turbo-0125) and Llama(Meta-Llama-3-70B) and the enumerative solver described previously. We set a total timeout of $T = 100$ seconds and a total cost budget of $C = 100,000$. For all $k$-NN predictors, we set $k = 15$. We conducted a parameter sweep of $k$ and found that values between 10 and 15 produce comparable results. We use CVC5 [12] as the SMT solver for validating the correctness of candidate solutions. We compare our approach to the base solvers and to a "virtual best" solver result, which is calculated by choosing the solver known to give the highest reward for

each query. To evaluate the utility of the time and budget allocations, we compare to a linear distribution of the time and cost budget (where we simply divide the total budget equally between all solvers), termed "linear" in the results.

## 3.5.1   Synthesis Queries and Scoring.

We evaluate our approach on synthesis queries from the Syntax-Guided Synthesis competition [4], ranking function synthesis [33], and automatically generated from the SMT competition [85]. The synthesis queries cover a broad range of use-cases of synthesis, from code generation and programming-by-example, to ranking function and invariant synthesis. The total number of queries is 1269.

We report the total reward achieved, calculated using the reward functions used for the prediction. We also report the score according to the Par-2 score used by the SAT competition [107]. This is calculated over $n$ queries:

$$\sum_{j=1}^{j=n} \begin{cases} t_i & \text{if } q_i \text{ is solved,} \\ 2 * T & \text{otherwise} \end{cases}$$

where $t_i$ is the runtime for solving query $q_i$, and $T$ is the total time budget per query.

## 3.5.2   Analysis of Results

The performance of the LLM-prompt pairs and enumerative solver is shown in Table 3.2. The best-performing single solver solves 64.3% of the queries. On the other hand, the virtual best solver solves 91.8% of queries. The best-performing instance of CYANEA solves 88.3% of the queries, achieving a score of 96.1% of the virtual best solver, and a Par-2 score < 40% of

| Solver | % Solved | # Solved ($r^b$) | Par-2 Score | $r^c$ | $r^t$ | avg time (s) | avg. cost |
|---|---|---|---|---|---|---|---|
| Virtual Best | 91.8% | 1165 | 23596 | 1106.2 | 1019.2 | 2.4 | 670.3 |
| Single $k$-NN ($r^c$) | 88.3% | 1120.6±7.3 | 37636.3 | 1008.7 | 904.4 | 7.1 | 3122 |
| Single $k$-NN ($r^t$) | 88.2% | 1119.1±8.2 | 37813.7 | 1006 | 905.7 | 7 | 3176.9 |
| Single $k$-NN ($r^b$) | 88.1% | 1117.5±8.3 | 38793 | 995.2 | 888 | 7.6 | 3446.5 |
| Single $k$-NN linear ($r^t$) | 87.0% | 1104±0 | 39072 | 1000.9 | 935.2 | 5.5 | 3249.6 |
| Single $k$-NN linear ($r^c$) | 87.0% | 1104±0 | 39072 | 1002.1 | 935.2 | 5.5 | 3211.4 |
| Single $k$-NN linear ($r^b$) | 87.0% | 1104±0 | 39182.4 | 995.4 | 930.4 | 5.6 | 3394.9 |
| Double $k$-NN ($r^c$) | 84.5% | 1071.7±24 | 53499.3 | 886.9 | 774.7 | 13.1 | 6366.8 |
| Double $k$-NN ($r^t$) | 84.4% | 1071.1±24.7 | 53504.3 | 884.6 | 776.1 | 13 | 6448.3 |
| Double $k$-NN ($r^b$) | 84.3% | 1069.8±24.5 | 53747.4 | 887.8 | 775.2 | 13 | 6262.7 |
| Double $k$-NN linear ($r^c$) | 71.8% | 910.7±159.2 | 80038.4 | 803.2 | 732.4 | 9.2 | 4589.1 |
| Double $k$-NN linear ($r^t$) | 71.8% | 910.7±159.2 | 80129.5 | 801.9 | 731.8 | 9.3 | 4656.1 |
| Double $k$-NN linear ($r^b$) | 71.8% | 910.7±159.2 | 80220.6 | 798.2 | 728.9 | 9.4 | 4809.2 |
| llama-p4 | 64.3% | 816 | 95251.2 | 752.3 | 664.2 | 5.7 | 2098.2 |
| llama-p3 | 61.7% | 783 | 106596 | 678.1 | 529.1 | 12 | 3794.1 |
| llama-p5 | 59.7% | 757 | 112543.8 | 644 | 490.6 | 13.4 | 4275.9 |
| gpt-p4 | 54.3% | 689 | 118273.7 | 637.1 | 607.4 | 3.3 | 2070.1 |
| enumerator | 52.2% | 662 | 122591.6 | 647.3 | 626.7 | 1.8 | 0.4 |
| gpt-p1 | 51.6% | 655 | 126533.5 | 591 | 531.6 | 5.7 | 2678.1 |
| gpt-p6 | 50.5% | 641 | 129638.3 | 582.8 | 513.4 | 6.3 | 2508.7 |
| gpt-p5 | 44.1% | 560 | 145160 | 510.4 | 453.1 | 6 | 2425 |
| gpt-p3 | 44.0% | 558 | 145101.6 | 502.3 | 464.1 | 5.2 | 2900.5 |
| gpt-p2 | 39.2% | 497 | 156238.9 | 428.4 | 431.7 | 3.7 | 3755.7 |
| llama-p1 | 37.2% | 472 | 162184.8 | 436.5 | 380.5 | 5.9 | 2008.6 |
| llama-p2 | 35.0% | 444 | 168108 | 361.6 | 341.5 | 7 | 5119.5 |
| llama-p6 | 34.8% | 442 | 167963.6 | 407.3 | 353.8 | 5.8 | 2071.6 |

Table 3.2: Performance of all instances of CYANEA, and all individual solvers. We report results from CYANEA over 20 runs, with the standard deviation shown for the number of queries solved. The "Virtual Best" solver reports the maximum scores we could achieve if we made the perfect choice for each query, using the best reward function for that score (e.g., the score for $r^c$ is reported, making choices using $r^c$.

the score of the best single solver (lower is better for Par-2 score), shown in Figure 3.3.
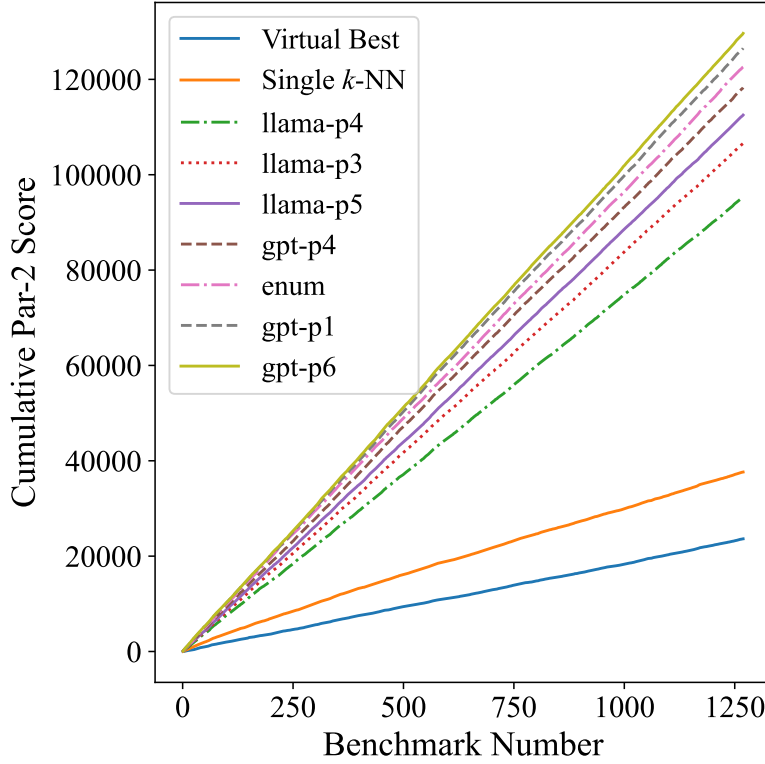


Figure 3.3: Cumulative Par-2 Score plotted against query number. Lower is better.

Since the order the selectors see the queries will affect the learning, we report the average scores of 20 runs, with the queries randomly shuffled in each run. When comparing the single to multi-layer approaches, we note that the multi-layer approach has a far greater standard deviation in the number of queries solved. We hypothesise that this is because the selectors in the lower layers see far less data than in the single-layer $k$-NN, making them more sensitive to this ordering.

When evaluating the reward functions used by a single $k$-NN, unsurprisingly, the best Par-2 score (which is based on solving time) is achieved using $r^t$, the lowest cost per query is achieved using $r^c$, and the highest

number of queries solved is achieved using $r^b$. This is not true for double $k$-NN, again we believe because the data is too sparse.

The results for CYANEA using the linear distribution of time and token budgets demonstrate that the token budget allocator is having an impact on both Par-2 score and average solving cost, although this effect is not as large as it would be if the total time budget and cost budget were tighter, as in many cases CYANEA can run all the solvers on a single query within the given budget. Whilst the double $k$-NN with the linear distribution of budgets does have lower average costs per query, we hypothesise that this is because it fails to solve many of the queries that require a higher cost to solve.

Overall, the predictions that the best-performing instance of CYANEA is making are close to those of the virtual best solver and, when the prediction is not perfect, the time/budget allocation allows CYANEA to correct the mistake.

### 3.5.3 Conclusions

We have presented an approach for online solver and prompt selection for program synthesis problems; CYANEA demonstrates the effectiveness of this, achieving a Par-2 score that is more than twice as good as the best single solver. It also demonstrated how prompt selection can make large language models more reliable for program synthesis when multiple models and prompting strategies are available. We will turn to a complementary question in next chapter: how can we embed LLMs more tightly into the synthesis loop itself?

# Chapter 4

# Guiding Enumerative Program Synthesis with LLMs

The material in this chapter comes from paper *Guiding Enumerative Program Synthesis with Large Language Models* [63], published at *CAV 2024*. I designed the frameworks, in collaboration with Professor Elizabeth Polgreen, implemented the algorithms and carried out the full experimental evaluation.

## 4.1   Introduction

The dominant techniques for formal program synthesis are based around enumeration [95, 6, 42], and a key challenge is how to guide this enumeration to search a huge space of possible programs efficiently. Syntax-Guided Synthesis(SyGuS) [3] allows the user to restrict the space of possible programs using a context-free grammar, and, in later work, this has been extended using pre-trained probabilistic models such as higher-order grammars [57] and neural networks [74], trained on a dataset of solved synthesis problems. However, obtaining these datasets for pre-training is

challenging.

In parallel, the use of pre-trained large language models (LLMs) to generate code is rapidly gaining traction, with impressive results being obtained on benchmarks with natural language specifications and input output examples [24]. These benchmarks are very different in style to the logical specifications that formal program synthesis tackles, as most are procedural code, in Python, and solve classic programming exercise questions that might be asked of students or interview candidates, and that one may find in abundance on sources used in training data, such as Stack-Overflow and GitHub. In contrast, formal program synthesis benchmarks, such as those in the SyGuS competition, require functional code, which must satisfy precise logical specifications derived from problems such as program analysis [27], and are certainly less abundant in sources of publicly available code for training machine learning models.

In this chapter, we set out to investigate whether off-the-shelf large language models can solve formal program synthesis problems. We craft a library of prompts, which enables us to solve roughly 50% of the SyGuS competition benchmarks. We hypothesise that, in the cases where the LLM returns only incorrect solutions, the correct solutions are most often in the vicinity of the incorrect solutions, and that, by searching in the neighbourhood of the incorrect solutions, we may be able to guide an enumerative synthesiser to find a solution faster. To that end, we construct a probabilistic Context-Free Grammar (pCFG) based on the incorrect solutions proposed by the LLM, and use this to guide an enumerative synthesiser within a Counter-Example Guided Inductive Synthesis (CEGIS) loop.

Our final contribution is a full integration of these techniques in a novel CEGIS algorithm with an inline syntactic oracle, in the form of an LLM that is queried by an enumerative synthesis phase. We incorporate informa-

tion obtained during the synthesis search into the queries, prompting the LLM with partially enumerated functions, incorrect solutions, and counter-examples, and requesting that it provide "helper functions", which we use to update the pCFG guiding the enumerator.

We implement all three techniques described above and evaluate them on benchmarks from the Syntax-Guided Synthesis competition. We compare with two baselines: the first is an enumerative synthesiser where all rules in the grammar are given equal likelihood, and the second is CVC5 [12], the state-of-the-art SyGuS solver. All techniques easily outperform the baseline enumerator, and the final technique outperforms CVC5. Our results demonstrate that, whilst large language models do have the potential to make significant contributions in the domain of formal program synthesis, this can currently only be achieved by combining these techniques with existing algorithms in the literature. Enumerative synthesis is not yet obsolete!

The main contributions of our work are as follows: A set of prompts for prompting a pre-trained Large Language Model to solve formal program synthesis problems (Section 4.3.1); A method for guiding an enumerative synthesiser using LLM-generated probabilistic context-free grammars (Section 4.4.1); A novel approach to integrating an LLM into an enumerative synthesiser (Section 4.5); And, finally, an implementation and evaluation of all of the above on benchmark problems taken from the Syntax-Guided Synthesis competition. The results outperform CVC5, the state-of-the-art synthesiser, as well as our baseline enumerators.

## 4.2  Overview

In this chapter, we first present a carefully tailored set of prompts that we use to evaluate an LLM's ability to solve formal synthesis problems.  We construct an iterative loop where we prompt the LLM, verify the candidate solution, and if the solution fails, we prompt the LLM again.

We then present two methods for integrating syntactic guidance from pre-trained LLMs into an enumerative CEGIS algorithm.  The first method, shown in Figure 4.1, prompts an LLM for solutions to the benchmark, and generates a pCFG from these solutions before deploying an enumerative synthesiser, increasing the chance of the LLM solving the synthesis problem outright. We refer to this method as pCFG-synth.
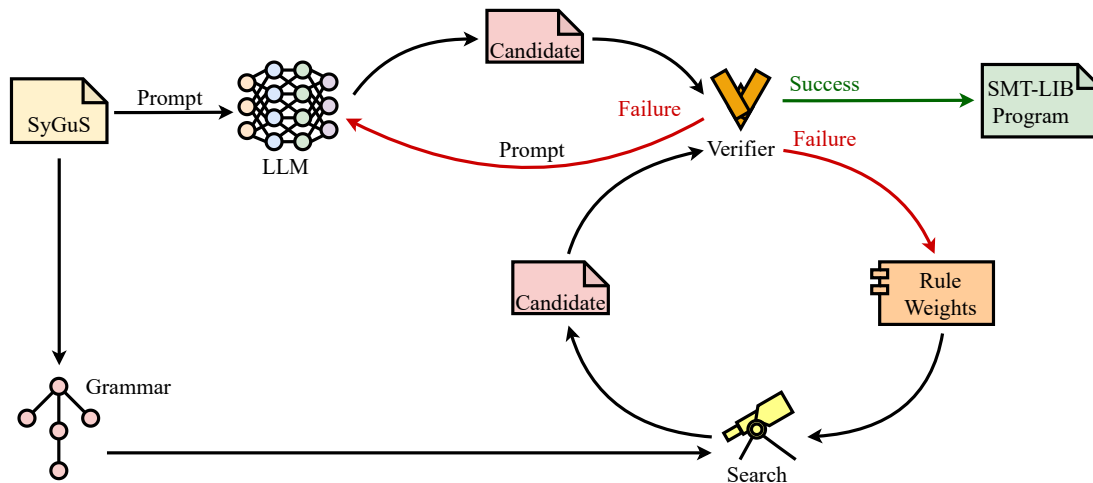


Figure 4.1: An overview of pCFG-synth. Both the verifier and the LLM have access to the specification $\phi$ (which is used to generate the prompt for the LLM, as well as to check whether candidate programs are correct).
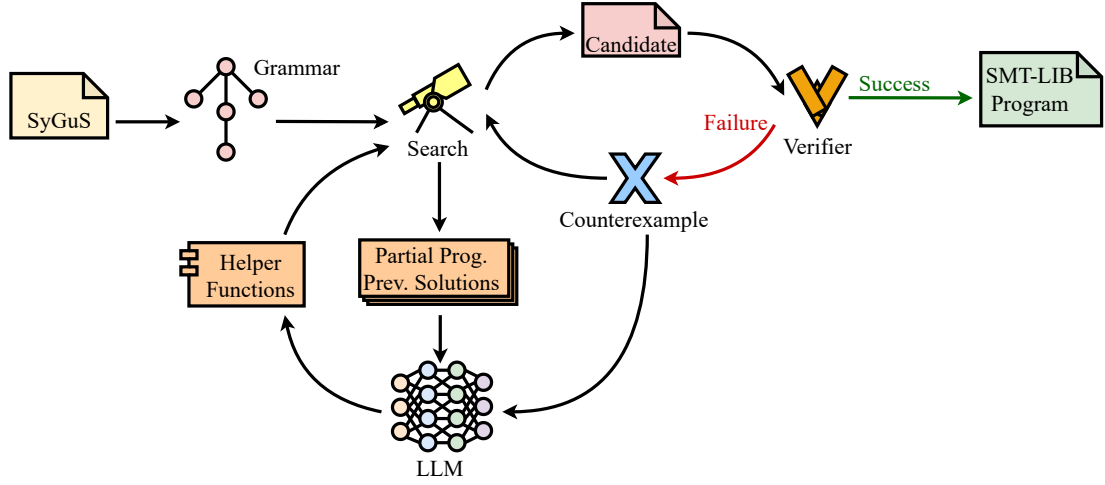
Figure 4.2: An overview of iLLM-synth. Both the verifier and the enumerator have access to the specification $\phi$ (which is used to generate the prompt for the LLM, as well as to check whether candidate programs are correct).

The second method, shown in Figure 4.2, integrates the prompting within the enumerative synthesiser, allowing the prompts to incorporate additional information obtained during the synthesis process. Here, instead of asking the LLM to provide a full solution, we ask it to provide helper functions to help "a student" complete the partially enumerated program. We use the responses to augment the set of production rules in the grammar and update the weights across the existing production rules. We refer to this approach, which integrates an LLM into an enumerative synthesiser, as iLLM-synth. In this section, we give an overview of these two approaches. The details of the components of both approaches and their relative performances are found in the subsequent sections. We integrate both approaches with a probabilistic top-down enumerator and a weighted search based on the $A^*$ algorithm [39, 57].

## 4.3   Stand-alone LLM

In this section, we describe how we prompt the LLM as a stand-alone synthesiser. These prompting techniques are then also deployed by pCFG-synth. We use GPT-3.5-turbo as the LLM. Note that the model is not fine-tuned to this problem setting. Furthermore, we rename any functions and variables in the SyGuS benchmarks to generic names to avoid the LLM producing solutions solely based on the function names.

### 4.3.1   Prompting the LLM

We design a library of prompts for program synthesis problems with logical specifications and a single target function to synthesise. These prompts are deployed in an iterative loop, until a correct solution is obtained, or the library of prompts is exhausted.

Prompting is an art rather than a science, but we hypothesise that it is better to ask the LLM to give a solution in a language that is more common in the training data, and then request it to translate it into our desired SMT-LIB, and experiment with both Python and Lisp. On a subset of 50 benchmarks, we observed that soliciting responses in Lisp resulted in a 6% enhancement in the resolution of benchmarks compared to utilising Python (and both were better than asking for SMT-lib directly). It is also reported in the literature that the efficacy of LLMs can be augmented by including emotional stimuli within prompts [58]. Incorporating the emotional prompt leads to an 8% increase in the likelihood of generating accurate solutions compared to standard, non-emotional prompts in our study.

The following is an example prompt and response sequence for the LIA example shown in Benchmark 1:

```
(set-logic LIA)
(synth-fun function ((x Int) (y Int) (z Int)) Int)
(constraint (>= (function x y z) x))
(constraint (>= (function x y z) y))
(constraint (>= (function x y z) z))
(constraint (or (= x (function x y z)) (or (= y (
    function x y z)) (= z (function x y z)))))
(check-synth)
```
You are a good synthesiser. Do you know what "(define-
    fun function ((x Int) (y Int) (z Int)) Int" is doing?
Write only one Lisp-like method "defun function" 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 fn" that never
    violates the SMT-LIB constraints above.

