Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus

YUFAN CAI, Ningbo University, China, National University of Singapore, Singapore
ZHÉ HÓU, Griffith University, Australia
DAVID SANAN, Singapore Institute of Technology, Singapore
XIAOKUN LUAN, Peking University, China
YUN LIN, Shanghai Jiao Tong University, China
JUN SUN, Singapore Management University, Singapore
JIN SONG DONG, National University of Singapore, Singapore

Recently, the rise of code-centric Large Language Models (LLMs) has reshaped the software engineering world with low-barrier tools like Copilot that can easily generate code. However, there is no correctness guarantee for the code generated by LLMs, which suffer from the hallucination problem, and their output is fraught with risks. Besides, the end-to-end process from specification to code through LLMs is a non-transparent and uncontrolled black box. This opacity makes it difficult for users to understand and trust the generated code. Addressing these challenges is both necessary and critical. In contrast, program refinement transforms high-level specification statements into executable code while preserving correctness. Traditional tools for program refinement are primarily designed for formal methods experts and lack automation and extensibility. We apply program refinement to guide LLM and validate the LLM-generated code while transforming refinement into a more accessible and flexible framework. To initiate this vision, we propose Refine4LLM, an approach that aims to: (1) Formally refine the specifications, (2) Automatically prompt and guide the LLM using refinement calculus, (3) Interact with the LLM to generate the code, (4) Verify that the generated code satisfies the constraints, thus guaranteeing its correctness, (5) Learn and build more advanced refinement laws to extend the refinement calculus. We evaluated Refine4LLM against the state-of-the-art baselines on program refinement and LLMs benchmarks. The experiment results show that Refine4LLM can efficiently generate more robust code and reduce the time for refinement and verification.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software notations and tools**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Program Refinement, Large Language Model, Program Synthesis

ACM Reference Format:
Yufan Cai, Zhé Hóu, David Sanan, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proc. ACM Program. Lang.* 9, POPL, Article 69 (January 2025), 33 pages. https://doi.org/10.1145/3704905

Authors' Contact Information: Yufan Cai, cai_yufan@u.nus.edu, Ningbo University, China, National University of Singapore, Singapore; Zhé Hóu, z.hou@griffith.edu.au, Griffith University, Australia; David Sanan, david.miguel@singaporetech.edu.sg, Singapore Institute of Technology, Singapore; Xiaokun Luan, luanxiaokun@pku.edu.cn, Peking University, China; Yun Lin, lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, China; Jun Sun, junsun@smu.edu.sg, Singapore Management University, Singapore; Jin Song Dong, dongjs@comp.nus.edu.sg, National University of Singapore, Singapore.

ACM 2475-1421/2025/1-ART69
https://doi.org/10.1145/3704905

1 Introduction

Challenges in LLM-based code generation. Large language models (LLMs) have recently advanced rapidly in mathematics, reasoning, and programming [52, 70]. Industrial products like GPT-4 [47] and Copilot [33] greatly assist programmers in coding-related tasks and have performed above the 50th percentile in programming competitions [48]. However, one of the critical challenges they face is the problem of *hallucination*, where the models generate plausible but factually incorrect information. Moreover, some user studies [27, 61] show that programmers find it hard to trust and debug the LLM-generated code as the generation procedure is opaque and out of control. Even worse, researchers found that over half of ChatGPT's answers to programming-related questions contain incorrect information [38], and more recently, there is mathematical proof that LLM's hallucination is inevitable [64]. We take the classic refinement example - square root algorithm shown in Figure 1. We show the code snippets generated by the latest LLMs OpenAI o1-preview [46], GPT-4 [47] and GitHub Copilot [33] with the prompt:

Find the square root of N within the error bound e.

The LLMs can generate **almost** correct code. However, these programs still contain some bugs. Both LLM-based GitHub Copilot and OpenAI GPT-4 generated code fall into infinite loops, while the OpenAI o1-preview model provides incorrect answers in some cases. In detail, the program generated by Copilot is wrong when the input $n < 1$. Mathematically, the choice of the variable `high` as the upper bound of the square root of N should be larger than $N+\frac{1}{4}$ because $\forall N > 0, (N+\frac{1}{4})^2 \geq N$. The GPT-4 generated code fails in some cases, like sqrt(5), because the variable x goes to a fixed point but does not terminate the loop due to float precision error. The code from the OpenAI o1-preview model sometimes gives incorrect answers when N is smaller than $(e/4)^2$ because x is initialized to a negative number.