Prompt 4: Prompt for Benchmark 1.

```
(defun function (x y z)
    (ite (>= x y) (ite (>= x z) x z) (ite (>= y vr2) y z
        )))
```

Response 1: Response from LLM after Prompt 4.

```
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.
```

Prompt 5: Prompt requesting a revised solution.

## 4.3.2  Prompts for invariant synthesis

Invariant synthesis is a specific instance of program synthesis: given a pre-condition $pre(x)$, transition-relation $trans(x, x')$ and post-condition $post(x)$, the synthesiser is required to provide an invariant $inv$ that satisfies the following constraint:

$$\forall x, x'.pre(x) \implies inv(x) \land (inv(x) \land trans(x, x)) \implies inv(x') \land inv(x) \implies post(x).$$

We find that LLMs struggle to reason about constraints presented in the above format. Inspired by "chain-of-thought" [109] prompting, where the LLM is asked to provide a step-by-step explanation, we augment our prompting strategy for invariants by asking the LLM first to explain the constraints. After requesting this explanation, we follow the same interactive prompt strategy as before.

```
(synth-inv inv-f ((x Int) (y Int)))
(define-fun pre-f ((x Int) (y Int)) Bool (and (= x 1) (=
    y 1)))
(define-fun trans-f ((x Int) (y Int) (x! Int) (y! Int))
    Bool (and (= x! (+ x y)) (= y! (+ x y))))
(define-fun post-f ((x Int) (y Int)) Bool (>= y 1))
(inv-constraint inv-f pre-f trans-f post-f)
Please explain the constraints above.
```

Prompt 6: Integrating LLM-generated explanations into the prompt

### 4.3.3 Lisp to SMT-LIB Converter

The final prompts in our prompt library are to ask the LLM to convert any functions given in Lisp to correct SMT-LIB functions:

```
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.
```

Prompt 7: Request for converting Lisp to SMT-LIB code for response 1.

Upon receiving a response from the LLM, we extracted the Lisp program

and subjected it to format verification. The resulting SMT-LIB code is represented in Response 2:

```
(define-fun function ((x Int) (y Int) (z Int)) Int
    (ite (>= x y) (ite (>= x z) x z) (ite (>= y vr2) y z
        )))
```

Response 2: Response from LLM after Prompt 7.

---

**Algorithm 1** CEGIS with weighted search
---

 1: **procedure** CEGIS($W_G, \phi$)

 2:      $cex \leftarrow \varnothing$

 3:      **while** $true$ **do**

 4:          $prog \leftarrow$ ENUMERATE($W_G, \phi, cex,$ )

 5:          **if** VERIFY($prog, \phi$) **then**

 6:              **return** $prog$

 7:          **else**

 8:              $c \leftarrow$ VERIFY.GET_CEX

 9:              $cex \leftarrow cex \cup \{c\}$

10:          **end if**

11:      **end while**

12: **end procedure**

---

## 4.4   Synthesis with pCFG Guidance: pCFG-synth

We hypothesise that, if the LLM did not propose a correct solution, the correct solution is likely to be roughly in the same "area" as the incorrect solutions it suggested, and so our synthesis algorithm aims to prioritise this area when searching for candidate programs. For simplicity, we use a simple weighted Context-Free Grammar to represent the area of solu-

tions proposed by the LLM. We then present methods for searching the space: the first is a probabilistic top-down search, shown in Algorithm 4; the second is based on an adaptation of the *A** algorithm [39, 57], and we integrate both into CEGIS searches as shown in Algorithm 1. The verification phase in Algorithm 1 is implemented via a call to an SMT solver, which checks, for a candidate solution $f$, whether there exists an input such that the specification is violated, i.e.,

$$\exists x. \neg \phi[F \mapsto f]$$

### 4.4.1 Inferring a Weighted CFG

In this section, we describe how we infer a weighted Context-Free Grammar from the incorrect solutions produced by the large language model.

**Definition 1** (Derivations). *Given a context-free grammar G, and a sentence s, the sentence is in the language of the grammar if $S \to^* s$, where S is the start symbol of the grammar. The derivation of s from S is a sequence of rules such that $S \xrightarrow{r_0} s_1 \xrightarrow{r_1} \dots s_n \xrightarrow{r_n} s$ and $r_0 \dots r_n \in R$. We denote the derivation of s by the sequence of rules $r_0, \dots r_n$ as $D_s = \{r_0, \dots r_n\}$. The left-most derivation is a derivation such that all rules expand the left-most non-terminal symbol in the sentential form.*

From here on in, all derivations are assumed to be the left-most derivation, and we assume the grammar is unambiguous, i.e., there exists a single left-most derivation for any sentence in the language.

Given a set of possible programs $prog \in \mathscr{L}G$ generated by the language model, we calculate a weight for each rule $r_i \in R$ as the number of times that rule appears in the left-most derivations of the programs. That is,

$$w[r_i] = \sum_{prog_i \in prog} |r_i| \in D_{prog_i}, \qquad (4.1)$$

where $|r_i|$ is the number of times $r_i$ appears in the derivation. For example, consider Response 1: the weights are calculated as

$$w[r_1] = 3,$$
$$w[r_2] = 3,$$
$$w[r_3] = 3,$$
$$w[r_4] = 4,$$
$$w[r_5] = 3.$$

These correspond to the rules from Benchmark 1:

$r_1$ : `Start → (ite StartBool Start Start)`

$r_2$ : `Start → vr0`

$r_3$ : `Start → vr1`

$r_4$ : `Start → vr2`

$r_5$ : `StartBool → (>= Start Start)`.

### *Probabilistic context-free grammar:*

Given a wCFG, we derive a simple pCFG by assuming that the probability associated with a rule $r_i: \alpha \to \beta$ is equal to the weight $w[\alpha \to \beta]$ of $r_i$, divided by

$$|\pi[\alpha]| = |\alpha \times (\Sigma \cup V)^* \in R|,$$

i.e., the total number of rules that could be applied to $\alpha$. That is

$$\mathbb{P}[\alpha \to \beta] = \frac{w[\alpha \to \beta]}{|\pi[\alpha]|}.$$

By extension,

$$\mathbb{P}_\Sigma[\alpha \to \beta] = \frac{w[\alpha \to \beta]}{|\pi[\alpha]}|$$

iff $\beta \in \Sigma$ and 0 otherwise.

## 4.4.2 Probabilistic Guided Search

The aim of our algorithm is thus to search the area of programs closest to those with the highest weights in the wCFG, or highest probabilities in the corresponding pCFG. We adapt and implement two search methods for doing this: the first is a probabilistic top-down search. To this end, we first introduce the notion of a grammar tree.

**Definition 2** (Grammar tree). *We represent the search space as a grammar tree. Given a context-free grammar $G = (V, \Sigma, R, S)$, the graph of sentential forms, or grammar tree, $\mathcal{T}(G)$ defined inductively: S is the root of the tree, and for all $x, y \in (V \cup \Sigma)^*$ with $x \to y$ and $x$ being a node of the tree, then $y$ is a child node of $x$.*

To implement our probabilistic guided search, we extend this definition to a probabilistic grammar tree. Given a pCFG, $P_G = (V, \Sigma, R, S, \mathbb{P})$, a probabilistic grammar tree $\mathcal{T}(P_G)$ is a directed labelled graph as defined before, but each edge has a corresponding weight $\omega$ given by $\mathbb{P}$. We limit the edges to only those needed for the left-most derivations, and so $\mathcal{E}$ and $\omega$ are defined as follows:

$$\mathcal{E} = \{xay \xrightarrow{a \to \beta} x\beta y \mid a \to \beta \in R, x \in \Sigma^*, a \in V, \beta, y \in (V \cup \Sigma)^*\},$$

$$\omega[a \to \beta] = \mathbb{P}[a \to \beta].$$

Note that this guarantees that, for any node, the sum of the weights on the edges leaving that node is equal to 1.

---

**Algorithm 2** Probabilistic top-down enumerator for pCFG-synth

---

1: **procedure** ENUMERATE($W_G, \phi, cex$ )

2:　　　$prog \leftarrow W_G.S$

3:　　　$d \leftarrow 0$

4:　　　$previousProgs \leftarrow \varnothing$

5:　　　$P_G \leftarrow$ BUILDPCFG($W_G$)

6:　　　**while** 1 **do**

7:　　　　　**if** $prog \in \Sigma^*$ **then**

8:　　　　　　　$previousProgs \leftarrow previousProgs \cup prog$

9:　　　　　　　**if** $\forall \vec{x} \in cex. \phi(prog, \vec{x})$ **then**

10:　　　　　　　　　**return** $prog$

11:　　　　　　　**else**

12:　　　　　　　　　$prog \leftarrow S$

13:　　　　　　　　　$d \leftarrow 0$

14:　　　　　　　**end if**

15:　　　　　**end if**

16:　　　　　$prog \leftarrow$ REPLACENONTERMINALS($prog, P_G$)

17:　　　　　$d \leftarrow d + 1$

18:　　　　　**if** $d = maxDepth$ **then**

19:　　　　　　　$prog \leftarrow$ COMPLETEPROGRAM($prog, P_G$)

20:　　　　　　　**if** $prog \in PreviousPrograms$ **then**

21:　　　　　　　　　$prog \leftarrow S$

22:　　　　　　　　　$d \leftarrow 0$

23:　　　　　　　**end if**

24:　　　　　**end if**

25:　　　**end while**

26: **end procedure**

---

We search this grammar tree using a top-down enumerative synthes-

iser, shown in Algorithm 2. This enumerates possible programs in the grammar in a top-down manner, expanding non-terminals by randomly sampling from the categorical distribution over the production rules. That is, the search algorithm starts by considering the node corresponding to the start symbol $S$. It then chooses the next node by sampling from a categorical distribution with event probabilities corresponding to the probabilities on the outgoing edges of the current node. The categorical distribution is a generalisation of the Bernoulli distribution and describes the possible results of a random variable that can take one of $K$ possible categories, with the probability of each category separately specified. Formally, to sample a rule $\alpha \times \beta$ to apply to a non-terminal symbol $\alpha$, we sample from the distribution:

$$(\alpha \times \beta) \sim Cat(|\pi[\alpha]|, \{\mathbb{P}[\pi[\alpha]_1], \mathbb{P}[\pi[\alpha]_2], ...\}),$$

where $|\pi[\alpha]|$ is the number of rules that could be applied to $\alpha$ and $\pi[\alpha]_i$ is the $i^{th}$ of those rules, and $\{\mathbb{P}[\pi[\alpha]_1], \mathbb{P}[\pi[\alpha]_2], ...\}$ is a vector of probabilities corresponding to those rules.

We then apply the sampled rule, and repeat the process. We use $prog.\{\alpha \rightarrow \beta\}$ to indicate the result of substituting the first occurrence of $\alpha$ in a partial program $prog$ with $\beta$.

---

**Algorithm 3** Replace non-terminals and complete program for pCFG-synth enumerator in Algorithm 2

---

1: **procedure** REPLACENONTERMINALS($prog$, $P_G$)

2:     $NT \leftarrow$ list of nonterminals in $prog$

3:     **for** $\alpha \in NT$ **do**

4:         $(\alpha \times \beta) \sim Cat(|\pi[\alpha]|, \{\mathbb{P}[\pi[\alpha]_1], \mathbb{P}[\pi[\alpha]_2], ...\})$          ▷ Sample from distribution

5:         $prog \leftarrow prog.\{\alpha \rightarrow \beta\}$          ▷ apply rule to $prog$

6:     **end for**

7:     **return** $prog$

8: **end procedure**

9: **procedure** COMPLETEPROGRAM($prog$, $P_G$)          ▷ Replaces non-terminal symbols with terminal symbols

10:     $NT \leftarrow$ list of nonterminal symbols in $prog$

11:     **for** $\alpha \in NT$ **do**

12:         $(\alpha \times \beta) \sim Cat(|\pi[\alpha]|, \{\mathbb{P}_{\Sigma}[\pi[\alpha]_1], \mathbb{P}_{\Sigma}[\pi[\alpha]_2], ...\})$          ▷ Sample

13:         $prog \leftarrow prog.\{nt \rightarrow nt'\}$          ▷ apply rule to $prog$

14:     **end for**

15:     **return** $prog$

16: **end procedure**

---

With a naive implementation of this algorithm, the probability of our algorithm generating any sentence $s$ is equal to $\prod_{r_i \in D_s} \mathbb{P}[r_i]$, where $D_s$ is the left-most derivation of $s$. However, this will result in the algorithm generating the same programs multiple times, so we modify this algorithm in two ways: First, if we enumerate a complete program that we have seen before, we discard it; Second, we give a maximum depth limit, and if we are approaching the maximum depth limit, we sample only from the outgoing edges that result in complete programs.

---

**Algorithm 4** pCFG-synth

---

1: **procedure** PCFG-SYNTH(*prompts*, $\phi$, *G*)

2:      *conv* ← [ ]

3:      *progs* ← ∅

4:      **while** *prompts* ≠ ∅ **do**

5:          *response* ← LLM(*prompts*.*pop*(), *conv*)

6:          *conv*.*append*(*response*)

7:          *currentProg* ← EXTRACTPROGRAM(*response*)

8:          **if** $\forall \vec{x}\, \phi(currentProg, \vec{x})$ **then**

9:              **return** *currentProg*

10:          **else**

11:              *progs* ← *progs* ∪ *currentProg*

12:          **end if**

13:      **end while**

14:      *W* ← WEIGHTCOUNTER(*prog*, *G*)

15:      $W_G$ ← (*G*, *W*)

16:      *prog* ← CEGIS($W_G$, $\phi$)

17:      **return** *prog*

18: **end procedure**

---

### 4.4.3   Weighted $A^*$ Search

We implement a second variation of pCFG-synth using the $A^*$ weighted search algorithm as the underlying enumerator. $A^*$ is a search algorithm that chooses which paths to extend based on minimizing the cost of the path so far and an estimate of the cost required to extend the path to the goal, i.e., it expands nodes that minimizes

$$f(x) = c(x) + g(x),$$

where $c(x)$ is the cost of the path to $x$ so far and $g(x)$ is the estimated cost of reaching a goal node from $x$. This technique was first used for guiding synthesis by Lee et al. [57], and we adapted the algorithm from their work.

To implement our $A^*$ search, we extend the definition of the grammar tree to a weighted grammar tree. Given a pCFG $P_G = (V, \Sigma, R, S, \mathbb{P})$, a weighted grammar tree $\mathscr{T}(W_G)$ is a directed labelled graph as defined before, but each edge has a corresponding weight, given as follows:

$$\omega(\alpha \to \beta) = \begin{cases} -\log_2(\mathbb{P}[\alpha \to \beta]) & \text{if } \mathbb{P}[\alpha \to \beta] > 0, \\ \inf & \text{otherwise.} \end{cases}$$

We use the negative log of the probability to ensure that higher weighted edges correspond to those with very low probabilities.

The $A^*$ algorithm, shown in Algorithm 5, relies on two key functions: first, the function $c(x)$, which computes the cost of the path so far, and second, the function $g(x)$ which estimates the cost to extend the path to a goal node. Assuming $x$ is a sentential form in our language, $c(x)$ and $g(x)$ are given by:

$$c(x) = \sum_{r_i \in D_x} -\log_2(\mathbb{P}[r_i]),$$

$$g(x) = \begin{cases} 0 & \text{if } x \in \Sigma^*, \\ -\sum_{x_i \in V} \log_2 h(x_i) & \text{otherwise,} \end{cases}$$

where $x_i$ indicates the $i^{th}$ symbol in $x$, and $h$ is the upper bound of the probabilities of expressions that can be derived from $x_i$, and is calculated as the fixed point of:

$$\forall \alpha \in V . h(\alpha) = \max_{\alpha \to \beta \in R} \left( \mathbb{P}[\alpha \to \beta] \times \prod_{\beta_i \in V} h(\beta_i) \right),$$

The function $g(x)$ can then be thought of as the product of the probability of each non-terminal symbol in $x$ being converted into a terminal symbol.

**Smoothing the probability distributions:** Since the $A^*$ algorithm will not enumerate any programs whose derivation uses a rule with zero probability, we smooth the weighted grammar as follows:

$$w'[\alpha \rightarrow \beta] = 10 \times \left( \frac{w[\alpha \rightarrow \beta] + 1}{10} \right)^{\gamma},$$

with $\gamma = 0.4$.

---

**Algorithm 5** $A^*$ search for pCFG-synth

---

1: **procedure** ENUMERATE($P_G, \phi, cex$ )

2:     $Q = \{0, S\}$                 ▷ Priority queue of candidates

3:     **while** $Q \neq \varnothing$ **do**

4:        $(f, prog) \leftarrow Q.pop()$     ▷ Remove program with minimal $f$

5:        **if** $\forall \vec{x} \in cex. \phi(prog, \vec{x})$ **then**

6:           **return** $prog$

7:        **end if**

8:        **for** $(nt \in prog) \times nt'$ **do**

9:           **if** $(nt \times nt') \in P_G.R$ **then**       ▷ For all applicable rules

10:             $prog \leftarrow prog.\{nt \rightarrow nt'\}$       ▷ apply rule to $prog$

11:             $Q \leftarrow Q \cup (c(prog) + g(prog), prog)$

12:           **end if**

13:        **end for**

14:     **end while**

15: **end procedure**

---

## 4.5   Enumerative Synthesis with an Integrated LLM (iLLM-synth)

The disadvantage of the method described in the preceding section is that the language model cannot benefit from any additional information that

the enumerator learns during enumeration, as all prompting happens prior to starting the enumerative synthesis. In this section, we describe how we integrate an LLM into an enumerative synthesis algorithm, allowing it to update a probability distribution over the search grammar and to augment the grammar with new production rules, as shown in Algorithm 6.

---

**Algorithm 6** Top-down enumerator for iLLM-synth

---

1: **procedure** ENUMERATE($W_G, \phi, cex$ )

2:    $prog \leftarrow W_G.S$

3:    $d \leftarrow 0; i \leftarrow 0$

4:    $P_G \leftarrow$ BUILDPCFG($W_G$)

5:    **while** 1 **do**

6:        **if** $prog \in \Sigma^*$ **then**

7:            **if** $\forall \vec{x} \in cex. \phi(prog, \vec{x})$ **then**

8:                **return** $prog$

9:            **else**

10:                $prog \leftarrow S$

11:                $d \leftarrow 0$

12:            **end if**

13:        **end if**

14:        **if** $i\%n = 0$ **then**

15:            $W_G \leftarrow$ SYNTACTICFEEDBACK($W_G, prog, cex$)

16:            $P_G \leftarrow$ BUILDPCFG($W_G$)

17:        **end if**

18:        $prog \leftarrow$ REPLACENONTERMINALS($prog, P_G$)

19:        $d \leftarrow d + 1$

20:        **if** $d = maxDepth$ **then**

21:            $prog \leftarrow$ COMPLETEPROGRAM($prog, P_G$)

22:            **if** $prog \in PreviousPrograms$ **then**

23:                $prog \leftarrow S$

24:                $d \leftarrow 0$

25:            **end if**

26:        **end if**

27:        $i \leftarrow i + 1$

28:    **end while**

29: **end procedure**

---

---

**Algorithm 7** $A^*$ search for iLLM-synth

---

1: **procedure** ENUMERATE($P_G, \phi, cex$ )

2:    $Q = \{0, S\}$                                  ▷ Priority queue of candidates

3:    $i \leftarrow 0$

4:    **while** $Q \neq \varnothing$ **do**

5:        $(f, prog) \leftarrow Q.pop()$          ▷ Remove program with minimal $f$

6:        **if** $prog \in \Sigma^*$ **then**

7:            **if** $\forall \vec{x} \in cex. \phi(prog, \vec{x})$ **then**

8:                **return** $prog$

9:            **end if**

10:        **end if**

11:        **if** $i\%n = 0$ **then**

12:            $W_G \leftarrow$ SYNTACTICFEEDBACK($W_G, prog, cex$)

13:            $P_G \leftarrow$ BUILDPCFG($W_G$)

14:        **end if**

15:        **for** $(nt \in prog) \times nt'$ **do**

16:            **if** $(nt \times nt') \in P_G.R$ **then**          ▷ For all applicable rules

17:                $prog \leftarrow prog.\{nt \rightarrow nt'\}$          ▷ apply rule to $prog$

18:                $Q \leftarrow Q \cup (c(prog) + g(prog), prog)$

19:            **end if**

20:        **end for**

21:        $i \leftarrow i + 1$

22:    **end while**

23: **end procedure**

---

---

**Algorithm 8** Syntactic feedback generator in Algorithm 6,7.

---

1: **procedure** SYNTACTICFEEDBACK($W_G, prog, cex$)

2:     $prompt \leftarrow$ GENERATEPROMPT($prog, cex$)

3:     $response \leftarrow$ LLM($prompt$)

4:     $candidate \leftarrow$ EXTRACTPROGRAM($response$)

5:     $W_G.W \leftarrow W_G.W +$ WEIGHTCOUNTER($response$)

6:     $W_G.R \leftarrow W_G.R \cup (W_G.S \times response)$

7:     **return** $W_G$

8: **end procedure**

---

## 4.5.1   Integrated Prompting

We construct a prompt that asks the LLM to provide helper functions to assist a student in writing SMT-lib code. We give the LLM the constraints from the target synthesis problem and the partially complete program at the point the enumerator calls the LLM. If the LLM fails to solve the problem with this prompt, we later add the most recently failed candidate solution and the counterexample it failed on. These prompts are shorter than the prompts in those used in Section 4.3 and, therefore, cheaper and faster to run. An example Prompt 8 is as follows:

```
You are teaching a student to write SMT-LIB. The student
    must write a function that satisfies the following
    constraints:
(constraint (>= (function x y z) x))
(constraint (>= (function x y z) y))
(constraint (>= (function x y z) z))
(constraint (or (= x (function x y z)) (or (= y (
    function x y z)) (= z (function x y z)))))
So far, the student has written this code:
```

```
(define-fun function ((x Int) (y Int) (z 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.
```

Prompt 8: Integrated prompt for Benchmark 1.