Therefore, reliable LLMs for program generation remain an unresolved challenge. Current strategies primarily focus on guiding LLMs during input and verifying their outputs. Guiding LLMs involves prompting them with relevant information about the task to steer them toward more straightforward solutions within the LLM's capability. Recent works often employ informal heuristics, such as the chain-of-thought approach, to direct the reasoning processes of LLMs [63]. On the other hand, thoroughly verifying the deep learning models as a white box is still limited to small quantized neural networks, far away from LLMs [36]. The latest LLM verification methods rely on multiple LLMs to assess outputs through majority voting or consensus in natural language [1, 43, 68]. However, existing work [64] mathematically proves that it is impossible to eliminate hallucination by simply changing the prompts. They also show that an ensemble of multiple LLMs is essentially a single LLM and cannot eliminate the hallucination. In contrast, our Refine4LLM integrates the LLM and the symbolic AI systems CoqHammer[21] to guide LLMs with constraints and verify the generated code through program refinement and verification. Our approach mimics how humans use tools like calculators and code interpreters to solve complex problems that go beyond their immediate capabilities. Intuitively, we regard the LLMs as "constraint solvers", whose powerful extensibility and rich background knowledge shed light on the potential of automation for program refinement. Our program refinement can "assert" constraints that help debugging and "verify" constraints to establish the correctness of the generated code. This methodology represents a significant advancement in applying LLMs to program generation, moving beyond error reduction to achieving reliable and verifiable code.

Challenges in traditional program refinement. The refinement calculus [5, 15, 44, 55, 60] is a formalization of the stepwise refinement method of program construction. Program refinement involves specifying a program's required behavior through a non-executable specification, which

```
''' Github Copilot '''
def find_sqrt(n):
  e = 0.0000001
  low, high = 0, n
  mid = (low + high) / 2
  while abs(mid * mid - n) > e:
    if mid * mid > n:
      high = mid
    else:
      low = mid
    mid = (low + high) / 2
  return mid
find_sqrt(0.1) = Infinite Loop

''' GPT-4 '''
def find_sqrt(N):
  x = N if N > 1 else 1
  while x * x > N:
    x = (x + N/x) / 2
  return x
find_sqrt(5) = Infinite Loop
```

```
''' OpenAI o1-preview '''
def find_sqrt(N, e):
  x = math.sqrt(N) - e / 2.0
  if x**2 < N < (x + e)**2:
    return x
  else:
    x_lower, x_upper = x,  x + e

    while x_upper - x_lower > 1e-10:
      x_mid = (x_lower + x_upper) / 2.0
      if x_mid**2<N and (x_mid + e)**2>N:
        x = x_mid
        break
      elif x_mid**2 >= N:
        x_upper = x_mid
      else:
        x_lower = x_mid
    return x_mid

e = 0.01; N = (e**2/16)/4
find_sqrt(N, e) = -0.00125
```

Fig. 1. Three wrong programs generated by LLMs. The code from GitHub Copilot uses a wrong upper bound. The GPT-4 generated code sometimes makes the variable x fall into a fixed point. The code from the OpenAI o1-preview model sometimes gives incorrect answers when N is small.

is then transformed into an executable program via a series of correctness-preserving steps. However, this transformation, grounded in program refinement calculus, is predominantly performed manually, which is time-consuming and error-prone. The necessity of manual code writing makes the program refinement labor-intensive and challenging to automate. Therefore, integrating LLMs' code generation ability into the refinement process is a logical progression.

Outline. We implement the approach in an automated tool called Refine4LLM that combines the formal program refinement calculus with the informal LLM to refine the specification and generate verified code step by step. Section 2 introduces the background of program refinement. Section 3 illustrates a motivating example with the square root algorithm. Section 4 shows the overview of the tool Refine4LLM and the following Section 5 defines the formal specification language and program language. To enhance the efficiency of the program refinement, Refine4LLM adopts a learning strategy to build new refinement laws to reduce the depth of the refinement process shown in Section 6. In Section 7, we introduce the formal system part that automatically and formally transforms the specification based on the refinement law and generates the associated proof obligations. Refine4LLM combines automated theorem provers (ATPs) like Z3 [22] to verify the code and justify the choice of the refinement laws. In Section 8, we show the informal system part, including a top-down algorithm to split the high-level specification into sub-components and then a bottom-up algorithm to refine the sub-specifications and generate the code one by one. We evaluate Refine4LLM on the classic program refinement benchmarks [39] and LLM benchmarks - Humaneval dataset from OpenAI [47] and the EvalPlus dataset with more test cases [41] shown in Section 9. Overall, our approach leverages the creativity of LLMs and rigorous proof of formal methods to provide a reliable code generator. The approach is complementary to LLMs, automated

theorem provers, and traditional verification tools. To the best of our knowledge, Refine4LLM is the first framework that combines the LLM and program refinement techniques.