## 4.5.2   Updating the Weighted Grammar

We initialise our algorithm with a weight of 1 for each rule in the gram-
mar.  We use the LLM-generated helper functions to augment the gram-
mar in the following way: first, any helper functions will be added directly
as new production rules to replace non-terminals of the correct type in
the grammar.  That is, if the LLM proposes the defined function $f$, a set
of rules of the form $V_i \times f$ are added to the grammar, for all non-terminal
symbols $V_i$ such that this rule results in syntactically correct expressions,
i.e., $V_i$ must be of the same type as the co-domain of $f$.  This is sufficient
to guarantee syntactically correct expressions because any functions pro-
posed by the LLM that are otherwise not well-formed, e.g., they reference
variables that are not defined, are discarded.  Any new rules are given a
weight equal to the average of all the current weights for rules relevant
to that non-terminal. The response parser also updates the weights of all
existing rules in the grammar, according to Equation 4.1, calculated from
the set of helper functions the LLM proposed.

### 4.5.3 Integrating Syntactic Feedback into Enumerative Search

We integrate the syntactic feedback generator into the probabilistic enumerator, shown in Algorithm 4, and into the $A^*$ weighted search, as shown in Algorithm 7. Both search algorithms call the syntactic feedback generator every $n^{th}$ iteration, where $n$ is a heuristic used to ensure the LLM is not called with the same partial program repeatedly and that the search algorithm has time to exploit the information obtained from the LLM. Note that, when the probabilistic grammar is updated, the $h$ values must be recalculated in the $A^*$ search.

## 4.6 Evaluation

We evaluate our approaches on benchmarks taken from the SyGuS competition [4], each with a grammar that corresponds to the full language of their respective theories. We evaluate across three SyGuS categories: Bit-Vector (BV), Linear Integer Arithmetic (LIA), and Invariants (INV). We evaluate both the LLM as a stand-alone synthesiser, the probabilistic enumerator, and $A^*$ implementations with a pre-trained pCFG and the enumerator with a pre-trained syntactic oracle. We utilise OpenAI's GPT-3.5-turbo-16k model to generate the prompts used for the pre-trained pCFG and the standalone LLM evaluation because this model supports longer prompts. We configure this with a temperature of 1.0, conversation-style messaging. We use GPT-3.5-turbo for iLLM-synth, which has shorter prompts. We use the 4.8.12 64-bit version of Z3 for verification and CVC5 version 1.1.0 as a baseline.

| Methods | BV (384) | | LIA (87) | | INV (138) | | Total (609) | |
|---|---|---|---|---|---|---|---|---|
| | # | 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% |

Table 4.1: Summary of results. We run nondeterministic results, marked °, 3 times and report the average (standard deviation is less than 1% for all methods except the baseline enumerator for the number of benchmarks solved).  We highlight the best result in terms of the number of benchmarks solved in each category. The timeout is 600s. Times in *italic* indicate results that may vary depending on the load on the OpenAI servers. The times for pCFG-synth do not include the time to call the standalone LLM and generate the wCFGs, but these are included in the times for LLM ∪ pCFG-synth.

## 4.6.1   Evaluation of the Stand-Alone LLM

We prompt the LLM until it produces up to 6 complete synthesis attempts per benchmark, with the results reported in line 1 of Table 4.1.  Any incomplete solutions are discarded (i.e., functions without a function body), although these are relatively rare, and we discard only 0.85% of programs we generate. In total, the LLM solves 49% of benchmarks, performing better in the invariant and LIA categories than the bit-vector category.  On

average, for the benchmarks it can solve, it takes 4 attempts to produce a correct solution. The average time for the LLM to generate a program is approximately 5$s$ using the OpenAI Python API. However, this is dependent on OpenAI, and we report these times only as estimates in Table 4.1. We allow the LLM only 6 attempts to solve the problem since, by the 6$^{th}$ iteration, the number of new solutions the LLM finds has dropped to < 2% (and it finds 0 new solutions for LIA).

## 4.6.2   Evaluation of pCFG-synth.

We evaluate both variants of pCFG-synth (with the probabilistic enumerator, denoted $e$-pCFG-synth, and with $A^*$, denoted $A^*$-pCFG-synth) using the wCFG obtained from the LLM. As a baseline, we run the same algorithms assigning a weight of 1 to every rule in the grammar (referred to as "enumerator" and $A^*$ respectively in the results). pCFG-synth increases the number of benchmarks the probabilistic enumerator can solve by 30%, but barely increases the number $A^*$ can solve, although the exact sets of benchmarks which $A^*$ and $A^*$-pCFG-synth solve do differ significantly. We hypothesise that this is because $A^*$, guided by the pCFG with equal weights for all rules, is very good at generating short solutions, and $A^*$-pCFG-synth is worse at short solutions but better at generating more complex solutions guided by the pCFG.

We also report the results obtained by the union of the LLM alone and pCFG-synth, i.e., if the LLM solves the benchmark, we do not deploy the enumerator. This is a more realistic representation of how such a technique would be used and demonstrates that the enumerator can overcome shortcomings of the LLM and vice versa. The union of the LLM and $A^*$-pCFG-synth substantially outperforms CVC5, solving 73 more benchmarks.

### 4.6.3  Evaluating iLLM-synth.

We evaluate both variants of iLLM-synth, denoted *e*-iLLM-synth and $A^*$-iLLM-synth. We set the temperature for *e*-iLLM-synth to 1, but find that $A^*$-iLLM-synth performs better with a temperature set to 0, which we hypothesise is due to the determinism of the algorithm. We find that iLLM-synth outperforms the enumerator of pCFG-synth, and gets close to the performance of CVC5, suggesting that the ability to prompt the LLM with additional information obtained during enumeration allows the LLM to provide better guidance to the enumerator, as well as to more frequently propose useful helper functions. We do find that iLLM-synth performs less well than methods incorporating the stand-alone LLM on the invariant benchmarks, which is likely because the invariant benchmarks benefit from the custom prompting technique described in Section 4.3.1. Future work would involve identifying further categories of benchmarks that benefit from custom prompts. It is worth noting that neither the probabilistic enumerator nor the $A^*$ implementation includes many of the optimisations that mature solvers such as CVC5 implement, and yet, by integrating these simple algorithms with syntactic feedback from an LLM, they have achieved performance on par with the state-of-the-art enumerative solver.

### 4.6.4  Failure Modes.

We manually examine a sample of the stand-alone LLM errors and give examples of such errors. Broadly, we identify the following common failures: Misunderstandings due to complex constraints (the LLM suggests solutions that are not syntactically close to the correct solution); simple syntactic errors, e.g., applying non-commutative operators to operands in the wrong order, concatenating bit-vectors in the wrong order or hallu-

cinating operations; simple semantic errors, e.g., operators in the wrong order. Errors in the first category are not helpful to our guided enumerators, but the remaining categories of error still allow us to generate a wCFG that is likely to indicate the area of the solution. The benchmarks that CVC5 can solve and our enumerative techniques cannot, tend to have complex constraints and relatively short solutions that use less common operators (e.g., bitwise operators). We hypothesise that the LLM guidance becomes an impediment to the enumerator in these scenarios. In contrast, the average length (in characters) of a solution for benchmarks uniquely solved by the LLM is 4.7x the length of a solution for benchmarks uniquely solved by CVC5. Using the LLM to guide the enumerators increases the length of solutions that the enumerators can find, for instance, all solutions found by $A^*$ contain fewer than 3 operators, but $A^*$-iLLM-synth finds solutions with greater than 20 operators.

### *Completing verification conditions:*

In the initial phase of our research, we constructed prompts based on asking the LLM to complete an SMT-LIB file that encodes the verification conditions for the synthesis problem, so that it would be unsatisfiable. Consider the following example:

```
1 (set-logic LIA)
2 (synth-fun function ((x Int) (y Int)) Int)
3 (declare-var x Int)
4 (declare-var y Int)
5 (constraint (>= (function x y) x))
6 (constraint (>= (function x y) y))
7 (constraint (or (= x (function x y)) (= y (function x y)
      )))
```

```
8  (check-synth)
```

Benchmark 2: A SyGuS specification that asks for a program that returns *x* if *x − y ≥* 0 otherwise returns *y*.

The corresponding prompt structure is as follows:

```
; Complete the following SMT file so that it is
    unsatisfiable
(set-logic LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (or (not (>= (function x y) x))
            (not (>= (function x y) y))
            (not (or (= x (function x y)) (= y (function
              x y))))))
(check-sat)
(define-fun function ((x Int) (y Int)) Int
(
```

Prompt 9: LLM direct fill-in task.

Here, the LLM's task was to fill in missing information. This initial approach did not yield satisfactory results.

### Fine-tuning:

We experimented with fine-tuning the Curie model on a subset of the benchmarks, using `\n;###\n` for prompt endings and `\n;END` for completion markers. An example structure for fine-tuning from the training set is provided here:

```
{"prompt":
    " (set-logic BV)\n(declare-fun x () (_ BitVec 32))\n
```

```
(define-fun hd03 ((x (_ BitVec 32))) (_ BitVec 32)\n
(bvand x (bvneg x)))\n
(assert (or (not (= (hd03 x) (f x)))))\n
(check-sat)\n
; define function f\n
(define-fun f ((x (_ BitVec 32))) (_ BitVec 32)\n
;###\n",
 "completion":
  " (bvand x (bvneg x))) \n;END"}
```

Post-adjustment, the Curie model was evaluated using a smaller test set, excluding invariant synthesis, and achieved a 25% success rate in benchmarks. Further refinements to the training set, such as expanding let expressions, did not improve results.

### Synthesis via Python code:

We experimented with asking the LLM to write solutions in Python and then translate the given solution into an SMT-LIB function.

```
(set-logic LIA)
(synth-fun function ((x Int) (y Int)) Int))
(declare-var x Int)
(declare-var y Int)
(constraint (>= (function x y) x))
(constraint (>= (function x y) y))
(constraint (or (= x (function x y)) (= y (function x y)
   )))
(check-synth)
Write a Python method function. Requirements: 1. No
   built-in Python functions. 2. Adhere to the above SMT
```

```
-LIB constraints.
```

Prompt 10: Prompt for `function` shown in Benchmark 2.

Upon the LLM generating a possible solution, we employed an additional prompt to convert the Python method into the SMT-LIB function format:

```
Translate the Python method into SMT-LIB code function,
    focusing solely on the SMT-LIB format.
```

Prompt 11: Request for converting Python to SMT-LIB code.

This did not work well, as the LLM frequently failed to translate Python into SMT-LIB accurately.

| Prompt Method | Solved | % |
|:---:|:---:|:---:|
| Lisp | 22/50 | 44% |
| Python | 19/50 | 38% |

Table 4.2: Comparative analysis of prompt methods: efficacy in solving benchmarks with Lisp, and Python prompts.

Table 4.2 presents a comparative study of various prompting methods in solving benchmarks, highlighting the effectiveness of Lisp, and Python-based prompts. The results indicate a notable enhancement in problem-solving efficiency when using Lisp prompts.

**Unravelling LLM's shortcomings:**

LLM often encounters difficulties with benchmarks characterized by multiple constraints involving longer derivations. These predefined functions contribute to the complexity and length of each constraint, posing a significant challenge for the LLM. For example:

```
1 (constraint (functionA (and (= a 0) (= 0 d)) (= 0 (
     functionB a d))))
2 (constraint (functionA (functionD (= g (functionB a d))
     (>= a d) (= b (+ a 1)) (= h (+ g 1)) (= i (+ h 1)))
     (= i (functionB b d))))
3 (constraint (functionA (functionD (= g (functionB a d))
     (>= a d) (= e (+ d 1)) (= h (+ g 1)) (or (and (< a e)
      (= j (+ h 1))) (and (>= a e) (= j h)))) (= j (
     functionB a e))))
4 (constraint (functionA (functionC (= g (functionB a d))
     (< a d) (= b (+ a 1)) (= h (+ g 1))) (= h (functionB
     b d))))
5 (constraint (functionA (functionD (= g (functionB a d))
     (< a d) (= e (+ d 1)) (= h (+ g 1)) (= i (+ h 1))) (=
      i (functionB a e))))
```

Benchmark 3: Multiple long constraints for `functionB`.

If we expand the function applications, we get Benchmark 4, which still contains long derivations.

```
1 (constraint (or (not (and (= a 0) (= 0 d))) (= 0 (
     functionB a d))))
2 (constraint (or (not (and (and (and (= g (functionB a d)
     ) (>= a d)) (= b (+ a 1))) (= h (+ g 1))) (= i (+ h
     1))) (= i (functionB b d))))
3 (constraint (or (not (and (and (and (= g (functionB a d)
     ) (>= a d)) (= e (+ d 1))) (= h (+ g 1))) (or (and (<
      a e) (= j (+ h 1))) (and (>= a e) (= j h)))) (= j (
     functionB a e))))
4 (constraint (or (not (and (and (= g (functionB a d)) (<
     a d)) (= b (+ a 1))) (= h (+ g 1))) (= h (functionB b
```

```
      d)))))
5 (constraint (or (not (and (and (and (= g (functionB a d)
      ) (< a d)) (= e (+ d 1))) (= h (+ g 1))) (= i (+ h 1)
      )) (= i (functionB a e))))
```

<div align="center">Benchmark 4: Expanded constraints for functionB.</div>

Such constraints are intricate and challenging for the LLM to under-
stand accurately, making it difficult to produce a response satisfying the
specification.  The LLM produced a program that contains functionB it-
self. However, the pCFG-synth and iLLM-synth can solve the problem with
the distribution of rules, whereas the state-of-the-art solver CVC5 cannot.

In contrast, the LLM exhibits a better understanding of benchmarks
with shorter derivation for each constraint, such as:

```
1 (constraint (functionA (and (< a 100) (< c 100)) (
      functionB a b c)))
2 (constraint (functionA (functionC (functionB a b c) (< a
      100) (< 100 c) (functionA vr4 (and (= e (+ a 1)) (=
      f c))) (functionA (not d) (and (= e (- a 1)) (= f (-
      c 1))))) (functionB e b f)))
3 (constraint (functionA (and (functionB a b c) (not (and
      (< a 100) (< 100 c)))) (or (>= a 100) (<= c 100))))
```

<div align="center">Benchmark 5: Shorter derivations in constraints for functionB.</div>

These constraints are less complex, making them more accessible for the
LLM to process.

Another challenge arises with benchmarks that necessitate very short
solutions. For instance, a benchmark requiring a solution like:

```
(define-fun function ((a Int)) Bool true)
```

<div align="center">Response 3: A concise program unattainable by LLM.</div>

In these cases, the LLM tends to propose lengthier solutions, failing to align with the benchmark's requirement for conciseness. This pattern is evident across all 6 attempts made by the LLM.

### Shuffling operator or variable:

Below is an example benchmark for finding f:

```
1 (set-logic BV)
2 (define-fun function ((x (_ BitVec 32))) (_ BitVec 32)
3     (bvor x (bvadd x #x00000001)))
4 (declare-var x (_ BitVec 32))
5 (constraint (= (function x) (f x)))
6 (check-synth)
```

Benchmark 6: A SyGuS specification asking for a Bit-Vector program.

LLM failed to provide this exact solution but contributed significantly by suggesting a similar program with the correct operators, albeit in an incorrect order. The program suggested by the LLM is:

```
(define-fun function ((x (_ BitVec 32))) (_ BitVec 32) (
    bvor x (bvadd x (concat #b1 (_ bv0 31)))))).
```

Response 4: Correct operators with incorrect order.

In this program, the LLM correctly identifies the use of a bvor and bvadd but misplaces the components of the bit-vector representing the number 1. The correct solution requires swapping the positions of #b1 and (_ bv0 31) in the concatenation.

This case demonstrates that while the LLM may not always generate a completely accurate solution, it can provide valuable insights or components of a solution. The enumerator, guided by the distribution of rules inferred from the LLM-generated programs, can utilise these insights to

reach a correct solution.

**Mistaken conditional statements:**

In situations where an individual constructs an if-then-else statement but makes an error in defining the correct condition, we can identify this issue through a specific example of finding `function`:

```
1 (constraint (>= (function a b c d e f g) a))
2 (constraint (>= (function a b c d e f g) b))
3 (constraint (>= (function a b c d e f g) c))
4 (constraint (>= (function a b c d e f g) d))
5 (constraint (>= (function a b c d e f g) e))
6 (constraint (>= (function a b c d e f g) f))
7 (constraint (>= (function a b c d e f g) g))
8 (constraint (or (= a (function a b c d e f g)) (or (= b
     (function a b c d e f g)) (or (= c (function a b c d
     e f g)) (or (= d (function a b c d e f g)) (or (= e (
     function a b c d e f g)) (or (= f (function a b c d e
     f g)) (= g (function a b c d e f g)))))))))
```

Benchmark 7:  SyGuS specification for constructing a program contains several *if-then-else* Operators.

    In this scenario, the accurate and appropriate solution should be

```
1 (define-fun function ((a Int) (b Int) (c Int) (d Int) (e
     Int) (f Int) (g Int)) Int
2    (ite (and (>= a b) (>= a c) (>= a d) (>= a e) (>= a
        f) (>= a g)) a
3       (ite (and (>= b a) (>= b c) (>= b d) (>= b e)
           (>= b f) (>= b g)) b
4          (ite (and (>= c a) (>= c b) (>= c d) (>= c e
```

```
              ) (>= c f) (>= c g)) c
5                 (ite (and (>= d a) (>= d b) (>= d c)
                    (>= d e) (>= d f) (>= d g)) d
6                      (ite (and (>= e a) (>= e b) (>= e
                         c) (>= e d) (>= e f) (>= e g))
                         e
7                          (ite (and (>= f a) (>= f b)
                             (>= f c) (>= f d) (>= f e)
                             (>= f g)) f g)))))))
```

Program 4.19: Solution program for Benchmark 7.