Contributions. The contributions of the paper are summarized below:

(1) A framework, Refine4LLM, for LLM aided automated program refinement, including a formal specification language L_{spec}, a programming language L_{pl} associated with our refinement calculus, and a verification strategy that verifies the generated code.
(2) A refinement law learning strategy to derive more advanced refinement rules to reduce the depth of the refinement process.
(3) A top-down splitting and bottom-up refining algorithm to build the library of program refinement for reducing the complexity of the specification.

2 Preliminaries

This section introduces the background knowledge and technical details of program refinement.

2.1 Notation

We follow the notations in Morgan's book on program refinement [44].

Specification. This work considers the formal specifications defined in first-order logic (FOL). A specification contains *variables*, a *precondition*, and a *postcondition*, in the form

$$variables : [precondition,\ postcondition]$$

Variables are the list of program variables; the precondition describes the program's initial states, and the postcondition describes the program's final states.

Program Refinement. Formally, the refinement relation is defined by the weakest preconditions of the related programs [14]. For program S and postcondition P, $wp(S, P)$ represents the weakest precondition where S is guaranteed to terminate in a state satisfying P. Program S_0 is refined by S_1 denoted as $S_0 \sqsubseteq S_1$, iff

$$\forall P, wp(S_0, P) \rightarrow wp(S_1, P) \tag{1}$$

which states that S_1 will preserve the total correctness of program S_0. The program refinement can be established in a linear sequence:

$$S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq S_3 ... \sqsubseteq S_n \tag{2}$$

which shows the refinement $S_0 \sqsubseteq S_n$ with the transitivity of the refinement relation.

2.2 The Basic Refinement Calculus

The refinement calculus is based on the weakest precondition semantics for programs from the literature [26]. Typically, the refinement process is a sequence of refinement law applications that refine formal specifications to a mixture of specifications and programs (mixed programs) and finally to only program code, as illustrated below:

$$specification \sqsubseteq mixed\ program \sqsubseteq \cdots \sqsubseteq mixed\ program' \sqsubseteq program.$$

The core refinement calculus listed in Morgan's book [44] is summarized as follows:

Lemma 2.1 (Strengthen Postcondition Law). *Let pre (precondition) and $post$, $post'$ (postconditions) be any FOL formula, if $post' \Rrightarrow post$, then $x : [pre, post] \sqsubseteq x : [pre, post']$.*

Lemma 2.2 (Weaken Precondition Law). *Let pre, pre' (preconditions) and $post$ (postcondition) be any FOL formula, if $pre \Rrightarrow pre'$, then $x : [pre, post] \sqsubseteq x : [pre', post]$.*

Lemma 2.3 (Skip Law). *If* $pre \Rrightarrow post$, *then* $x : [pre, post] \sqsubseteq \textbf{\textit{skip}}$.

Lemma 2.4 (Sequential Composition Law (Seq)). *Let mid be any formula except for pre or post.* $x : [pre, post] \sqsubseteq x : [pre, mid]; x : [mid, post]$.

We denote $post\langle x := E\rangle$ as a new condition that assigns all occurrences of x in $post$ by E.

Lemma 2.5 (Assignment Law). *Let E be any Expression,* $post\langle x := E\rangle$ *assigns every x in post with E. If* $pre \Rrightarrow post\langle x := E\rangle$, *then* $x : [pre, post] \sqsubseteq \boldsymbol{x = E}$.

Lemma 2.6 (Alternation Law). *Let GG be the disjunctive normal form of the guards* $G_0, G_1, , ..., G_i, ..., G_n$, *if* $pre \Rrightarrow GG$, *then* $x : [pre, post] \sqsubseteq \textbf{\textit{if}} \bigsqcup_i (G_i \textbf{\textit{ then }} x : [G_i \wedge pre,\ post])$ *where* ***if*** $\bigsqcup_i G_i$ ***then*** *means if* G_0 *then ... else if* G_i *then ...* .

Iterations (while loops) are built by loop conditions, invariants, and variants. An invariant *Inv* is a formula that, if it is true initially, stays true for each repetition. The variant V of the iteration is a value changing in the iteration and guarantees the termination of the iteration.

Lemma 2.7 (Iteration Law). *Let Inv, the invariant, be any formula; let V, the variant, be any integer-valued expression. Let GG be the disjunctive normal form of the guards* $G_0, G_1, ..., G_i, ..., G_n$ *then* $x : [Inv,\ Inv \wedge \neg GG] \sqsubseteq$ ***while*** $\bigsqcup_i (G_i$ ***do*** $x : [Inv \wedge G_i, Inv \wedge (0 \leq V < V_0)])$ *where* V_0 *is the initial value of V,* ***while*** $\bigsqcup_i G_i$ ***do*** *means while* G_0 *do ... else* G_i *do ... else* G_n *do.*

Lemma 2.8 (Expand Law). *Let x be the origin variant and y be another variant and* y_0 *be the initial value of y, then* $x : [pre, post] = x, y : [pre, post \wedge y = y_0]$

Lemma 2.9 (Assertion Law). *Let E be a boolean condition for the variable x, then* $x : [pre, post] = assert\ E; x : [pre \wedge E, post]$

Procedure. A procedure is declared by a name, some parameters, and a program.