However, the LLM generates:

```
(define-fun function ((vr0 Int) (b Int) (c Int) (d Int)
    (e Int) (f Int) (g Int)) Int
    (ite (and (<= vr0 b) (<= b c) (<= c d) (<= d e) (<= e
        f) (<= f g)) g
      (ite (and (<= vr0 b) (<= b c) (<= c d) (<= d e)
         (<= e f)) f
        (ite (and (<= vr0 b) (<= b c) (<= c d) (<= d
           e)) e
          (ite (and (<= vr0 b) (<= b c) (<= c d))
             d
          (ite (and (<= vr0 b) (<= b c)) c
             (ite (<= vr0 b) b vr0)))))))
```

Response 5: Program generated by LLM for Example 7.

, which, although sharing a similar structure, inaccurately defines the conditions.

***Unorthodox nesting and syntax errors:***

A common problem occurs when LLM endeavours to construct a nested if-then-else statement.  However, their nesting level is insufficient, and they also introduce operators incompatible with SyGuS. For instance, while the correct solution should involve a properly nested if-then-else structure

```
1  (define-fun function ((a Int) (b Int)) Int
2  (ite (and (>= (+ a b) 2)
3       (ite (>= (+ a b) 3)
4            (ite (>= (+ a b) 4)
5                 (ite (>= (+ a b) 5)
6                      (or (>= (+ a b) 6)
7                           (= a (+ 11 (* (- 1) b))))
8                      (= a (+ 1 (* (- 1) b))))
9                 (= a (+ 3 (* (- 1) b))))
10               (= a (+ 1 (* (- 1) b)))))
11      (+ (- 60) (* 60 a) (* 60 b))
12      (ite (ite (>= (+ a b) 2)
13           (and (>= (+ a b) 3)
14                (ite (>= (+ a b) 4)
15                     (ite (>= (+ a b) 5)
16                          (or (= a (+ 11 (* (- 1) b)))
17                              (not (>= (+ a b) 6)))
18                          (= a (+ (- 9) (* (- 1) b))))
19                     (= a (+ (- 1) (* (- 1) b)))))
20           (= a (+ (- 1) (* (- 1) b))))
21           (+ 50 (* 50 a) (* 50 b))
22           (ite (and (>= (+ a b) 2)
23                (ite (>= (+ a b) 3)
24                     (ite (>= (+ a b) 4)
```

```
25                        (or (not (>= (+ a b) 5))
26                            (ite (>= (+ a b) 6)
27                                 (= a (+ 1 (* (- 1) b)))
28                                 (= a (+ (- 9) (* (- 1) b))))
                                 )
29                            (= a (+ 7 (* (- 1) b))))
30                        (= a (+ 1 (* (- 1) b)))))
31                    (+ (- 40) (* 40 a) (* 40 b))
32                    (ite (and (>= (+ a b) 2)
33                          (ite (>= (+ a b) 3)
34                               (ite (>= (+ a b) 4)
35                                    (or (>= (+ a b) 6)
36                                        (not (>= (+ a b) 5))
37                                        (not (= a (+ (- 1) (* (- 1)
                                                  b)))))
38                                    (not (= a (+ (- 1) (* (- 1) b)))
                                         ))
39                               (not (= a (+ 3 (* (- 1) b))))))
40                         (+ (- 20) (* 20 a) (* 20 b))
41                         (+ 10 (* 10 a) (* 10 b)))))))
```

Program 4.21: Solution program with nested *if-then-else* operators.

The LLM generates the program below Program 4.22, which does not achieve the required nesting level and erroneously incorporates operators. For example, `if` is syntactically incorrect in the SyGuS framework.

```
1  (define-fun function ((a Int) (b Int)) Int
2      (ite (<= (functionA a b) 1)
3          (if (= (functionA a b) 1)
4              (functionG (functionB a b 1))
5              0)
```

```
6          (ite (<= (functionA a b) 2)
7            (if (= (functionA a b) 2)
8              (functionG (functionH (functionB a b (-
                 1))))
9              0)
10           (ite (<= (functionA a b) 3)
11             (if (= (functionA a b) 3)
12               (functionG (functionC (functionB a b
                   1)))
13               0)
14             (ite (<= (functionA a b) 4)
15               (if (= (functionA a b) 4)
16                 (functionG (functionD (functionB
                     a b (- 1))))
17                 0)
18                 (ite (<= (functionA a b) 5)
19                   (if (= (functionA a b) 5)
20                     (functionG (functionE (
                         functionB a b 1)))
21                     0)
22                   (functionG (functionF (functionB
                       a b (- 1))))))))))))
```

Program 4.22: Program with insufficient nesting and syntactic incompatibilities.

## 4.6.5  Programming-by-Example.

We omit benchmarks from the syntax-guided synthesis competition tracks that solely focus on programming-by-example (PBE)(i.e., specifying a pro-

gram only using input-output examples and a grammar). We omit these benchmarks for two reasons: first, since training data is trivial to generate for PBE, unlike general logical specifications [85], there are many other successful machine-learning driven synthesis techniques that can be trained for PBE techniques[11]. Second, our approaches are effective when the LLM can provide guidance to the enumerator, which comes from prompting the LLM with the logical constraints that form the specification. If we prompt the LLM using the prompting techniques outlined in Section 4.3.1 with a PBE specification, it tends to provide a solution in the form of a large case split over the input examples, which returns specific outputs for each input. This is not useful for guiding the enumerator because the LLM overfits to the examples in the specification and fails to provide any bias towards operators other than "if-then-else". To extend our approach to PBE, we would need to use a prompting approach tailored to input-output examples.

## 4.7 Threats to Validity

**LLM Training Data:** The SyGuS problems are publicly available and might be part of the training data for the LLM we use, although we believe the solutions were not publicly available at the time of training.

**Reproducibility:** These experiments use GPT-3.5, an LLM available via API from OpenAI. We have recorded the responses and parameters generated by the LLM in all experiments, but these may not be reproducible [82] since GPT-3.5 behaves non-deterministically in a way that cannot be seeded. However, we observe very small variations in the number of benchmarks solved in our experiments (although greater variation in the average solving time). It is also possible that OpenAI

deprecates this LLM and its associated API or updates it and changes its behaviour in the future.

**Benchmark Bias:** The benchmark set is taken from the SyGuS competition [4], but may not be very diverse and may not be representative of synthesis problems "in the wild". Nevertheless, this is a standard benchmark set used in many formal synthesis papers.

**Hyperparameters:** We have not invested time in parameter tuning, and better or worse results may be obtained by changing the LLM parameters (temperature), or adjusting the weights, enumeration depth, and heuristic functions in the probabilistic enumerator and $A^*$ algorithms.

# Chapter 5

# Guided Tensor Lifting

The material in this chapter comes from paper *Guided Tensor Lifting* [61], published at *PLDI 2025*. I designed and implemented the framework, algorithms and carried out the evaluation. The verification components were developed jointly with José Wesley de Souza Magalhães and Alexander Brauckmann.

## 5.1 Introduction

Building on the probabilistic-guidance framework introduced in Chapter 4, this chapter explores how to transplant those ideas into a very different setting, automatically lifting dense tensor kernels from C into the TACO domain-specific language (DSL).

Recent years have witnessed rapid growth in the number and importance of machine learning workloads. While used in a diverse number of applications, their fundamental building block is tensor contractions, which dominate execution time. For this reason, a large number of specialised tensor domain-specific languages (DSLs) have appeared, capable of producing high-performance code ([52, 38, 86, 1, 18, 37]). Their associated

73

compilers are capable of extracting domain-specific information to exploit hardware-specific features and vendor-tuned libraries.

To access such performance, applications have to be written in one or more high-level DSLs.  While this is acceptable for new applications, it means that existing workloads written in standard programming languages are unable to directly access a platform's potential performance. While manually rewriting a program to a DSL may be a worthwhile cost, it becomes a serious impediment if it has to be repeated for new emerging DSLs.  This problem of manual porting or *lifting* existing code to higher-level DSLs has been identified by several recent works that propose automated techniques. The most popular approaches use varying forms of program synthesis, where a DSLs space is searched for a matching program ([49, 98, 70]).  However, program synthesis is expensive and struggles to scale to multi-dimensional tensor workloads.

To overcome this scalability issue, existing schemes rely on aggressive hard-wired heuristics that trade-off coverage for time.  In [70], domain-specific polyhedral analysis is used to prune the search space. This works well on low-dimensional problems but suffers from exponential growth. Similarly, in [92], the user provides a template to aid search. While narrowly effective, such heuristics limit portability and are a limit to generalisation.

A completely different approach is to use neural machine translation based on large language models (LLMs).  They have proved highly successful with a number of program generation tasks [25, 30]. They are fast and scale with program complexity, but unfortunately, they are inaccurate. What we would like is to combine the power of LLMs with the accuracy of synthesis.

This chapter explores a novel combination of LLMs and program synthesis. It uses an LLM to suggest a number of possible solutions. It then

builds a probabilistic grammar of templates, based on the proposed solutions, and then uses this grammar to drive an enumerative search of grammar templates. Our approach, termed STAGG (Synthesis of Tensor Algebra Guided by Grammars) is able to outperform all existing approaches. It achieves 99% lifting accuracy on a pre-existing large-scale benchmark suite of dense tensor algebra and is able to do this without any pre-wired heuristics.

**Contributions:**

This chapter makes the following contributions:

- Two novel synthesis algorithms that combine LLM guesses and program synthesis to scalably lift dense tensor code.

- A large-scale evaluation of state-of-the-art tensor program lifting.

- Greater coverage than existing techniques.

## 5.2   Motivation

The tensor DSL we target in this chapter is TACO [52]. Whilst TACO may be best known for sparse computation, it also gives superior performance over dense code on multicores and GPUs. Although recent work has tackled *matching* certain sparse computation to specific high-performance APIs [55, 35], this chapter focuses solely on *lifting* legacy dense tensor computation to a high-level programming language [51].

TACO syntax is based on Einstein summation (einsum) notation, a language for representing linear algebra operations using indexing expressions. The TACO compiler takes a tensor expression as input and generates highly optimised kernels. TACO-generated code exploits the parallel nature of both dense tensor algebra operations and multi-core/GPU

architectures.  It uses domain-specific knowledge to optimise and auto-parallelise, which related work has reported results in average speedups of 1.8× and 24.1× over the original program on CPUs and GPUs respectively [70].

**Einsum notation:**

TACO supports einsum notation, as do other frameworks including Py-Torch [86], Halide [94].  While these alternatives may support a larger set of operations, targeting TACO allows a direct apples-to-apples comparison against prior work [70, 93].

Einsum expressions consist of a sequence of indexing variables, each one representing an iterator over a different tensor dimension. The traditional einsum notation expresses tensor multiplication and implicit summation on the indices that are absent in the output tensor. TACO uses an extended version of the original notation that also supports subtraction and division.  Unlike other einsum-based frameworks such as the NumPy [38] einsum API, the tensors in TACO programs must be explicitly declared. Figure 5.4 shows the TACO grammar addressed in this chapter.

**Problem statement:**

Formally, given a legacy program $p_s$, written in a low-level language such as C, STAGG aims to find an equivalent program $p_t$ written in TACO, that meets the specification $\forall \vec{x}.p_s(\vec{x}) = p_t(\vec{x})$, where $\vec{x}$ is a vector of input arguments.  That is, $p_t$ produces the same output as $p_s$ on all possible inputs.

**Example:**

The synthesis task that STAGG solves is to synthesise a TACO program $p_t$ such that $\forall \vec{x}.p_s(\vec{x}) = p_t(\vec{x})$, where $\vec{x}$ is a vector of inputs, in this case `Mat1`, `Mat2`, `Result`. We now illustrate this approach on the example input C

program shown in Benchmark 8.

Given this program, STAGG first queries a large language model to ask for a set of candidate solutions. The prompt template we use is shown below in Prompt 12. This gives us a set of candidate solutions in Response 6.

STAGG then learns a probabilistic context-free grammar that captures this set of solutions as templates. We describe how we learn this grammar in Section 5.4.2. The grammar below in Figure 5.1 shows probabilities for each production rule in parentheses. Each tensor and constant in the grammar is treated as a symbolic variable, which can later be replaced when the template is instantiated.

```c
void function(int N, int* Mat1, int* Mat2, int* Result){
    int* p_m1;
    int* p_m2;
    int* p_t;
    int  i, f;
    p_m1 = Mat1;
    p_t  = Result;
    for (f = 0; f < N; f++) {
        *p_t = 0;
        p_m2 = &Mat2[0];
        for (i = 0; i < N; i++)
            *p_t += *p_m1++ * *p_m2++;
        p_t++;
    }
}
```

Benchmark 8: A C implementation of $\sum_{i=0}^{N-1} \text{Mat1}(f \times N + i) \cdot \text{Mat2}(i)$. The result is a dot product between the $f$-th row of `Mat1` and vector `Mat2`. The equivalent synthesized TACO expression is `a(i) = b(i,j) * c(j)`

You are a scientific assistant that knows a lot about
    transpilation. Translate the following C code to an
    expression in the TACO tensor index notation. The
    expression must be valid as input to the taco
    compiler. Return a list with 10 possible expressions.
     Return the list and only the list, no explanations.
{the input C program}

Prompt 12:  The prompt requesting 10 TACO expressions for a given C
program.  The temperature we use is 1.0, and the role is  "You  are  a
scientific assistant that knows a lot about transpilation"
.

We use a weighted $A^*$ search to explore the space of the grammar, in-
spired by work in the literature [64, 57], enhanced with penalty functions
that penalise (partial or complete) templates that fail to adhere to syntactic
constraints.  When a complete template is found, this is passed to a tem-
plate validator, which searches for all possible instantiations of the tem-
plate and evaluates them against a set of input-output examples.  A valid
template, in this instance, would be the template `a(i) = b(i,j) * c(j)`.
This is instantiated to the concrete program `Result(i) = Mat1(i,j) *
Mat2(j)`. We compile this TACO program using the TACO compiler into C
code, and check with bounded model checking that the two pieces of C
code are equivalent.

r(f) = m1(i, f) * m2(f)
Result(i) = Mat1(i,f)*Mat2(f)
Result(i) := Mat1(f,i) * Mat2(i)

```
Result(f) = sum(f, mat1(f, i) * mat2(i))
```

Response 6: LLM-generated candidate solutions for matrix product computations based on the implementation in Benchmark 8. Displayed are a subset of 10 generated solutions, trimmed for brevity.

```
1  PROGRAM        ::=  TENSOR1 "=" EXPR (1)
2  TENSOR1        ::=  "a(i)" (1)
3  EXPR           ::=  TENSOR2 (0) | CONSTANT (0) |
                       EXPR OP EXPR (1)
4  OP             ::=  "+" (0.2) | "-" (0) | "*" (0.8) |
                       "/" (0)
5  TENSOR2        ::=  "b(i,j)" (0.2) | "b(j, k)" (0.1) |  …
                       "c(i)" (0.3) | "c(j)" (0.2) | "c(k)"
```

Figure 5.1: A probabilistic context-free grammar template.

## 5.3    Overview of STAGG
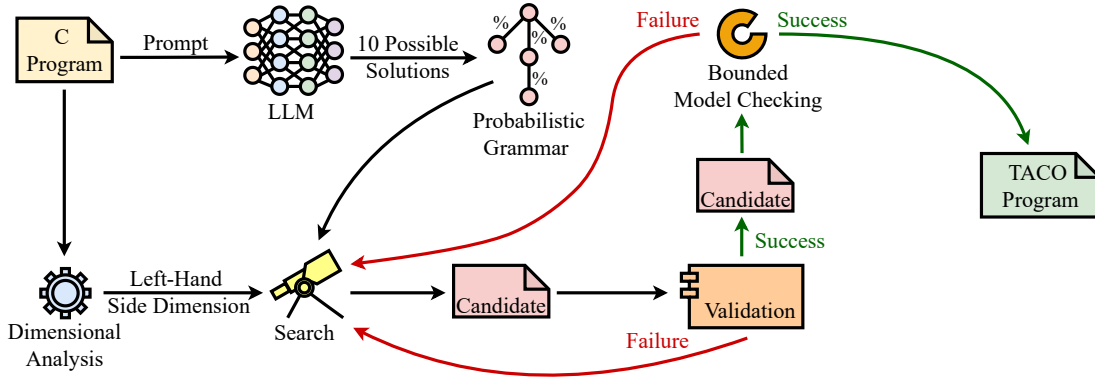


Figure 5.2: Overview of STAGG. We query the LLM to provide 10 possible solutions in TACO that are equivalent to the input code C. Based on the LLM response, we build a probabilistic grammar and enumerate the space of template programs described by the said grammar. We validate a candidate using I/O examples, and if it passes all tests, we proceed to verification to prove equivalence with the original C implementation. The input code is also analysed to predict the dimensionality of the left-hand side tensor in the solution.

Lifting to tensor DSLs is a challenging problem for program synthesis, and existing enumerative techniques are capable of accurate translation but rely on hand-written heuristics in order to scale. In contrast, highly scalable machine-learning-based approaches like language models fail to give accurate translations due to the complexity of the benchmarks. The key insight behind STAGG is that we can achieve the best of both worlds by using an LLM to *learn* the heuristics for an enumerative solver.

To that end, STAGG, as shown in Figure 5.2, implements a multi-staged hybrid synthesis approach.

① First, we construct a prompt based on the input C code and ask the LLM to propose 10 translated solutions.

② We proceed to construct a probabilistic grammar, which represents the space of solutions in the form of templates.

③ We then search this space of templates with a two-stage enumerative search: first, we search the space of templates with a search inspired by $A^*$, then, given a template, we search for a valid completion of the template against a set of input-output examples.

④ If a completed template is found that satisfies all input-output examples, we perform bounded verification with a bounded model checker to validate that the completed template is equivalent to the original C code. If it fails verification, we return to the template enumeration stage.

## 5.4   Learning a Grammar of Templates

The first step of STAGG uses a large language model to generate a set of candidate solutions for the synthesis problem in hand, using the prompt shown in Section 5.2. This gives us a set of candidate solutions, *P*. We ask for 10 solutions, but we parse in as many solutions as the LLM gives us (which is sometimes more than 10) and discard any syntactically incorrect solutions.

Given a set of incorrect candidate solutions from the LLM, we hypothesise that, even though none of the candidate solutions were precisely correct, the correct solution is likely to lie in the neighbourhood of the LLM's guesses. To that end, we characterise this neighbourhood using a probabilistic grammar of templates. We use a context-free grammar, but note that, in principle, any probabilistic model that characterises the neighbourhood of guesses could be used. First, let us define some of the preliminar-

ies we will need for this section.

### 5.4.1   Constructing a Grammar of Templates

Capturing the search space as a grammar of templates rather than complete TACO programs has two advantages: first, in comparison to using the full TACO grammar, it reduces the search space that the enumerative synthesis process has to search; and second, it allows us to group semantically equivalent but syntactically different candidate expressions together as one when constructing the weighted context-free grammar. For example, expressions like `1 t(f) = m1(i, f) * m2(f)` and `3 Target(i) := Mat1(f,i) * Mat2(i)` in LLM Response 6, are equivalent in structure (we use a preprocessing to swap := to = before parsing), yet they would yield different terminal rules in the full grammar due to variations in notation.
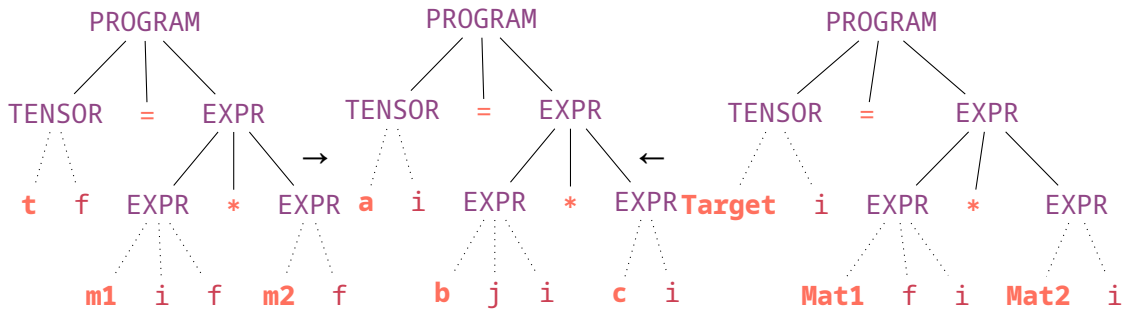


Figure 5.3: Expression standardisation. We omit part of the derivation for brevity.