Definition 2.10 (Procedure). $procedure\ \langle Name\rangle\ (\langle Variable\rangle\ : \langle Type\rangle) \triangleq \langle Prog\rangle$.

Lemma 2.11 (Procedure Value Specification). *Given a procedure* $procedure\ \langle Name\rangle\ (value\ f\ : \langle Type\rangle) \triangleq w, f : [pre, post]$ *where w and f are different variables. Let A be the expression of the* $\langle Type\rangle$, *then* $w : [pre\langle f := A\rangle, post\langle f_0 := A_0\rangle] \sqsubseteq procedure\langle Name\rangle(A)$.

Lemma 2.12 (Procedure Result Specification). *Given a procedure* $procedure\ \langle Name\rangle\ (result\ f : \langle Type\rangle) \triangleq w, f : [pre, post\langle a := f\rangle]$ *where w, a, and f are different variables. Then* $w, a : [pre, post] \sqsubseteq procedure\langle Name\rangle(a)$.

The literature [4, 45] has established the correctness of the laws in this section. In particular, define a Hoare-triple-like notation $\{pre\}prog\{post\}$ that means that starting from a precondition that satisfies *pre*, if the program *prog* terminates, then the postcondition satisfies *post*. We represent the result as follows sans proof:

Theorem 2.13 (Soundness of Core Refinement Laws). *If* $\vec{x} : [pre, post] \sqsubseteq prog$ *is derivable from the laws in Section 2.2, then* $\{pre\}prog\{post\}$ *holds.*

3 Motivating Example

This section illustrates our intuitions for Refine4LLM with the square root (sqrt) algorithm. We extend the sqrt algorithm from integers to real numbers compared to the other program refinement works [44, 55], presenting how we guide LLMs and verify the generated code. Then, we illustrate the learning strategy for extending the refinement law to evolve and extend the refinement calculus to reduce the complexity of program refinement.

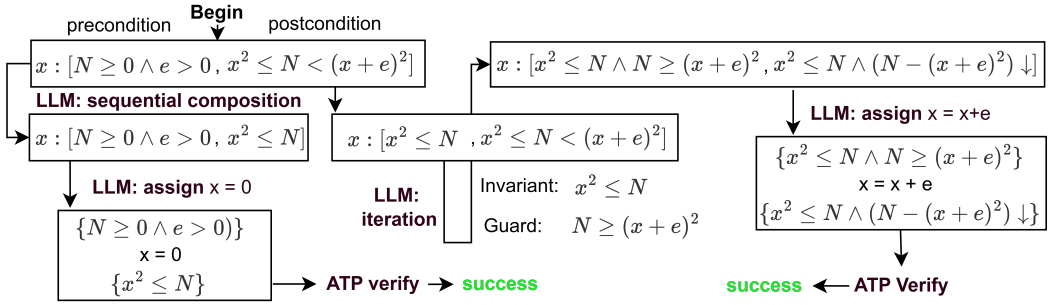

Fig. 2. Square Root with Program Refinement: success version. The specification format follows the notations in Section 2.1, and the proof obligation follows the Hoare Logic format. The LLM selects one law introduced in Section 2.2 and generates the associated code. The symbol ↓ means the variant is strictly decreasing.

3.1 Guide the LLM

The specification of the square root example reads as follows: for any positive constant N and e, the program C will change the variable x until x^2 is less or equal to N, but $(x+e)^2$ is larger than N. Program refinement aims to refine the specification stepwise into smaller pieces while building the program step by step. We begin with the basic refinement calculus consisting of the core laws defined in Section 2 and show the refinement process in Figure 2.

The first possible choice is the sequential composition law to separate the two constraints $x^2 \leq N$ and $N < (x+e)^2$. For the first part, the LLM can utilize the specification to find a possible correct assignment $x = 0$, which could be verified by Hoare logic and automatic theorem provers (ATPs). Refine4LLM can then apply the iteration law for the second part with the specification structure $[Invariant, Invariant \wedge \neg Guards]$ for iteration. The iteration law will split the postcondition into two parts: a guard condition and an invariant, further establishing an iterative structure that preserves the invariant while changing the variant until the guard condition is violated. The symbol ↓ means that the variant is strictly decreasing. Refine4LLM will use the constraints "$Invariant \wedge Guards \rightarrow Invariant \wedge Variant\ is\ strictly\ decreasing$" as a prompt and instruct the LLM to generate code like $x = x + e$. Finally, Refine4LLM will verify whether the generated assignment can preserve the invariant and decrease the variant. Note that it is *partially* correct because it only verifies that the variant is decreasing. To achieve *total* correctness, we need to verify that there exists one state in the iteration where the guard condition is violated.