Given a set of candidate solutions, the first step is to construct a grammar of *templates* that captures the full set of solutions. The full grammar for TACO programs, $G_{TACO}$ is shown in Figure 5.4. A TACO program is any program in $\mathscr{L}(G_{TACO})$.

**Definition 3** (Templates). *We define a* TACO *template $\tau$ to be any string ob-*

tained by taking a program $p \in \mathcal{L}(G_{TACO})$ *and replacing all tensors with symbolic tensor variables, denoted* $t_1, t_2, ...$ *and all constants with constant symbols* $const_1, const_2, ...$

Given a set of candidate programs from the LLM, $P$, we aim to find a grammar $G_\tau$ that contains a set of templates $\mathcal{T}$ that allow us to generate all programs $p \in P$. A substitution

$$S = (t_1 \mapsto s_1^1, t_2 \mapsto s_2^1, ... , const_1 \mapsto s_1^c)$$

is mapping from symbolic tensor variables and constant symbols to terminal symbols in the grammar $G_{TACO}$. A template *generates* a program if

$$\exists S.\tau.\{S\} = p,$$

where $\tau.S$ indicates the result of replacing all occurrences of $t_1$ with $s_1^t$, and $t_2$ with $s_2^t$ etc in $t$. Thus, our requirement on our grammar is two constraints: first, that

$$\forall \tau.(\tau \in \mathcal{T}) \implies \tau \in \mathcal{L},$$

and second, that

$$\forall p \in P.\exists \tau \in \mathcal{T} \wedge \exists S.\tau.\{S\} = p.$$

This is obviously trivially satisfied by the complete grammar of TACO programs, yet it is undesirable as it has not reduced our search space. We also aim to create a grammar that is as small as possible, whilst avoiding over-fitting, and we attempt to optimise this trade-off by the method of construction described in the following section.

### ***Templatized candidate solution:***

We obtain the grammar $G_\tau$ by first inferring a template for each solution in $P$. The first step involves parsing each $p \in P$ into an Abstract Syntax Tree

(AST), a structured representation that captures the hierarchical organisation of the expression. The AST organises operations, tensor identifiers, and indices as distinct nodes, allowing systematic traversal and manipulation. For example, consider the expression `1 t(f) = m1(i, f) * m2(f)` in Response 6 can be parsed as the left tree in Figure 5.3. We then transform the AST in three stages: *Tensor Templatization*, *Index Standardisation*, and *Constant Templatization*.

**Tensor templatization:**

We replace each tensor name in the expression with a symbolic tensor variable. From hereon, we use `a, b, c`, as the symbolic tensor variables $t_1, t_2, t_3, ...$, to align with the variable names in the code examples. The identifiers are assigned in alphabetical order—starting with a for the left-hand side tensor and using b, c, d, ... sequentially for tensors on the right-hand side, based on their order of first appearance. The expression `t(f) = m1(i, f) * m2(f)` will be transformed into `a(f) = b(i,f) * c(f)` by this step.

**Index standardisation:**

The index standardisation step ensures that each tensor expression in the grammar uses a consistent set of index variables, irrespective of the original indices in the input expression. Each unique index variable encountered in an expression is mapped to the next available symbol from the canonical set $\{i, j, k, l\}$ in alphabetical order. The expression `a(f) = b(i,f) * c(f)` will be transformed to `a(i) = b(j,i) * c(i)` by this step, as shown in the middle in Figure 5.3. The index variables do not need to be replaced by template instantiation as they are local variables to the program.

**Constant templatization:**

Any constants in the candidate solutions are replaced with a symbolic

constant `Const`. The template instantiation step will instantiate from a list of constants found in the input source code.

***Refining the grammar:***

Given the templatized candidate solutions $\mathscr{T}$, we wish to construct a probabilistic grammar that represents the space of these solutions, without substantially over-fitting. The first step is to construct a context-free grammar that defines this set of templates.

We start with the base grammar of TACO programs, shown in Figure 5.4. We then restrict the set of tensor names to be the names we have chosen as symbolic tensor variables and constants, namely *a*, *b*, *c*, ... and *const*, and also limit the set of index variables to be *i*, *j*, *k*, *l*, .... In theory, this permits 26 tensor IDs and 4 index variables, because one can always infer whether a variable is an index or a tensor identifier by context. In practice, we never need this many, and searching a space that includes up to 26 4-dimensional tensors is obviously impractical. This section addresses how we initially reduced this search space. Namely, by predicting the dimensions of the tensors in order to refine the grammar.

```
1  PROGRAM      ::=  TENSOR "=" EXPR
2  TENSOR       ::=  IDENTIFIER |
                     IDENTIFIER "(" INDEX-EXPR ")"
3  EXPR         ::=  TENSOR | CONSTANT | "(" EXPR ")" |
                     "−" EXPR | EXPR "+" EXPR |
                     EXPR "−" EXPR | EXPR "*" EXPR |
                     EXPR "/" EXPR
4  INDEX-EXPR   ::=  INDEX-VAR | INDEX-VAR "," INDEX-EXPR
5  INDEX-VAR    ::=  "i" | "j" | "k" | "l"
6  IDENTIFIER   ::=  LETTER ( LETTER | INTEGER )*
7  CONSTANT     ::=  INTEGER
8  INTEGER      ::=  DIGIT+
9  LETTER       ::=  "a" | "b" | ... | "z" |
                     "A" | "B" | ... | "Z"
10 DIGIT        ::=  "0" | "1" | "2" | ... | "9"
```

Figure 5.4: The grammar for TACO expression in Extended Backus–Naur form, defining the syntax for tensor expressions, identifiers, constants, and basic arithmetic expressions. The $^*$ symbol denotes Kleene star, indicating zero or more repetitions of the preceding element, while $^+$ denotes Kleene plus, requiring one or more occurrences.

### *Predicting tensor dimensions:*

To accurately predict tensor dimensions for a given program, we combine insights from a language model (LLM) with static code analysis. Static analysis is used to predict the left-hand side (LHS) tensor dimensions of an expression, while the LLM is used to predict dimensions for the right-hand side (RHS). Static analysis, by analysing the source code, can determine precisely the dimensions for the LHS tensor, but cannot do the same for the RHS, so we fall back on heuristics learned by the LLM for the RHS.

**Definition 4** (Dimension list)**.** *We define a dimension list $L_\tau$ to be a list of integers $(d_1, d_2, d_3, ...)$ where $d_i$ is the dimension for the $i^{th}$ unique tensor in the template $\tau$. We use $L[i]$ to indicate accessing the $i^{th}$ element of the dimension list $L$, and $|L|$ to indicate the length of the list $L$. We list the dimensions of*

*constants and variables as* 0.

For example, the list [1, 2, 4] indicates that the first tensor in the expression has 1 dimension, the second tensor has 2 dimensions, and the third tensor has 4 dimensions.

**Dimension prediction for RHS tensors using LLM:**

Given a set of templatized solutions

$$\{\tau_1, \dots \tau_{10}\} \in \mathcal{T},$$

generated by the LLM, we compute the dimension list for each candidate solution, giving a set of lists

$$L_{\mathcal{T}} = \{L_{\tau_1}, \dots, L_{\tau_{10}}\}$$

. We then filter this list by length and remove any list that has a length less than the maximum length, giving the filtered set

$$L_{\mathcal{T}}^* = \{l \in L_{\mathcal{T}} \mid |l| \geq |L_{\tau_i}| \text{ for all } L_{\tau_i} \in L_P\mathcal{T})\}.$$

Finally, we return the list that appears most frequently in the filtered set, i.e.,

$$L_{\tau} = \arg \max_l |l \in L_{\mathcal{T}}^*|.$$

From here on, we denote this predicted list *L*, and refer to this as the predicted dimension list.

**Integrating static analysis for LHS tensors.** We use static program analysis to examine the original program AST and predict the LHS dimension. We apply a dataflow analysis to recover the dimensions in the array accesses to recover the original dimensionality. For standard array accesses, e.g., *a*(*i*, *j*), we simply count the number of variables used to index the base pointer. However, it is common that C programs access multi-dimensional elements using affine linear expressions on index variables. In such cases,

we use array delinearization [78] to recover the standard array access form and predict the dimensionality by counting the number of indexing variables.

Additionally, some applications use explicit pointer arithmetic to iterate over arrays. We implement array recovery [32] to retrieve array access expressions from pointers and then apply delinearization and analyse the indexing expression. In case the output variable is not accessed through any memory indexing operation, we assume it is a scalar and predict zero-dimensionality.

As the left-hand side tensor necessarily appears first in the expression, we replace $L[1]$ with the predicted dimension for the first tensor from the static analysis.

### *Generating the context-free grammar:*

Given a dimension list, we wish to generate a grammar that ensures we only enumerate combinations of indices required to make all possible tensor expressions that match the predicted dimensions (or, at least, reduce this space as far as possible without increasing the complexity of the grammar significantly). For instance, if the dimension list is $[0, 1, 3]$, and at least one of the predicted solutions is `a = b(i) + c(i,j,k)`, we will modify the grammar to fix the production rule for the first tensor to restricted to `a`, and ensure for any remaining tensor that appear in the expression, the grammar can enumerate `b(i)` and `c(i,j,k)`, `c(i,k,j)`, `c(j,i,k)`, `c(k,i,j)`, `c(j,k,i)`, `c(k,j,i)`. This permits all possible combinations of indexing of the tensors of the predicted dimensions, allowing the index used for `b(i)` to be repeated in any position when indexing c.

Formally, we define this grammar generator as a set of constraints reasoning over the predicted dimension list $L$ and the set of templates $\mathscr{T}$, which,

if true, indicate that the production rule should be included in the grammar. We use `[c]`$r_i$ to denote that a production $r_i$ is included within the grammar if c is true. $i(P)$ denotes the number of unique index variables in the set of programs $\mathcal{T}$. Rules without a constraint automatically appear in the grammar.

```
1                PROGRAM  ::=  TENSOR1 "=" EXPR
2                EXPR     ::=  TENSOR | EXPR OP EXPR
3                OP       ::=  "+" | "-" | "*" | "/"
4   [L[1] = 0]   TENSOR1  ::=  "a"
5   [L[1] = 1]   TENSOR1  ::=  "a(i)"
6   [L[1] = 2]   TENSOR1  ::=  "a(i,j)"
7                   ...
8   [L[2] = 0]   TENSOR   ::=  "b" | "Const"
9   [L[2] = 1]   TENSOR   ::=  "b(i)" | [i(P) > 1]"b(j)" |
                             [i(P) > 2]"b(k)" |...
10                  ...
11  [L[2] = 2]   TENSOR   ::=  "b(i,j)" | "b(j,i)" |
                             [i(P) > 2]"b(i,k)" |...
12                  ...
13  [L[3] = 0]   TENSOR   ::=  "c" | "Const"
14  [L[3] = 1]   TENSOR   ::=  "c(i)" | [i(P) > 1]"c(j)" |
                             [i(P) > 2]"c(k)" |...
15                  ...
```

For every element in the dimension list, we add a new tensor id, `a`, `b`, `c` ..., and index it with the number of index variables that correspond to the element in the dimension list. For an element $i$, where $L[i] = n$, and for a set of candidate programs where $i(\mathcal{T}) = m$, we add a production rule for every possible way of choosing $n$ indices from $m$ index variables. We then remove any production rules that could not be used for parsing any candidate $\tau \in \mathcal{T}$, e.g., if no template in $\mathcal{T}$ contains a 2-dimensional tensor indexed with the same index variable twice, we will remove `b(i,i)`. An example generated template grammar is shown in Figure 5.5.

```
1 PROGRAM    ::=  TENSOR1 "=" EXPR
2 TENSOR1    ::=  "a(i)"
3 EXPR       ::=  TENSOR | CONSTANT | EXPR OP EXPR
4 TENSOR     ::=  "b(i,j)" | "b(j, i)" | "b(i, k)" | …
                  "c(i)" | "c(j)" | "c(k)"
5 CONSTANT   ::=  "Const"
6 OP         ::=  "+" | "−" | "*" | "/"
```

Figure 5.5: An example generated template grammar, for the dimension list $[1, 2, 1, 0]$, with a maximum of 3 unique indices appearing in the candidate LLM solutions.

### 5.4.2   Assigning Probabilities to the Grammar

We now wish to assign a probability to each production rule in the grammar of TACO templates according to their frequency in the left-most derivations of the candidate solutions.

Given a set of templatized solutions

$$\mathcal{T} \in \mathcal{L}(G_{template}),$$

we calculate a weight for each rule $r_i \in R$ as the number of times that rule appears in the left-most derivations of the programs. That is,

$$w[r_i] = \sum_{\tau_i \in \mathcal{T}} |r_i| \in D_{\tau_i},$$

where $|r_i|$ is the number of times $r_i$ appears in the derivation $D_{\tau_i}$. These weights reflect the usage frequency of each tensor with specific indices in the expressions provided by the LLM. Note that, for any production rules used to replace the tensor nonterminal symbols, e.g., 1DTENSOR, which do not appear in any of the candidate solutions, we assign a default weight of 1. This assignment ensures that these combinations are considered during the synthesis process with a lower priority.

Using the weights calculated for both operators and tensors, we construct the corresponding probabilistic Context-Free Grammar ($pCFG_\tau$) by

normalising the weights into probabilities. For each non-terminal symbol $\alpha$, the probability of applying the production rule $\alpha \rightarrow \beta$ is calculated by the equation in Section 4.4.1.

## 5.5   Searching the Template Space

We present two algorithms for searching the space of TACO templates. The first is based on a weighted $A^*$ search in the literature [57, 64], which searches the grammar of TACO templates in a top-down manner. We extend this algorithm to incorporate a penalty score that accounts for known syntactic constraints on the target solution. The second is an adapted version of $A^*$, which combines bottom-up search with $A^*$ heuristics.

### 5.5.1   Top-Down Weighted $A^*$

Algorithm 9 outlines the top-down weighted $A^*$ search with penalties. The search operates over a probabilistic Context-Free Grammar derived from large language model (LLM) outputs. The $A^*$ search in this chapter is different from Chapter 4, whose $A^*$ doesn't have a penalty function. It maintains a queue of partial templates, represented as abstract syntax trees, which we can think of as the frontier of its search. This initially contains just the start symbol of the grammar. At each iteration, it must determine which of the partial templates should be further explored, which it does based on the cost of the path taken to reach those partial templates and an estimate of the cost required to extend the path all the way to the goal. The goal is, ultimately to find a complete template that we believe is likely to satisfy the specification, that is a template can generate a program $p_t$ such that

$$\forall \vec{x}.p_t(\vec{x}) = p_s(\vec{x}).$$

Thus, when choosing which partial template to further explore, the algorithm chooses the template with the minimum $f(x)$, where $x$ is the partial template, defined as:

$$f(x) = c(x) + g(x) + \mathscr{X}(x),$$

where $c(x)$ calculates the accumulated cost from the start node $S$ to current node $x$. $g(x)$ is the heuristic estimate of the minimal cost to complete the expression from $x$ to a goal node (a template we believe is likely to satisfy the specification), and $\mathscr{X}(x)$ is a penalty term for expressions that violate domain-specific syntactic constraints. These are calculated as follows:

The accumulated cost $c(x)$ is calculated as the sum of the costs of the production rules applied along the path to $x$:

$$c(x) = \sum_{r_i \in D_x} -\log_2 \mathbb{P}[r_i],$$

where $D_x$ is the sequence of production rules used to reach the node $x$ and $\mathbb{P}[r_i]$ is the probability of production rule $r_i$. This cost function transforms probabilities into additive costs, suitable for the $A^*$ search.

The heuristic function $g(x)$ estimates the minimal additional cost required to complete the partial expression at node $x$ to a full expression. It is defined as:

$$g(x) = \begin{cases} 0 & \text{if } x \in \Sigma^*, \\ -\sum_{x_i \in V} \log_2 h(x_i) & \text{otherwise,} \end{cases}$$

where $x_i$ are the non-terminal symbols in the partial expression $x$, and $h(a)$ is the maximal probability of deriving any terminal string from non-terminal $a$. The value $h(a)$ is defined recursively for each non-terminal $a$:

$$\forall a \in V, \quad h(a) = \max_{a \to \beta \in R} \left( \mathbb{P}[a \to \beta] \times \prod_{\beta_i \in \beta} h(\beta_i) \right),$$

with the base case $h(a) = 1$, if $a \in \Sigma$. This equation represents the maximal probability of deriving a terminal string from $a$, accounting for the probabilities of production rules and the maximal probabilities of its components.

The penalty function $\mathscr{X}(x)$ this assigns additional costs to expressions that do not meet specific domain criteria. This function can be formalised as follows:

$$\mathscr{X}(x) = \begin{cases} \sum_{a \in A} \mathscr{X}_a(x) & \text{if } x \text{ violates criterion } a, \\ 0 & \text{otherwise,} \end{cases}$$

There are 5 criteria $\{a_1, \dots a_5\} \in A$, and their penalty scores are defined as follows (note that an infinite penalty score effectively means these expressions will never be considered):

- $\mathscr{X}_{a_1}(x) = 10$, where $a_1$ is violated if the grammar includes a constant expression, the length of $x$ exceeds 3, and $x$ either 1) contains fewer than 2 tensors with index `i` or 2) lacks a constant expression. This penalty biases the search against expressions with multiple tensors but inadequate index variety or missing constants.

- $\mathscr{X}_{a_2}(x) = 100$, where $a_2$ is violated iff $x$ is not the same length as the length of the dimension list.

- $\mathscr{X}_{a_3}(x) = \infty$, where $a_3$ is violated if the tensor symbols in $x$ are not in alphabetical order by order of first appearance. This penalty rule avoids enumerating templates that are structurally identical, and therefore can be instantiated into identical sets of programs.

- $\mathscr{X}_{a_4}(x) = \infty$, where $a_4$ is violated if $x$ is a complete template (no non-terminal symbols), and repeatedly applies addition, subtraction, or division operations on the same tensor.

- $\mathscr{X}_{a_5}(x) = \infty$, where $a_5$ is violated if $x$ is a complete template (no non-terminal symbols), and employs fewer than half of the operations defined in the grammar.

---

**Algorithm 9** Top-Down Enumerator

---

1: **procedure** ENUMERATE($pCFG_\tau$)

2:     $Q \leftarrow \{(0, pCFG_\tau.S)\}$ ▷ Initialize queue with start symbol of grammar

3:     **while** $Q \neq \varnothing$ **do**

4:         $(f, x) \leftarrow Q.pop()$                    ▷ Remove template with minimal $f$

5:         **if** depth($x$) > maxDepth **then**

6:             **continue**                    ▷ Skip if maximum depth exceeded

7:         **end if**

8:         **if** $x \in \Sigma^*$ **then**                    ▷ If no non-terminals remain in $x$

9:             $S \leftarrow$ VALIDATE($x$)                    ▷ Try to instantiate $x$

10:            **if** $S \neq \perp$ **then**

11:                **if** VERIFY($x.\{S\}$) **then**

12:                    **return** $x.\{S\}$

13:                **end if**

14:            **end if**

15:        **end if**

16:        **for** $x'$. s.t. $(x \xrightarrow{r} x' \wedge r \in pCFG_\tau.R)$ **do** ▷ Iterate over all possible expansions of $x$

17:            $Q \leftarrow Q \cup \{c(x') + g(x') + \mathscr{X}(x'), x'\}$

18:        **end for**

19:     **end while**

20:     **return** Failure        ▷ Return Failure if no valid expression is found

21: **end procedure**

---

**Search.** The search is shown in Algorithm 9. The algorithm keeps a queue

of expressions in a queue, which you can consider to be the frontier of the search. At each iteration, it selects the expressions with the lowest total score $f$ from the queue. If the expression is a complete expression, i.e., it contains only terminal symbols from the grammar, we then send this to the validation procedure described in Section 5.6. If the expression is a partial expression, the leftmost non-terminal of the expression is expanded according to all applicable production rules in the grammar, creating a new template for each production rule. These new expressions are all added to the queue and the process is repeated.