3.2 Failure Feedback

The program refinement can be implemented in many ways. Figure 3 shows another direction: initializing the x with a large number and decreasing it until the constraints are satisfied. The associated program will begin by initializing the variable x to satisfy the invariant $N < (x+e)^2$. Traditional synthesis methods are hard to infer the correct assignment following the idea that $\forall N > 0, e > 0, (x > N + \frac{1}{4} \rightarrow (x+e)^2 > N)$ since $\forall N > 0, (N + \frac{1}{4})^2 > N$. In contrast, the LLM may *guess* the assignment could be $x = N$ or $x = 1$ but lacks verification and guidance to the correct answer. Refine4LLM can verify the output to ensure correctness or give counterexample feedback to guide the LLM to the correct answer. If the LLM generates the assignment $x = N$, Refine4LLM will reject the assignment and give the counterexample feedback to the LLM until some correct assignments like $x = N + 1$ are generated. Then, the Refine4LLM will apply the iteration law and instruct the LLM to generate the code that satisfies the constraints for the new specification. The LLM may generate code like $x = x - e$, and Refine4LLM will formally generate the associated proof

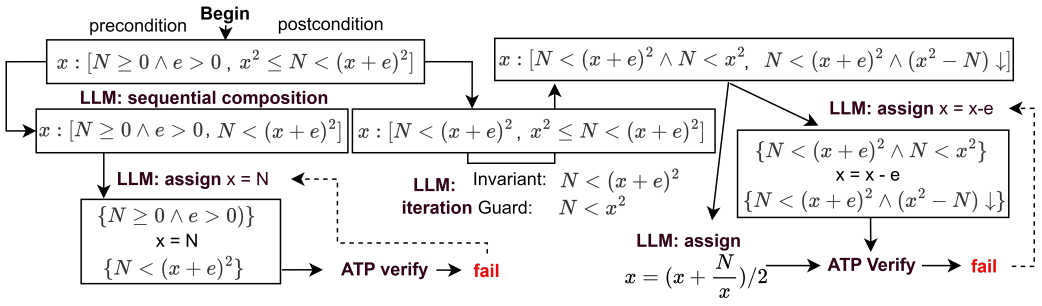

Fig. 3. Square Root with Program Refinement: failed version. The dash lines mean the failure feedback.

obligation:

$$(N < (x+e)^2 \wedge N < x^2) \rightarrow (N < (x'+e)^2 \wedge (N - x'^2 < N - x^2) \wedge x' = x - e) \tag{3}$$

However, ATPs will reject the generated code because the variant $N - x^2$ is not strictly decreasing in some cases. Interestingly, despite similarities in the two symmetric directions of program refinement for the square root algorithm, the previous program in Figure 2 succeeds, but the current program will fall into an infinite loop. This asymmetry highlights the importance of formal program verification for seemingly simple algorithms. After the first failure, the LLM will receive feedback on the failure and generate a new assignment like $x = (x + \frac{N}{x})/2$, which may be based on Newton's method. However, ATPs show that the variant will not decrease and reach a fixed point in some cases due to the float number precision error. If the LLM repeatedly fails to generate the verified code several times, Refine4LLM will trace back to the last refinement step and apply a new refinement law to find another refinement direction.

3.3 Optimizing the Algorithm with Binary Search

Identifying an appropriate intermediate condition for sequential composition law will facilitate further refinement. As shown in Figure 4, the specification introduces a new variable y so that x, y delimits the search space for the square root. We can interpret the refinement procedure first by initializing x and y to satisfy the intermediate condition $x^2 \leq N < y^2$ and then adjusting x, y to ultimately fulfill the postcondition $x^2 \leq N < y^2 \wedge y < x + e$. The setup naturally leads to a looping structure, as the invariant $x^2 \leq N < y^2$ and the guard condition $y \geq x + e$ can be derived directly from the precondition and postcondition. The LLM may apply the alternation law to ensure the loop's termination by diminishing the variant $y - (x + e)$ while maintaining the invariant. One way is to set the condition $(\frac{x+y}{2})^2 \leq N$ in the precondition with the alternation law. Then, based on the new specification, the LLM will assign x with $\frac{x+y}{2}$ to decrease the variant $y - (x + e)$ while preserving the invariant. The proof obligation is automatically generated by Refine4LLM as follows:

$$(x^2 \leq N < y^2 \wedge x+e < y \wedge (\frac{x+y}{2})^2 \leq N) \rightarrow (x'^2 \leq N < y^2 \wedge y-(x'+e) < y-(x+e) \wedge x' = \frac{x+y}{2}) \tag{4}$$

Similarly, the other side should assign y with $\frac{x+y}{2}$. The new refinement direction will generate a binary search-based algorithm that is more efficient than the above programs. The detailed refinement procedure and generated code are shown in Section 10. However, synthesizing an optimized algorithm is a non-trivial task. In this work, we mainly depend on the LLM's knowledge and code generation ability to generate an optimized algorithm to pass the test cases in the specific time and space complexity requirements.

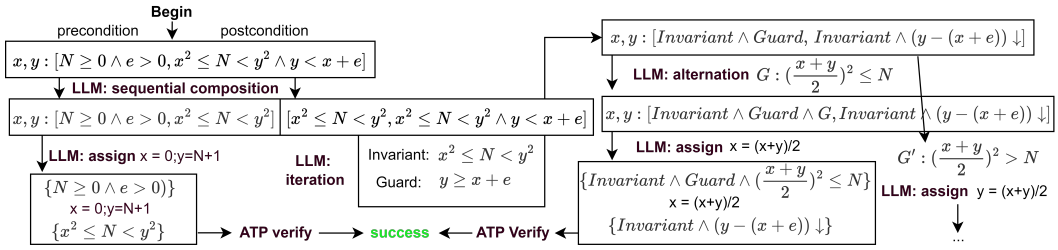

Fig. 4. Square Root with Program Refinement: binary search version.

3.4 Learning Strategies for Extending Refinement Calculus

The above program refinement procedures share some similarities, which start with the sequential composition law and then use the assignment law for the first part and the iteration law for the second part, as illustrated in Figure 5. Semantically, this kind of refinement procedure first initializes the variables to satisfy the invariant, then preserves the invariant, and changes the variant with the iteration until the postcondition is satisfied. On the one hand, the sequential composition law constructs the structure $[Invariant, Invariant \wedge Guard]$ for further iteration. On the other hand, the *future* iteration law can hint at the first step of sequential composition law to split the specification. If the LLM has the *future* information on the iteration law, it will be more possible to split the specification in the expected way.

To capture the common patterns of program refinement, we design a learning algorithm to extract the *patterns* of refinement procedures and extend the refinement calculus. The algorithm takes a collection of refining processes and extracts the law patterns and specification patterns from the refinement history dataset. The new extended laws are summarized from the common patterns in the dataset and allow the specifications to be refined with fewer refinement steps, reducing the depth of the search and verification. All the new laws are built upon the core refinement laws, and their correctness can also be derived based on correctness-by-construction [11]. Different from the typical program synthesis methods like [12, 29] that recover higher-order functions such as map, fold, and filter, Refine4LLM tries to conclude higher-level refinement laws such as initialized iteration rule and traverse rule, that capture the high-level synthesis idea of the program synthesis. In summary, extending refinement laws has the following advantages:

- They broaden the horizon of LLM from one-step refinement to considering future directions.
- They reduce the depth of refinement and save time and resources of interaction with LLMs.
- They may simplify program verification and reduce the requirement of ATPs.

4 Overview

Figure 6 shows an overview of our approach. Generally, it can be divided into two parts: the formal and informal systems. In the formal system, we first define the formal specification language L_{spec} and program language L_{pl} defined in Section 5. The formal specification will first be transformed into an abstract syntax tree. Then, with one specific law, Refine4LLM will formally transform the specification into the new specification and build the proviso constraints of the law. Finally, after extracting the generated code from LLM, Refine4LLM will automatically build the proof obligations and the verification scripts for ATPs to verify. ATPs will try to verify the scripts automatically and output the success or error message. If the code fails to verify, Refine4LLM will provide feedback to the LLM with possible counterexamples. In the informal system, the user needs to first formalize the natural language specification into a formal specification with the aid of the LLM. Secondly, Refine4LLM will build the prompt and describe the refinement laws to the LLM. Then, the LLM