We set a depth limit of 6, and if any expression exceeds this depth, it is discarded. We calculate depth as the depth of the maximum child in the abstract syntax tree, excluding index expressions, e.g., `b(i)` and `c(i,j)` are both expressions of depth 1, and `b(i) + c(i,j)` is an expression of depth 2.

### 5.5.2 Bottom-Up Weighted $A^*$

The algorithm presented in the previous section takes a top-down approach to enumerating through the search space. This has advantages over bottom-up search, namely that it is known to find longer programs faster than bottom-up search, which is biased towards shorter programs. Nevertheless, recent work has shown that guided bottom-up search can produce promising results [13]. To that end, we develop a new $A^*$ inspired bottom-up search algorithm, which we term bottom-up $A^*$. The bottom-up search, shown in Algorithm 10 constructs expressions incrementally by starting with basic tensors and systematically combining them using operators, following a probabilistic context-free grammar. Again, the algorithm maintains a queue of expressions, and uses the same combination of cost, estimated cost to reach a goal state and the penalty function to determine

which expression to expand first.

One key difference in the bottom-up search is the way we generate the template grammar. For the bottom-up search, where the production rules only permit extending an expression by adding an operator and a new tensor to the end, effectively forcing the algorithm to enumerate programs shortest first. The grammar generator, given a predicted dimension list $L$, and a function $i(\mathscr{T})$ which calculates the number of unique indices in $\mathscr{T}$, is shown below:

```
 1            PROGRAM    ::=  TENSOR1 "=" EXPR
 2            EXPR       ::=  TENSOR2 TAIL1
 3            TAIL1      ::=  ε | [|L| > 2] OP TENSOR3 TAIL2
 4            TAIL2      ::=  ε | [|L| > 3] OP TENSOR4 TAIL3
               ...
 5            OP         ::=  "+" | "-" | "*" | "/"
 6  [L[1] = 0]TENSOR1   ::=  "a"
 7  [L[1] = 1]TENSOR1   ::=  "a(i)"
 8  [L[1] = 2]TENSOR1   ::=  "a(i,j)"
 9  [L[2] = 0]TENSOR2   ::=  "b" | "Const"
10  [L[2] = 1]TENSOR2   ::=  "b(i)" | [i(P) > 1]"b(j)" |
                             [i(P) > 2]"b(k)"
11  [L[2] = 2]TENSOR2   ::=  "b(i,j)" | "b(i,j)" | "b(j,i)" |
                             [i(P) > 2]"b(i,k)" |...
12  [L[3] = 0]TENSOR3   ::=  "c" | "Const"
13  [L[3] = 1]TENSOR3   ::=  "c(i)" | [i(P) > 1]"c(j)" |
                             [i(P) > 2]"c(k)" |...
14            ...
```

An example of generated grammar is shown in Figure 5.6.

Weights and probabilities over the grammar are then calculated as described in Section 5.4.2.

```
1  PROGRAM     ::=  TENSOR1 "=" EXPR
2  TENSOR1     ::=  "a"
3  EXPR        ::=  1DTENSOR TAIL1
4  TAIL1       ::=  ε | OP 2DTENSOR TAIL2
5  TAIL2       ::=  ε | OP 1DTENSOR
6  2DTENSOR    ::=  "c(i, j)" | "c(j, i)" | "c(i, k)" | …
7  1DTENSOR    ::=  "b(i)" | "d(k)"
```

Figure 5.6: An example generated template grammar, for the dimension list [0, 1, 2, 1], with a maximum of 3 unique indices appearing in the candidate LLM solutions. The rules for each tensor index expression include all possible permutations of indices. The symbol $\varepsilon$ denotes the empty string.

The search algorithm maintains a queue of partial programs, as with the top-down search, which is initialised with the start symbol from the grammar. At each iteration, the program with the minimum cost function, as before, is popped from the queue, expanded, and all the resulting programs are added to the queue. The total cost function $f(x)$ for each partial expression $x$, is again defined as:

$$f(x) = c(x) + g(x) + \mathscr{X}(x),$$

where $c(x)$ is calculated as before. In the bottom-up search, we use a simplified estimate of the cost to complete the program, $g(x)$, which is defined as:

$$g(x) = \sum_{i=k}^{|L|} m(L[i + 1]),$$

where $k$ is the current number of tensors in $x$, $L$ is the predicted dimension list, and $m(d)$ is the minimal cost to add a tensor of dimension $d$. The minimal cost $m(d)$ is computed as follows, where Tensors($d$) is the list of tensors in the grammar of dimension $d$, and $\mathscr{P}[t]$ is the maximum probability of any production rule in the grammar which adds the tensor of

dimension $d$:

$$m(d) = -\log_2\left(\max_{t \in \text{Tensors}(d)} \mathbb{P}[t]\right).$$

The penalty function is calculated as before, but with the criteria $\{b_1, b_2\} \in B$ defined as:

- $\mathscr{X}_{b_1}(x) = 100$, where $b_1$ is violated if the tensor symbols in $x$ are not in alphabetical order by order of first appearance.

- $\mathscr{X}_{b_2}(x) = \infty$, where $b_2$ is violated if $x$ contains at least as many tensors as predicted by the dimension list, and it uses fewer than half the operations available in the grammar, and 0 otherwise.

The bottom-up search uses fewer penalty criteria than the top-down search because the construction of the grammar encapsulates a number of these criteria already (for instance, the tensors are enumerated by predicted dimension list order).

The main difference between the top-down and the bottom-up search is that the bottom-up grammar is generated in a way that, at each intermediate step, a complete program can be inferred from the partial program and checked against the specification. Every time an expression is dequeued from the queue, if the expression contains a tail nonterminal symbol, e.g., `TAIL1`, `TAIL1`; we can remove the tail nonterminal symbol to give a complete template (i.e., a template that contains no non-terminal symbols). We can then return this template to the template validator. If it fails validation, we will re-append the nonterminal symbol to the end of the expression and generate new expressions by expanding the non-terminal using all applicable production rules. The new expressions are added into the queue.

---

**Algorithm 10** Bottom-Up Enumerator

---

1: **procedure** ENUMERATE($pCFG_\tau, L$)

2:      $Q \leftarrow \{(0, pCFG_\tau.S)\}$      ▷ Priority queue initialized with start node

3:      **while** $Q \neq \varnothing$ **do**

4:          $(f, x) \leftarrow Q.pop()$      ▷ Remove template with minimal $f$

5:          **if** $|tensors(x)| = |L|$ **then**      ▷ If number of tensors in $x$ is the predicted number

6:              **if** $x \notin \Sigma^*$ **then**

7:                  $x \leftarrow$ RemoveTail($x$)      ▷ Remove any tail nonterminal symbol

8:              **end if**

9:              $S \leftarrow$ VALIDATE($x$)      ▷ Try to instantiate $x$

10:              **if** $S \neq \perp$ **then**

11:                  **if** VERIFY($x.\{S\}$) **then**

12:                      **return** $x.\{S\}$      ▷ Return instantiated template

13:                  **end if**

14:              **end if**

15:          **end if**

16:          **for** $x'$. s.t. $(x \xrightarrow{r} x' \wedge r \in pCFG_\tau.R)$ **do** ▷ Iterate over all possible expansions of $x$

17:              $Q \leftarrow Q \cup \{c(x') + g(x') + \mathcal{X}(x'), x'\}$

18:          **end for**

19:      **end while**

20:      **return** Failure      ▷ Return Failure if no valid expression is found

21: **end procedure**

---

## 5.6   Validation

Once the synthesiser produces a complete template $\tau$, we wish to check whether it can generate a program $p_t$ that satisfies the requirement

$$\forall \vec{x} \, p_s(\vec{x}) = p_t(\vec{x}).$$

Since checking this universally quantified formula is expensive, we first generate a set of tests in the form of input-output examples.  This set of examples $\langle I, O \rangle$ is a list of input-output pairs where

$$I = \langle x_1 \mapsto v_1, ..., x_n \mapsto v_n \rangle$$

is a map that binds the $\tau$ input arguments

$$\vec{x} = (x_1, ..., x_n)$$

to concrete values $(v_1, ...v_n)$ randomly generated and $O = (o_1, ..., o_n)$ is the corresponding output produced when we execute $p_s$ on the elements from $I$.

The TACO templates generated during the synthesis phase contain symbolic placeholders for tensors and constants.  To validate a candidate $\tau$, we build a set

$$S = (s_1, ..., s_m)$$

of substitutions $s \mapsto x$ that map the symbolic symbols $s$ in *tau* to input arguments $x$.  We iterate through all possible permutations of $S$, where tensor symbols are mapped to concrete tensor inputs, and constants are mapped to a set of constants

$$C = (c_1, ..., c_m)$$

containing the constant values present in the source code of $p_s$.  We can then generate a concrete program

$$p_t = \tau.\{S\},$$

and execute $p_t$ on the input-output examples in $\langle I, O \rangle$. If any instantiated concrete program satisfies all the input-output examples, we return this to the next stage of verification. We use $S$ to assign concrete values to TACO symbols and run $P_T$ to check its output. When building $S$, we rule out invalid substitutions based on the type of arguments and TACO symbols. More specifically, we discard substitutions that try to bind tensor symbols with dimension > 1 to scalars and vice versa. If the validator succeeds, it returns the valid substitution $S$, if not it returns $\perp$ to indicate there was no valid substitution.

We explore all possible valid substitutions until we find a substitution $s^*$ that satisfies $\phi$. This validation process returns a tuple $\langle P_T, s^* \rangle$ that is given as input to the verification phase.

Figure 5.7 shows a subset of the substitutions set $S$ given the program in Benchmark 8 and the TACO candidate $P_T$ produced by the synthesiser, `a(i) = b(i,j) * c(j)`. Each substitution binds a symbol in the right-hand side of $P_T$, i.e., b and c to one of the inputs of `function`. The substitutions with a ✗ mark next to it are invalid, since they contain unsound bindings. For example, substitution $S_3$ binds c, a 1-dimensional tensor to $N$, which is a scalar. Such substitutions are discarded and the valid ones are tested to run the program until we find one in which $P_T$ satisfies the specification. In this example, the correct substitution is $S_5$, which binds b and c to arguments `Mat1` and `Mat2` respectively.

$S_1 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{Mat1} \rangle$

$S_2 : \langle b \mapsto \texttt{Mat2}, c \mapsto \texttt{Result} \rangle$

$S_3 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{N} \rangle$ ✗

$S_4 : \langle b \mapsto \texttt{N}, c \mapsto \texttt{Mat1} \rangle$ ✗

$S_5 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{Mat2} \rangle$

$S_6 : \langle b \mapsto \texttt{Mat2}, c \mapsto \texttt{N} \rangle$ ✗

$\vdots$

$S_m : \langle b \mapsto \texttt{Result}, c \mapsto \texttt{Mat2} \rangle$

$\rightarrow$

$S_1 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{Mat1} \rangle$

$S_2 : \langle b \mapsto \texttt{Mat2}, c \mapsto \texttt{Result} \rangle$

$S_5 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{Mat2} \rangle$

$\vdots$

$S_m : \langle b \mapsto \texttt{Result}, c \mapsto \texttt{Mat2} \rangle$

$\rightarrow$

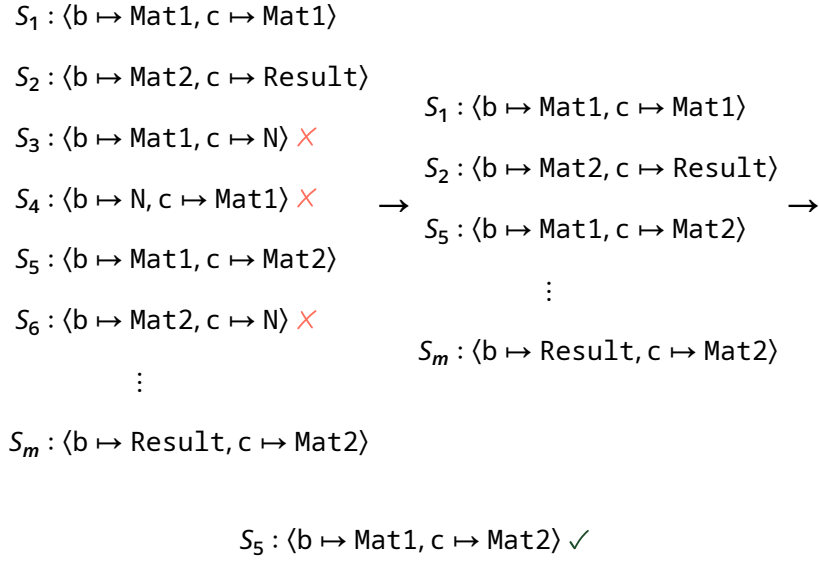$S_5 : \langle b \mapsto \texttt{Mat1}, c \mapsto \texttt{Mat2} \rangle$ ✓

Figure 5.7: A set of possible substitutions for the TACO program `a(i) = b(i,j) * c(j)` and the inputs from the legacy program in Benchmark 8. We discard invalid substitutions and try the valid ones until we find one that satisfies the specification.

## 5.7   Verifier

We verify the correctness of a synthesised TACO program using bounded model checking. We compile both the original C and the TACO program to a common language within the MLIR compiler infrastructure [56]. Given a TACO program $T$ and a substitution $S$ returned by the validator, we create NumPy code based on the indexing expressions of $T$ and replace its variables for the concrete values specified in $s^*$. If the model checker fails to verify equivalence with the tuple $\langle T, s^* \rangle$, we return to the validation step and keep exploring different substitutions until we find one that satisfies the specification and passes verification. We then use the JAX compiler [18] to lower the NumPy code to MLIR.

From the MLIR files, we automatically generate C programs that create non-deterministic inputs, execute the original C and TACO code on copies of those inputs, and assert that the outputs are identical. We give this C program as input to CBMC [54], a bounded model checker for C, that verifies said assertion holds for all possible inputs up to a certain bound.

Floating-point equivalence is both challenging to verify and, in many cases, undesirable. For instance, many compiler optimisations simply do not preserve floating-point optimisations in order to achieve runtime speed-ups. For this reason, we extend CBMC to support rational datatypes, and verify equivalence using rational datatypes.

## 5.8   Evaluation

To evaluate STAGG and its various components, we compare its performance against several established techniques on a diverse suite of queries. The query set includes 10 artificial examples and 67 real-world problems

(61 derived from codebases reported in the literature [70] and 6 from the C++ based inference code of Llama [68].)

STAGG is implemented using an extended version of CBMC 6.3.1 with CVC5 version 1.0.5 as the underlying SMT solver. To generate initial candidate solutions, we use GPT-4 with the temperature set to 1.0. A timeout of 60 minutes is applied to each query. All experiments are conducted on a system equipped with an 11th Gen Intel Core i5-1135G7 processor, 16 GiB of RAM, and running Ubuntu 22.04.5 LTS. Additional configuration details, including grammar refinements and penalty modifications, are provided in the subsequent sections.

We compare the following approaches: (**STAGG** $^{TD}$) our approach, using the top-down $A^*$ search described in Section 5.5.1; (**STAGG** $^{BU}$) our approach, using the bottom-up search described in Section 5.5.2; (**C2TACO**) An enumerative synthesis tool for lifting C to TACO code [70]. We compare to C2TACO both with and without the domain-specific heuristics; (**Tenspiler**) An enumerative synthesis tool based on the verified lifting framework [93]; (**LLM only**) A baseline approach that employs a large language model (GPT-4) to directly generate candidate solutions without additional heuristic-driven refinement or search. In addition, we perform ablation studies to evaluate the contribution of several components of STAGG. Namely, the grammar refinement; the probabilities of the grammar; and the penalty functions.

### 5.8.1 Performance Comparison of STAGG to the State-of-the-Art Solvers.
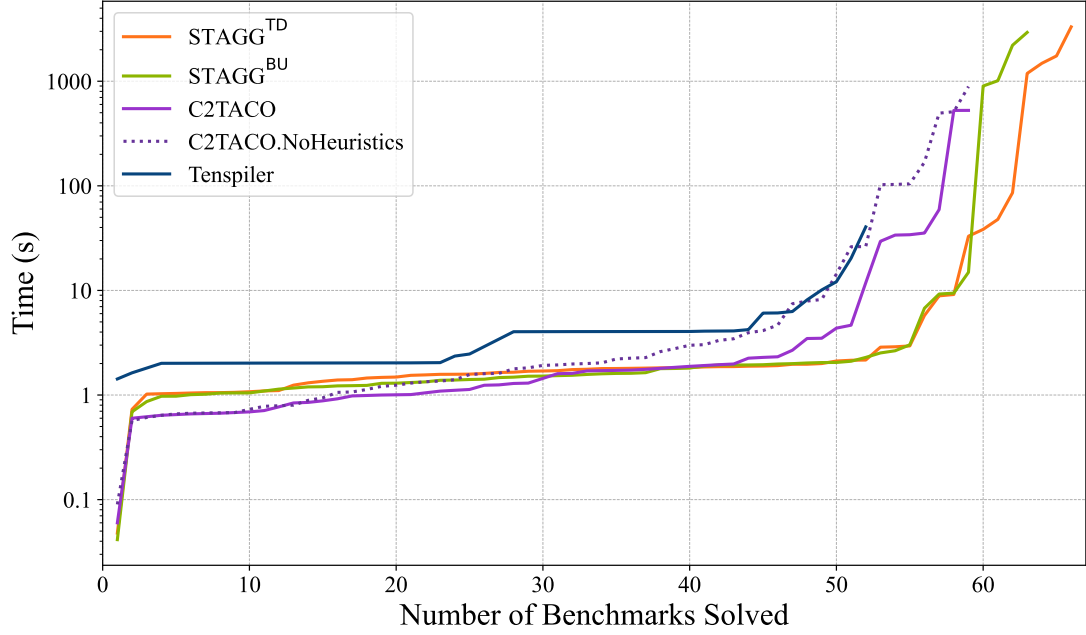


Figure 5.8: Cactus plot showing the number of benchmarks solved (*x*-axis) vs. time (*y*-axis, logarithmic) on the 67 real-world benchmarks. Each line corresponds to a different synthesiser, and the point at which each line indicates how many benchmarks the synthesiser solved before the time.

Figure 5.8 depicts the cumulative time each method takes over the 67 real-world benchmarks, and Figure 5.9 shows the success rate of different techniques. We compare STAGG to C2TACO on the full set of 77 benchmarks in Table 5.1. STAGG$^{TD}$ solves 76 benchmarks, compared to C2TACO which solves 67. STAGG$^{TD}$ solves all the benchmarks that C2TACO can solve, with an average solving time of 3.19*s*, compared to C2TACO's 21.15*s*. STAGG$^{BU}$ solves 73 benchmarks, and solves 66/67 of the benchmarks that C2TACO solves, with an average solving time of 2.11*s* on the mutually solved benchmarks. C2TACO without the domain heuristics enabled is significantly slower.

We are only able to run Tenspiler on the 67 real-world benchmarks,
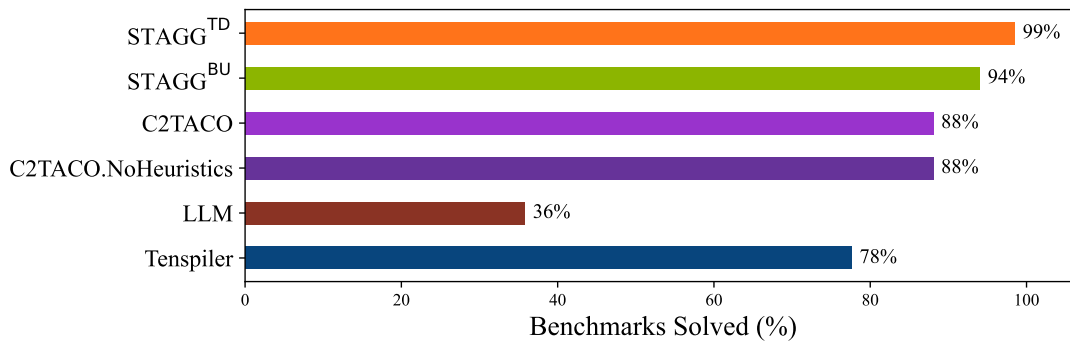
Figure 5.9: Success rates of different approaches on the set of 67 real-world benchmarks.

where it solves 52. $STAGG^{TD}$ solves all 52 benchmarks that Tenspiler can solve, with an average time of 3.45$s$ compared to Tenspiler's average time of 4.56$s$. $STAGG^{BU}$ solves only 50/52 of the benchmarks that Tenspiler can solve, but with an average time of 2.03$s$. This comprehensively answers RQ1: STAGG outperforms the state-of-the-art solvers, both in terms of coverage and speed.

## 5.8.2 Performance Comparison of Top-Down vs Bottom-Up Search.

Our results show that, while $STAGG^{TD}$ solves more benchmarks than $STAGG^{BU}$, $STAGG^{BU}$ is faster on commonly solved benchmarks (for queries solved by both, $STAGG^{BU}$ achieves a lower average solving time (98.81 seconds) compared to $STAGG^{TD}$ (108.83 seconds)), and it enumerates fewer candidates. It has one big disadvantage though, which is that it can only expand expressions by appending to the previous expression, rather than by expanding nodes on the left-hand side of the AST. In particular, this means it cannot solve benchmarks that require expressions with more balanced Abstract Syntax Trees or benchmarks that contain parentheses.

Table 5.1: Comparison of benchmark-solving performance across different methods: The table reports the number of benchmarks solved (#), average solving time (time in seconds), and attempts across various benchmarks. The benchmarks are categorised into real-world benchmarks (67 in total), real-world + artificial benchmarks (77 in total), benchmarks solved by C2TACO, and benchmarks solved by Tenspiler. STAGG $^{TD}$ and STAGG $^{BU}$ demonstrate superior solving capabilities, solving more benchmarks overall compared to C2TACO and Tenspiler, with STAGG $^{BU}$ achieving the fastest solving times for benchmarks solvable by C2TACO and Tenspiler. The results highlight the efficacy of STAGG over existing methods.

| | Real-World (67) | | Real-World + Artificial (77) | | | Solved by C2TACO | | Solved by Tenspiler | |
|---|---|---|---|---|---|---|---|---|---|
| Methods | # | time | # | time | attempts | # | time | # | time |
| STAGG$^{TD}$ | 66 | 121.88 | 76 | 106.13 | 44.55 | 67 | 3.19 | 52 | 3.45 |
| STAGG$^{BU}$ | 63 | 113.86 | 73 | 98.81 | 35.62 | 66 | 2.11 | 50 | 2.03 |
| LLM | 24 | 2.61 | 34 | 2.59 | 1.62 | 31 | 2.57 | 20 | 2.72 |
| C2TACO | 59 | 22.57 | 67 | 21.15 | 18.45 | 67 | 21.15 | 50 | 23.69 |
| C2TACO.NoHeuristics | 59 | 43.08 | 67 | 49.41 | 48.81 | 67 | 49.41 | 50 | 43.76 |
| Tenspiler | 52 | 4.56 | | | | | | 52 | 4.56 |

Table 5.2:  Impact of penalty rules on performance over 77 benchmarks (real-world + artificial).  The table compares the number of benchmarks solved (#), the percentage of benchmarks solved (%), and the average solving time (time in seconds) for various configurations of STAGG. Removing penalty rules (e.g., Drop(A), Drop(B)) reduces the number of solved benchmarks and influences solving times. While STAGG $^{TD}$ and STAGG $^{BU}$ achieve high solving rates with the full penalty rules, dropping specific penalties often results in faster solving times but at the cost of reduced solving capability, as it failed solving complex benchmarks.

|  | Real-World + Artificial (77) | | |
| --- | --- | --- | --- |
| Methods | # | % | time |
| STAGG$^{TD}$ | 76 | 98.70% | 106.13 |
| STAGG$^{TD}$.Drop(A) | 71 | 92.21% | 7.21 |
| STAGG$^{TD}$.Drop(a1) | 72 | 93.51% | 79.24 |
| STAGG$^{TD}$.Drop(a2) | 75 | 97.40% | 91.66 |
| STAGG$^{TD}$.Drop(a3) | 72 | 93.51% | 21.01 |
| STAGG$^{TD}$.Drop(a4) | 75 | 97.40% | 90.58 |
| STAGG$^{TD}$.Drop(a5) | 75 | 97.40% | 83.34 |
| STAGG$^{BU}$ | 73 | 94.81% | 98.81 |
| STAGG$^{BU}$.Drop(B) | 70 | 90.91% | 68.18 |
| STAGG$^{BU}$.Drop(b1) | 71 | 92.21% | 48.95 |
| STAGG$^{BU}$.Drop(b2) | 70 | 90.91% | 68.75 |

Table 5.3: Performance comparison of different methods and grammar configurations over 77 benchmarks (real-world + artificial). The table shows the number of benchmarks solved (#), the percentage of benchmarks solved (%), the average solving time (time in seconds), and the number of synthesis attempts. STAGG $^{TD}$ and STAGG $^{BU}$ outperform C2TACO variants in solving more benchmarks. Variations of STAGG demonstrate the impact of grammar refinement, where dropping penalty rules (Drop(A), Drop(B)) or using alternative configurations (e.g., EqualProbability, LLM-Grammar) affects the solving capability, time, and attempts.

| Methods | Real-World + Artificial (77) | | | |
| | # | % | time | attempts |
|---|---|---|---|---|
| STAGG $^{TD}$ | 76 | 98.70% | 106.13 | 44.55 |
| STAGG $^{TD}$.Drop(A) | 71 | 92.21% | 7.21 | 13.65 |
| STAGG $^{TD}$.EqualProbability | 73 | 94.81% | 28.14 | 37.27 |
| STAGG $^{TD}$.LLMGrammar | 52 | 67.53% | 3.77 | 5.25 |
| STAGG $^{TD}$.FullGrammar | 69 | 89.61% | 91.15 | 874.29 |
| STAGG $^{BU}$ | 73 | 94.81% | 98.81 | 35.62 |
| STAGG $^{BU}$.Drop(B) | 70 | 90.91% | 68.18 | 10.07 |
| STAGG $^{BU}$.EqualProbability | 74 | 96.10% | 180.31 | 62.78 |
| STAGG $^{BU}$.LLMGrammar | 52 | 67.53% | 2.74 | 2.60 |
| STAGG $^{BU}$.FullGrammar | 68 | 88.31% | 96.57 | 259.35 |
| LLM | 34 | 44.16% | 2.59 | 1.62 |
| C2TACO | 67 | 87.01% | 21.15 | 18.45 |
| C2TACO.NoHeuristics | 67 | 87.01% | 49.41 | 48.81 |
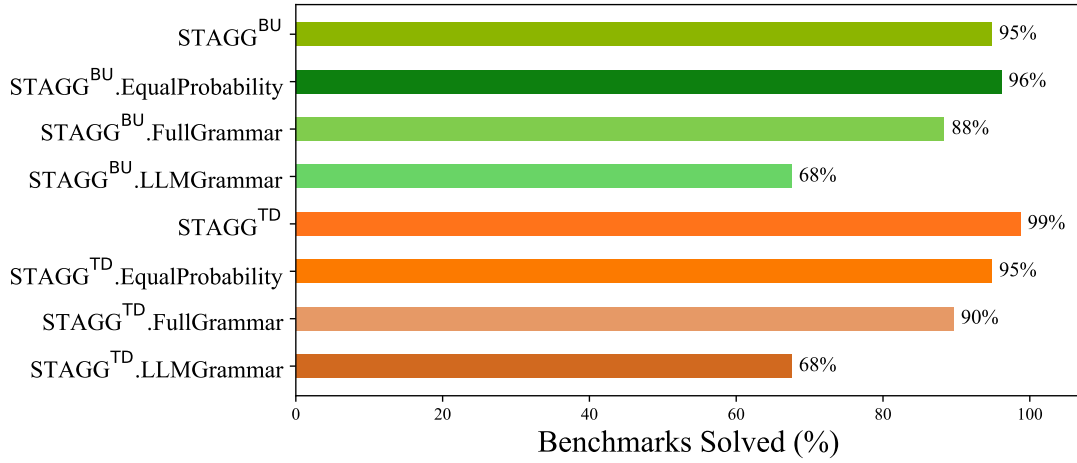
Figure 5.10: Impact of different grammar configurations in STAGG on success rates across all 77 benchmarks.

### 5.8.3 Contribution of the Penalty Functions.

Table 5.2 shows the decline in performance for both STAGG $^{TD}$ and STAGG $^{BU}$ approaches when individual penalty rules are removed. As each penalty rule is dropped, the number of queries solved decreases, highlighting the importance of these rules in achieving high query-solving efficiency.

### 5.8.4 Contribution of Grammar Refinement and Probabilities.

Figure 5.10, 5.11 and Table 5.3 show the performance of difference configurations of STAGG: `EqualProbability` uses the refined grammar but replaces all probabilities in the generated pCFG with equal probabilities; `FullGrammar` uses the full TACO grammar in Figure 5.4 with equal probabilities; `LLMGrammar` uses the full TACO grammar in Figure 5.4 with probabilities learned from the LLM responses. All these configurations use the penalty functions. Thus, in order to compare the contribution of the grammar refinement, we can compare the performance of `LLMGrammar`, which
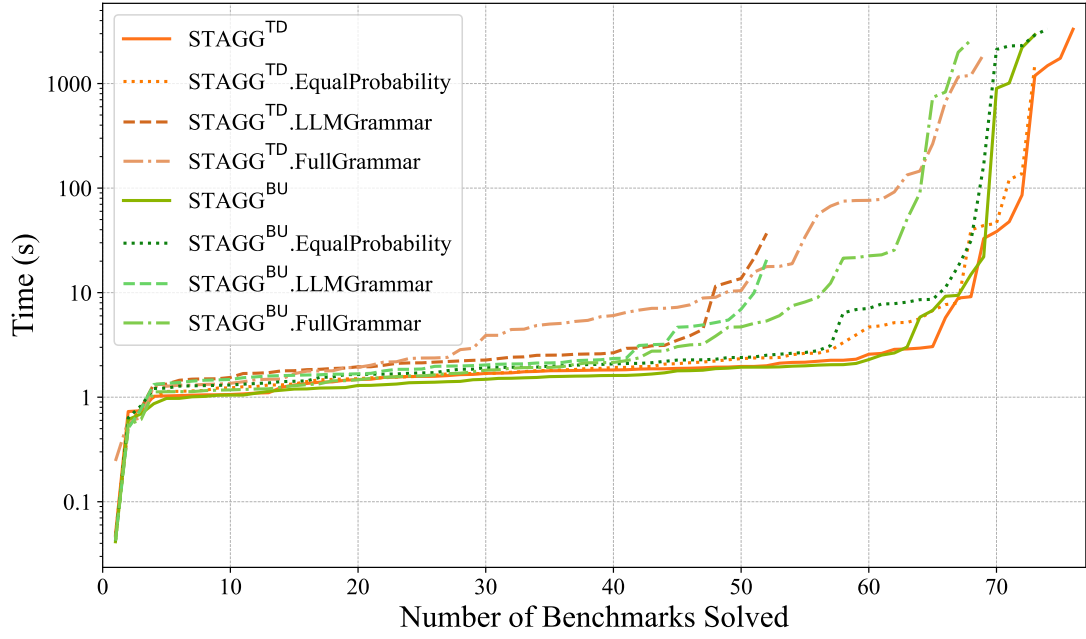
Figure 5.11: The performance of difference configurations of STAGG on all 77 benchmarks.

uses learned probabilities but no refinement, to STAGG. Dropping the refinement here, results in us solving 31% fewer benchmarks. If we compare `FullGrammar` to `EqualProbability`, we can see that grammar refinement has less impact when learned probabilities are not used, but still results in a significant number of benchmarks being dropped.

In order to compare the contribution of the probabilities, we can compare the performance of `EqualProbability` to STAGG, where we note that using equal probabilities on the refined grammar results in an increase in the number of benchmarks solved for STAGG$^{TD}$, and an increase in solving speed for STAGG$^{BU}$, although neither is as impactful as the grammar refinement. In fact, the comparison between `FullGrammar` and `LLMGrammar` demonstrates that learned probabilities can have a negative impact if they are used in a grammar that is not general enough. Thus, our answer to RQ4 and RQ5 is that grammar refinement in combination with probabilities has a bigger impact on performance than either compon-

ent part, but the refinement alone is more powerful than the probabilities alone.

## 5.9   Conclusions

This chapter presented STAGG, a novel approach that combines LLMs and program synthesis to lift legacy tensor code to DSLs. We use a set of LLM responses to infer a probabilistic context-free grammar that drives an enumerative search over the space of possible solutions. Our technique successfully lifts 99% of a large suite of benchmarks with an average lifting time of 3.19 seconds, outperforming existing state-of-the-art lifters in terms of coverage and synthesis time. Additionally, STAGG is able to automatically learn a search space, and it does not rely on any pre-defined heuristics. Future work will focus on expanding our technique to application domains other than tensor computation.

# Chapter 6

# Related Work

## 6.1   SyGuS Solvers

Many state-of-the-art SyGuS solvers are based on enumerative synthesis
[6, 95, 57, 42] and use clever heuristics to improve the search speed. Closest
to our work in Chapter 4 and Chapter 5 is Euphony [57], which accelerates
enumerative syntax-guided synthesis by biasing the search towards likely
programs using a learned probabilistic model.  Specifically, it employs an
$A^*$ search algorithm that enumerates candidate programs in order of de-
creasing probability, effectively treating the negative log probability of a
program as the search cost. The search is guided by a probabilistic higher-
order grammar  [17] trained on previously solved synthesis benchmarks,
so program derivations are ranked by their learned likelihood rather than
assumed uniform.  This approach significantly prunes the search space
by prioritising derivations that mirror patterns from the training solutions,
leading to faster discovery of correct programs.  However, Euphony ' s ef-
fectiveness hinges on the availability of a library of known solutions to train
its model, and obtaining such domain-specific training data can be chal-
lenging.  In contrast, newer methods avoid this requirement by guiding

the search with pre-trained large language models (LLMs), which provide probabilistic grammar guidance without the need for custom training corpora. Weighted grammars have also been used to guide programming by example [73], and to encode syntactic objectives [40], for instance, for optimising the length of solutions.

The SyGuS framework itself [3] formalises synthesis as the task of finding a program from a user-specified grammar that satisfies a logical specification. Almost all synthesis algorithms use oracles to give feedback to the synthesis process [47, 46]. The majority of these use semantic oracles, which give feedback on the meaning of the program. For example, the counterexample-guided inductive synthesis (CEGIS) paradigm [101]. CEGIS frames synthesis as an iterative loop between a learner and a verifier. The learner proposes a candidate program consistent with the examples seen so far, while the verifier checks the candidate against the full specification. If the candidate is incorrect, the verifier returns a counterexample input that exposes the failure. This counterexample is then added to the learner's example set, refining the next candidate. The loop continues until either a correct program is found or the search space is exhausted. This iterative loop underpins many later solvers. A refinement of the approach, CEGIS($\mathcal{T}$) [2], integrates deductive reasoning from theory solvers directly into the loop, enabling the synthesiser to solve for constants or prune infeasible candidates within background theories. Similarly, conflict-driven learning techniques such as Feng et al.[31] strengthen the search by learning constraints from failed partial programs, thereby eliminating entire regions of the search space. EUSolver [6] pioneered a divide-and-conquer approach, decomposing problems into smaller sub-expressions and unifying them into complete solutions. Reynolds et al. [96] introduce the first program synthesis engine, which takes a different path by

embedding synthesis directly inside an SMT solver. Candidate functions are represented as algebraic datatypes, and the solver explores them via counterexample-guided quantifier instantiation (CEGQI), which systematically proposes instantiations for the unknown function and refines them with counterexamples. This "synthesis in the solver" approach leverages SMT machinery for pruning and theory reasoning, and has proven highly competitive across SyGuS competition tracks. DryadSynth [42] exemplifies a more recent trend of hybrid solvers that combine deductive reasoning with concurrent enumerative search. It decomposes specifications into subproblems and applies lightweight logical inference to rule out infeasible branches before attempting enumeration. Multiple threads then explore different parts of the search space in parallel, exchanging deductions to accelerate convergence. This cooperative model scales to domains such as strings and bit-vectors where pure enumeration or deduction alone is insufficient.

Machine learning techniques have been deployed to improve the efficiency of enumerative synthesis. Parsert et al. [85] use reinforcement learning with Monte Carlo tree search to guide grammar-based enumeration. A policy network selects grammar rules, while a value network estimates whether a partial program is likely to lead to a solution, and both are trained using automatically generated SyGuS benchmarks. Chen et al.[26] instead integrate reinforcement learning directly into a deductive synthesiser: partial programs are checked with an SMT solver, and infeasible candidates trigger counterexample-guided policy updates, so the policy improves online during synthesis. Bunel et al., [22] address the program aliasing problem in neural synthesis by combining a grammar-constrained decoder with policy-gradient training, rewarding the model for any correct program that satisfies the specification rather than only reproducing ref-

erence solutions. Morton et al., [74] propose grammar filtering, where a neural classifier predicts which production rules are relevant for the given specification, allowing the solver to discard irrelevant rules and focus on a much smaller, problem-specific grammar. Together, these methods demonstrate how learned heuristics can be used either to prioritise expansions, to learn from deductive counterexamples, or to prune the grammar itself before search begins. Balog et al. [10] synthesised array manipulation programs from I/O examples using a feedforward neural network (FNN) to build a probabilistic distribution over the target language. During the search, the synthesis algorithm expands partial programs based on the probabilities predicted by the FNN for the given I/O specification. Neural-guided synthesis has also been applied to solve string manipulation tasks by [79, 99], inductive logic programming [99], dataframe [15] and tensor [98, 76] processing, and code transpilation [71] using distinct models. SketchAdapt [77] uses a model to produce a program sketch as a starting point and completes said sketch through symbolic enumeration.

## 6.2   Large Language Models

LLMs, such as GPT-4 [81] and CoPilot [36], have demonstrated impressive capabilities in generating code and assisting in diverse programming tasks with natural language and input-output specifications [20, 21]. In [9], it is shown that performance scales log-linearly with model size, improves with fine-tuning, and can be further enhanced by interactive natural language feedback, though models still struggle with semantic grounding and execution understanding. Jigsaw [44] augments black-box LLMs like GPT-3 and Codex with program analysis and synthesis modules, combining natural language intent and I/O examples to synthesise Pandas code.

It corrects common LLM errors (e.g., variable references, argument mismatches, semantic mistakes) via AST-to-AST transformations and learns from user feedback, leading to significantly higher accuracy than raw LLM outputs. Nevertheless, their tendency to produce hallucinations, factually incorrect, or contextually inappropriate outputs poses challenges to users [88, 97, 87].

SYMLLM [50] is a framework that recovers from LLM synthesis failures by decomposing incorrect programs into prefix and suffix subprograms, then recursively solving the resulting subproblems. CodeARC [108] introduces an interactive benchmark where LLM agents query hidden target functions with inputs, invoke a differential testing oracle, and iteratively refine synthesised code. Closest to our work in Chapter 4 is Kamath et al., who use LLMs to synthesise loop invariants directly [48]. Our work also demonstrates that LLMs are surprisingly good at synthesising invariants, but additionally addresses how to use LLMs in other formal synthesis problems and when they cannot find the solution in one shot. LEMUR [111] introduces a hybrid framework that combines the high-level reasoning ability of LLMs with the precise low-level reasoning of automated verifiers. In this approach, LLMs propose candidate invariants and intermediate proof goals, while automated reasoners validate or repair them within a sound proof calculus. Clover [103] presents a complementary paradigm of closed-loop verifiable code generation, in which LLMs generate code together with natural-language documents and formal annotations. Jha et al. [45] and Song et al. [102] integrate an LLM into a CEGIS loop, but unlike our work, the entire synthesis phase is carried out by the LLM, which prevents them from leveraging the combined strengths of enumerative solving and LLMs. Wang et al. [106] propose grammar prompting, where an LLM uses a BNF grammar to enforce syntactic constraints during generation, en-

abling highly structured languages such as DSLs to be generated with minimal data. Xander [91] is a neurosymbolic architecture for SQL generation that explores multiple candidate queries using best-first search, applies symbolic checks on partial and complete queries, and repairs them when needed. Similarly, Tao et al. [105] address trust concerns in LLM-generated code by restricting the search space to predefined "safe" BNF grammars, avoiding programs with risky vulnerabilities. Another work close to our work in Chapter 4 is HYSYNTH [13]. HYSYNTH first samples candidate programs from an LLM and extracts their production statistics to build a probabilistic context-free grammar (pCFG). This pCFG is then used to weight rules in a bottom-up enumerative synthesiser, effectively biasing search toward components that the LLM deems relevant, thereby reducing the search space and speeding up synthesis.