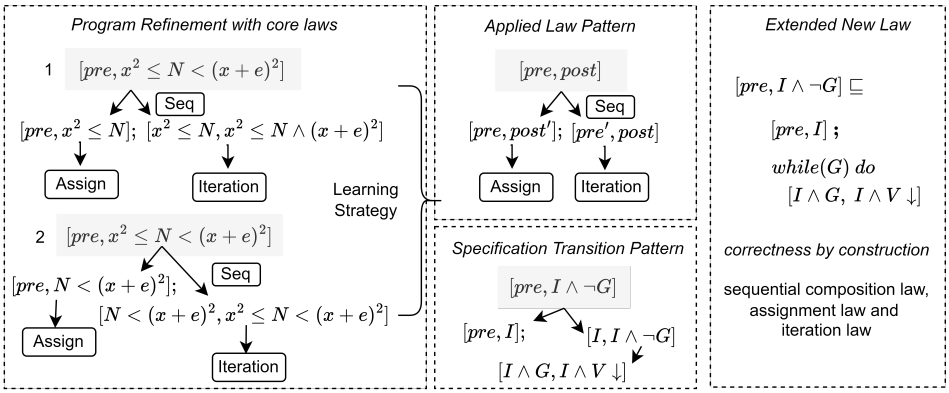

Fig. 5. One Example for the Refinement Law Learning and Extending. *pre* means precondition and *post* means postcondition. I, G, V means the invariant, guard condition, and variant of an iteration.

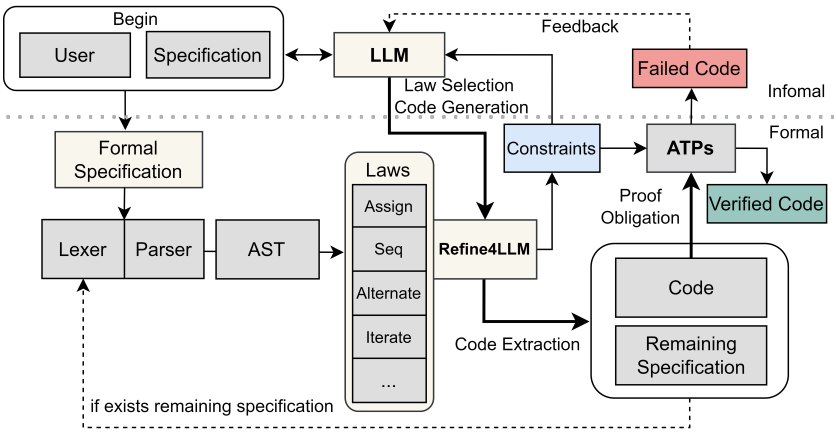

Fig. 6. Overview of Refine4LLM that combines LLMs and program refinement.

selects one refinement law based on the specification's description and constraints and generates the associate code. The LLM will regenerate the code based on the verification result. If getting multiple times of failure, Refine4LLM will trace back to the last refinement step and interact with the LLM to choose another refinement law and re-generate the associated code.

5 Refine4LLM's Languages

We introduce our formal specification language L_{spec} used to describe the specification and the programming language L_{pl} for our generated program. As these languages interact closely with LLMs, we target designing languages that are well understood and applied by LLMs.

5.1 The Specification Language

Our specification language L_{spec} extends first-order logic (FOL) and is a subset of the language of Coq [6]. Most LLMs are familiar with both FOL and Coq grammar. We follow the standard syntax and semantics of FOL and highlight the following notations.

Table 1. Syntax of the specification language L_{spec}.

$$
\begin{aligned}
\langle Type\rangle &::= \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{array}\ \langle Type\rangle \\
\langle Spec\rangle &::= \texttt{Precondition:}\langle Definition\rangle\ \texttt{Postcondition:}\langle Definition\rangle \\
\langle Definition\rangle &::= \langle Name\rangle\langle Params\rangle\ \texttt{:=}\ \langle Expr\rangle\,. \\
\langle Params\rangle &::= \texttt{(}\ \langle Name\rangle\ \texttt{:}\ \langle Type\rangle\ \texttt{)} \\
\langle Expr\rangle &::= \langle Logit\rangle \mid \langle Logit\rangle \wedge \langle Expr\rangle \mid \langle Logit\rangle \vee \langle Expr\rangle \mid \neg\langle Expr\rangle \mid \langle QExpr\rangle \\
\langle QExpr\rangle &::= \texttt{forall}|\texttt{exists}\ \langle Params\rangle\ \langle Expr\rangle \\
\langle Logit\rangle &::= \langle Term\rangle \mid \langle Term\rangle < \langle Logit\rangle \mid \langle Term\rangle <= \langle Logit\rangle \mid \langle Term\rangle = \langle Logit\rangle \\
&\quad \mid \langle Term\rangle > \langle Logit\rangle \mid \langle Term\rangle >= \langle Logit\rangle \mid \langle Term\rangle <> \langle Logit\rangle \\
\langle Term\rangle &::= \langle Factor\rangle \mid \langle Factor\rangle + \langle Term\rangle \mid \langle Factor\rangle - \langle Term\rangle \\
\langle Factor\rangle &::= \langle atom\rangle \mid \langle atom\rangle * \langle Factor\rangle \mid \langle atom\rangle\ /\ \langle Factor\rangle \\
\langle atom\rangle &::= \langle Number\rangle \mid \langle Variable\rangle \mid \langle Const\rangle \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid \texttt{-}\ \langle Expr\rangle \mid \texttt{(}\ \langle Expr\rangle\ \texttt{)} \mid \langle Variable\rangle_{\emptyset} \\
&\quad \mid \langle Name\rangle\texttt{[}\langle atom\rangle\texttt{]} \mid \langle Name\rangle\texttt{[}\langle atom\rangle : \langle atom\rangle\texttt{]}
\end{aligned}
$$