Pre-trained LLMs have also been used for code lifting, which will be introduced in Chapter 5. LLMLift [16] applies LLMs directly for verified lifting, leveraging GPT-4 [81] to guess candidate solutions and loop invariants to prove equivalence, with feedback given to the LLM to correct mistakes. This approach is highly effective but depends on the LLM's ability to repair itself from feedback, which is not always trivial in complex domains [80]. Oxidizer [113] is a modular translation framework that applies feature mapping rules to guide LLMs through subtle cross-language differences and performs type-compatibility checks before validating I/O equivalence. AlphaTrans [43] addresses the complexity of migrating from one language to another, which is infeasible with a naive "feed it to GPT-4" approach due to context limits and error accumulation. Pan et al. [84] systematically evaluate general-purpose and code-focused LLMs across many language pairs, introducing a taxonomy of translation bugs and showing that most errors arise from syntactic or semantic mismatches or from viol-

ating target-language constraints. These findings suggest that while LLMs hold promise for automated translation, reliable results require complementary techniques such as program analysis or prompt engineering.

Close to our work in Chapter 5 is C2TACO [70]. C2TACO is a synthesis tool that generates TACO code from I/O examples. It implements a bottom-up enumerative algorithm, and it uses code analysis to restrict the search space of programs. mlirSynth [19] also has a similar approach, but it lifts tensor programs across different MLIR dialects. In both methods, correctness is asserted using only I/O testing, while our work in Chapter 5 performs bounded model checking to verify that the lifted programs are equivalent to their original counterpart. A different synthesis method was used in Tenspiler [93], which employs symbolic synthesis to generate programs in six different tensor DSLs. Tenspiler builds verification conditions and loop invariants to prove that the lifted program is equivalent to the original one. Unlike our work, which learns how to explore the search space in a fully automated way, all those techniques require hard-coded heuristics to make the search space tractable.

Another approach to lift code is API matching, in which source code is replaced by optimised library routines to improve performance. Examples in the tensor domain include KernelFaRer [28], which focuses on general matrix multiplication (GEMM), and ATC [72], which targets both GEMM and convolutions. SpEQ [55] introduces a method of translating sparse linear algebra codes to optimised targets using equality saturation applied to LLVM IR. However, these approaches are often tailored to specific APIs and are not portable. Our work in Chapter 5 leverages the great learning capabilities of Large Language Models to infer the search space, which makes our technique extensible to different targets and to more unrestricted back-ends such as DSLs.

## 6.3   Automatic Prompt Generation

Manual prompting refers to human-designed prompt strategies. For example, Chain-of-Thought prompting [109] or Knowledge prompting [67] are clever prompting strategies designed by humans that often improve the performance of LLMs across reasoning tasks. In Chapter 3, we differ from manual prompting in that we take these strategies as inputs and learn to select the best combination of them for a given task.

Continuous and discrete prompting both refer to automated prompting techniques. Continuous prompting techniques, like Prefix-Tuning [59], aim to learn domain and task-specific vectors that are then used to guide LLMs to better performance. These vectors can usually not be represented by a sequence of tokens, so continuous prompting is not often considered interpretable.

In contrast, discrete prompting techniques optimise the text input to the LLM. Some discrete prompting techniques, like PRewrite [53], RLPrompt [29], TEMPERA [114] and GRIPS [90], take a starting prompt and search for a new version of the prompt that outperforms the original. To understand the starting prompt, these techniques usually use a second language model in the loop. Other discrete prompting techniques, like Auto-Prompt [100], Liu et al. [66] and Lu et al. [69], generate prompts using templates. AutoPrompt uses LLM gradients to learn task-specific keywords that are then included in the prompt. Liu et al. select examples to include in a few-shot prompt template. In a similar vein, Lu et al. optimise the order of examples.

Our work in Chapter 3 is most like the latter category of discrete prompts. We rely on templates and existing prompts to offer inexpensive prompt optimisation.

Another related work in Chapter 3 is "portfolio solver", which refers to any algorithm that deploys multiple solvers or solver configurations on a given problem. Wintersteiger et al. [110] uses the concurrent portfolio solver for a given SMT formula, run multiple Z3 instances configured with different heuristics in parallel, sharing learned clauses so that the fastest solver can terminate the search early. MedleySolver [89] predicts a sequence of SMT solvers for a given input query to deploy based on minimising Par-2 score (a proxy for time with a penalty for timeouts). PAK-UCB [41] frames prompt-aware model selection as a contextual bandit problem, which learns prompt-dependent performance using kernel-based predictors, with random Fourier features for efficiency. OPTS [7] chooses a prompt design strategy via a bandit mechanism to improve downstream performance.

# Chapter 7

# Conclusions

Program synthesis sits at the intersection of programming languages and artificial intelligence, aiming to automatically generate correct programs from high-level specifications. This thesis has explored a hybrid approach that bridges formal synthesis and Large Language Models (LLMs), seeking to combine the scalability and adaptability of LLMs with the formal guarantees of program synthesis.

## 7.1   Summary of Contributions

This thesis makes four key contributions that advance the state-of-the-art in program synthesis:

- **CYANEA, the online solver selection:**

  We developed a contextual multi-armed bandit framework that dynamically selects between LLM–prompt pairs and symbolic solvers based on features of the synthesis task. CYANEA consistently outperformed the best individual solver in terms of success rate and computational efficiency.

- **LLM-guided probabilistic grammars:** We proposed two complementary techniques, *pCFG-synth* and *iLLM-synth*, that mine probabilistic guidance from incorrect LLM outputs and integrate this information into enumerative search. While *pCFG-synth* achieved the highest overall success rate (solving 80.1% of SyGuS benchmarks), *iLLM-synth* demonstrated the feasibility of interactive prompting and dynamic grammar refinement during search.

- **STAGG: guided tensor lifting:** We applied the hybrid synthesis framework to lift low-level tensor kernels into high-level DSLs. STAGG combined LLM-driven heuristics with weighted A* search, achieving 99% correctness on real-world tasks and up to a 12× speedup over previous state-of-the-art methods.

## 7.2   Implications and Impact

This thesis demonstrates that LLMs can be effectively integrated into program synthesis pipelines as heuristic engines rather than standalone solvers. By transforming incorrect or partial LLM outputs into actionable probabilistic grammars, we achieved significant improvements in both scalability and correctness compared to purely symbolic or purely statistical methods. The proposed frameworks are model-agnostic, ensuring compatibility with future LLM architectures without requiring retraining.

Furthermore, the success of STAGG in lifting tensor kernels highlights the potential of this approach for real-world software engineering tasks, including code transpilation, optimisation, and DSL migration.

## 7.3  Limitations and Future Work

While the hybrid frameworks presented in this thesis demonstrate clear benefits, several limitations remain:

- **Benchmark diversity:**

  The evaluation primarily relied on SyGuS benchmarks and dense tensor lifting tasks, which, while standard in the field, may not fully capture the diversity of real-world synthesis challenges. Future work should expand the evaluation to broader domains.

- **LLM prompt engineering:**

  Although the frameworks leverage pre-trained LLMs, prompt engineering in CYANEA is limited to selecting from a fixed, manually constructed set of prompt styles. While this selection improves robustness over single-prompt baselines, it still relies on hard-coded templates. Future work could explore automatic construction and continual refinement of the prompt library.

- **Model bias and hallucination:**

  While the frameworks exploit the strengths of LLMs, they also inherit model biases and the tendency to hallucinate or produce semantically invalid code. Developing more reliable correction mechanisms remains an open challenge.

## 7.4  Applicability to Other Domains

By combining LLM-derived heuristics with enumerative search techniques, this thesis demonstrates that it is possible to build synthesis systems that

are both efficient and reliable. This hybrid approach opens the door to scalable, formally correct program synthesis that leverages the best of both statistical and symbolic reasoning. As large language models continue to evolve, integrating them effectively into symbolic frameworks holds promise for the next generation of intelligent software development tools.

In this thesis, we have demonstrated its effectiveness in diverse synthesis settings, from SyGuS benchmarks to tensor algebra lifting, showing that a learned heuristic from LLMs can reliably guide enumerative or repair-based search. This principle applies equally well to domains that expose a well-defined domain-specific language (DSL) and verifiable semantics. For example, in structured query languages such as SQL or formula-driven environments like Excel, the space of target programs can be concisely described by a grammar or template. Given such a DSL, an LLM can propose plausible candidate functions that capture the user intent, and when these initial guesses fail, an enumerative or constraint-guided search can repair or complete them toward correctness. Hence, we believe, by providing a compact DSL and a mechanism for automatic verification, the hybrid framework can be adapted to many real-world applications where correctness and interpretability are essential, ranging from data transformation and spreadsheet automation to robotic task planning and scientific computation.

# Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2016.

[2] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17, 2013.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Abhishek Udupa. Syntax guided synthesis competition. `https://sygus-org.github.io`, 2024. Accessed: 2024-01-16.

[5] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 319–336. Springer, 2017.

[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[7] Rin Ashizawa, Yoichi Hirose, Nozomu Yoshinari, Kento Uchida, and Shinichi Shirakawa. Bandit-based prompt design strategy selection

improves prompt optimizers. In Wanxiang Che, Joyce Nabende, Eka-
terina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of
the Association for Computational Linguistics: ACL 2025*, pages 20799–
20817, Vienna, Austria, July 2025. Association for Computational Lin-
guistics.

[8] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Scha-
pire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*,
32(1):48–77, 2002.

[9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma,
Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael
Terry, Quoc Le, and Charles Sutton. Program synthesis with large
language models. *arXiv preprint arXiv:2108.07732*, 2021.

[10] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian No-
wozin, and Daniel Tarlow. Deepcoder: Learning to write programs.
*arXiv preprint arXiv:1611.01989*, 2016.

[11] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian No-
wozin, and Daniel Tarlow. Deepcoder: Learning to write programs.
In *ICLR (Poster)*. OpenReview.net, 2017.

[12] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer,
Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mo-
hamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner,
Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5:
A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume
13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer,
2022.

[13] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla,
Taylor Berg-Kirkpatrick, and Nadia Polikarpova. HYSYNTH: context-
free LLM approximation for guiding program synthesis. *CoRR*,
abs/2405.15880, 2024.

[14] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare
Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule,
Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability
- Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and
Applications*, pages 1267–1329. IOS Press, 2021.

[15] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and
Ion Stoica. Autopandas: neural-backed generators for program
synthesis. *Proceedings of the ACM on Programming Languages*,
3(OOPSLA):1–27, 2019.

[16] Sahil Bhatia, Jie Qiu, Sanjit A Seshia, and Alvin Cheung. Can llms
perform verified lifting of code? *Technical Report*, 2024.

[17] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2933–2942. JMLR.org, 2016.

[18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[19] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael FP O'Boyle. mlirsynth: Automatic, retargetable program raising in multi-level ir using program synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 39–50. IEEE, 2023.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[21] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712*, 2023.

[22] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[23] Sarah E. Chasins and Julie L. Newcomb. Using SyGuS to synthesize reactive motion plans. In *SYNT@CAV*, volume 229 of *EPTCS*, pages 3–20, 2016.

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[25] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.

[26] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[27] Cristina David, Daniel Kroening, and Matt Lewis. Using program synthesis for program analysis. In *LPAR*, pages 483–498. Springer, 2015.

[28] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. Kernelfarer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)*, 18(3):1–22, 2021.

[29] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. *arXiv preprint arXiv:2205.12548*, 2022.

[30] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro Sandoval, Rakia Segev, Eric Weiner, and Robert Keller. Program language translation using a grammar-driven tree-to-tree model. *ICML*, 2018.

[31] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435. ACM, 2018.

[32] Björn Franke and Michael O'boyle. Array recovery and high-level transformations for dsp applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(2):132–162, 2003.

[33] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In *ESEC/SIGSOFT FSE*, pages 633–645. ACM, 2022.

[34] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In *TACAS (3)*, volume 11429 of *Lecture Notes in Computer Science*, pages 156–166. Springer, 2019.

[35] Philip Ginsbach, Bruce Collie, and Michael FP O'Boyle. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 179–190, 2020.

[36] GitHub and OpenAI. GitHub Copilot, 2021.

[37] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on apple silicon, 2023.

[38] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[39] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968.

[40] Qinheping Hu and Loris D'Antoni. Syntax-guided synthesis with quantitative syntactic objectives. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 386–403. Springer, 2018.

[41] Xiaoyan Hu, Ho fung Leung, and Farzan Farnia. Pak-ucb contextual bandit: An online learning approach to prompt-aware selection of generative models and llms, 2025.

[42] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.

[43] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

[44] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *ICSE*, pages 1219–1231. ACM, 2022.

[45] Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *MILCOM 2023-2023 IEEE Military Communications Conference (MILCOM)*, pages 944–949. IEEE, 2023.

[46] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224, 2010.

[47] Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54:693–726, 2017.

[48] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models, 2023.

[49] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *SIGPLAN Not.*, 51(6):711–726, 2016.

[50] Ruhma Khan, Sumit Gulwani, Vu Le, Arjun Radhakrishna, Ashish Tiwari, and Gust Verbruggen. Llm-guided compositional program synthesis, 2025.

[51] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948, 2017.

[52] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *OOPSLA*, 2017.

[53] Weize Kong, Spurthi Amba Hombaiah, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. Prewrite: Prompt rewriting with reinforcement learning. *arXiv preprint arXiv:2401.08189*, 2024.

[54] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[55] Avery Laird, Bangtian Liu, Nikolaj Bjørner, and Maryam Mehri Dehnavi. SpEQ: Translation of sparse codes using equivalences. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[56] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[57] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 436–449, New York, NY, USA, 2018. Association for Computing Machinery.

[58] Cheng Li, Jindong Wang, Yixuan Zhang, Kaijie Zhu, Wenxin Hou, Jianxun Lian, Fang Luo, Qiang Yang, and Xing Xie. Large language models understand and can be enhanced by emotional stimuli. *arXiv preprint arXiv:2307.11760*, 2023.

[59] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[60] Yixuan Li, Lewis Frampton, Federico Mora, and Elizabeth Polgreen. Online prompt selection for program synthesis. *Proceedings of the*

*AAAI Conference on Artificial Intelligence*, 39(11):11282–11289, Apr. 2025.

[61] Yixuan Li, José Wesley de Souza Magalhães, Alexander Brauckmann, Michael F. P. O'Boyle, and Elizabeth Polgreen. Guided tensor lifting. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[62] Yixuan Li, Federico Mora, Elizabeth Polgreen, and Sanjit A Seshia. Genetic algorithms for searching a matrix of metagrammars for synthesis. *arXiv preprint arXiv:2306.00521*, 2023.

[63] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *CAV (2)*, volume 14682 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2024.

[64] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *Computer Aided Verification*, CAV 2024, pages 280–301, Cham, 2024. Springer Nature Switzerland.

[65] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, pages 639–646. Citeseer, 2010.

[66] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[67] Jiacheng Liu, Alisa Liu, Ximing Lu, Sean Welleck, Peter West, Ronan Le Bras, Yejin Choi, and Hannaneh Hajishirzi. Generated knowledge prompting for commonsense reasoning. *arXiv preprint arXiv:2110.08387*, 2021.

[68] llamacpp. `https://github.com/leloykun/llama2.cpp/`, 2024. Accessed: 2024-01-19.

[69] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786*, 2021.

[70] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2023, page 42–56, New York, NY, USA, 2023. Association for Computing Machinery.

[71] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022.

[72] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael FP O' Boyle. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 85–97, 2023.

[73] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195. PMLR, 2013.

[74] Kairo Morton, William T. Hallahan, Elven Shum, Ruzica Piskac, and Mark Santolucito. Grammar filtering for syntax-guided synthesis. In *AAAI*, pages 1611–1618. AAAI Press, 2020.

[75] In Jae Myung. Tutorial on maximum likelihood estimation. *J. Math. Psychol.*, 47(1):90–100, February 2003.

[76] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. Predictive synthesis of api-centric code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 40–49, 2022.

[77] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[78] Michael FP O'Boyle and Peter MW Knijnenburg. Integrating loop and data transformations for global optimization. *Journal of Parallel and Distributed Computing*, 62(4):563–590, 2002.

[79] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

[80] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2023.

[81] OpenAI. GPT-4 technical report. *arXiv*, pages 2303–08774, 2023.

[82] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. LLM is like a box of chocolates: the non-determinism of ChatGPT in code generation. *arXiv preprint arXiv:2308.02828*, 2023.

[83] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The sygus language standard version 2.1. `https://sygus.org/assets/pdf/SyGuS-IF_2.1.pdf`, 2021.

[84] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[85] Julian Parsert and Elizabeth Polgreen. Reinforcement learning and data-generation for syntax-guided synthesis. In *AAAI*, pages 10670–10678. AAAI Press, 2024.

[86] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037. Curran Associates, Inc., 2019.

[87] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[88] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2785–2799, 2023.

[89] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. Medleysolver: Online SMT algorithm selection. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 453–470. Springer, 2021.

[90] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. *arXiv preprint arXiv:2203.07281*, 2022.

[91] Henrijs Princis, Cristina David, and Alan Mycroft. Enhancing sql query generation with neurosymbolic reasoning. *Proceedings of*

the AAAI Conference on Artificial Intelligence, 39(19):19959–19968, Apr. 2025.

[92] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations. *ECOOP*, 2024.

[93] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, 2024.

[94] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[95] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[96] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 198–216, Cham, 2015. Springer International Publishing.

[97] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at C: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2205–2222, Anaheim, CA, August 2023. USENIX Association.

[98] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–36, 2022.

[99] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

[100] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*, 2020.

[101] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

[102] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. LLM-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2998–3009, 2023.

[103] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. *arXiv preprint arXiv:2310.17807*, 2023.

[104] Weizhi Tang, Yixuan Li, Chris Sypherd, Elizabeth Polgreen, and Vaishak Belle. Hygenar: An llm-driven hybrid genetic algorithm for few-shot grammar generation. *arXiv preprint arXiv:2505.16978*, 2025.

[105] Ning Tao, Anthony Ventresque, Vivek Nallur, and Takfarinas Saber. Grammar-obeying program synthesis: A novel approach using large language models and many-objective genetic programming. *Computer Standards & Interfaces*, 92:103938, 2025.

[106] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 65030–65055. Curran Associates, Inc., 2023.

[107] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.

[108] Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. Codearc: Benchmarking reasoning capabilities of llm agents for inductive program synthesis, 2025.

[109] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[110] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. A concurrent portfolio approach to smt solving. In

Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 715–720, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[111] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870*, 2023.

[112] Leiqi Ye, Yixuan Li, Guy Frankel, Jianyi Cheng, and Elizabeth Polgreen. Unlocking hardware verification with oracle guided synthesis. In *The 25th Conference on Formal Methods in Computer-Aided Design*, pages 235–245. TU Wien Academic Press, 2025.

[113] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. Scalable, validated code translation of entire projects using large language models. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[114] Tianjun Zhang, Xuezhi Wang, Denny Zhou, Dale Schuurmans, and Joseph E Gonzalez. Tempera: Test-time prompting via reinforcement learning. *arXiv preprint arXiv:2211.11890*, 2022.

[115] Mingqian Zheng, Jiaxin Pei, and David Jurgens. Is "a helpful assistant" the best role for large language models? A systematic evaluation of social roles in system prompts. *CoRR*, abs/2311.10054, 2023.