Variants and Constants. We use lower case words like x, y, z to denote the *variants* that will change in the refinement and upper case words like N, M to denote *constants*. Both variants and constants should be typed.

Relations and Functions. We use common *relation* operators and *function* operators in SMT, such as $<, =, +, -, *, /, Array[Int]$. The theory of integer-indexed arrays following [13] is defined below:

$$
\begin{aligned}
\sum_A^{\mathbb{N}} : \sum_A \cup \sum_{\mathbb{N}} &::= \{a[i], a\langle i \triangleleft v\rangle, =, 0, 1, +, \geq\} \\
\forall a, i, j, v.\ i = j &\rightarrow a[i] = a[j] \\
\forall a, i, j, v.\ i = j &\rightarrow a\langle i \triangleleft v\rangle[j] = v \\
\forall a, i, j, v.\ i \neq j &\rightarrow a\langle i \triangleleft v\rangle[j] = a[j]
\end{aligned}
$$

Syntax. We define our specification based on the first-order theory and theory of arrays. The full syntax of L_{spec} is given in Table 1, where $\langle Specification\rangle$ defines the specification that needs to be refined, $\langle Definition\rangle$ defines the condition that the variants should satisfy, $\langle Params\rangle$ defines the variants and constants. In the case of $\langle atom\rangle$, $\langle Variable\rangle_{\emptyset}$ denotes the previous value of the variable, $\langle Name\rangle\texttt{[}\langle atom\rangle\texttt{]}$ specifies the array selecting operation, and $\langle Name\rangle\texttt{[}\langle atom\rangle : \langle atom\rangle\texttt{]}$ is used for array slicing operation. The remainder of the syntax is standard FOL used in SMT solving.

Semantics. We follow the standard FOL semantics defined in Coq and only present the notable elements in Table 2. Note that the theory of arrays is realized by relations and functions, similar to its treatment in the literature [20].

Table 2. The semantics of the specification language L_{spec}.

$$
\begin{aligned}
type\ T &\iff A\ value\ set\ T\ \ [\![e_T]\!] \in T \\
variants\ v : T &\iff A\ value\ v \in T\ \ [\![v]\!] \\
constant\ c : T &\iff A\ value\ c \in T\ \ [\![c]\!] = c \\
functional\ operator\ f(T_1, T_2, ...) : T &\iff [\![f(a, b, ...)]\!] = f([\![a]\!], [\![b]\!], ...) \\
relational\ operator\ R(T_1, T_2, ...) : Bool &\iff [\![R(a, b, ...)]\!] = R([\![a]\!], [\![b]\!], ...)
\end{aligned}
$$

Table 3. Syntax of the program language L_{pl}.

$$
\begin{aligned}
\langle \mathit{Type} \rangle &::= \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{array}\langle \mathit{Type} \rangle \\
\langle \mathit{Expr} \rangle &::= \langle \mathit{Number} \rangle \mid \langle \mathit{Name} \rangle \mid \texttt{true} \mid \texttt{false} \mid \langle \mathit{Variable} \rangle \\
&\quad \mid \langle \mathit{Expr} \rangle \texttt{ == } \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ < } \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ <= } \langle \mathit{Expr} \rangle \\
&\quad \mid \langle \mathit{Expr} \rangle \texttt{ > } \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ >= } \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ != } \langle \mathit{Expr} \rangle \\
&\quad \mid \langle \mathit{Expr} \rangle \texttt{ and } \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ or } \langle \mathit{Expr} \rangle \mid \texttt{not}\langle \mathit{Expr} \rangle \\
&\quad \mid \langle \mathit{Expr} \rangle + \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle - \langle \mathit{Expr} \rangle \\
&\quad \mid \langle \mathit{Expr} \rangle \cdot \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle \texttt{ / } \langle \mathit{Expr} \rangle \\
&\quad \mid \langle \mathit{Variable} \rangle \texttt{[} \langle \mathit{Expr} \rangle \texttt{]} \\
&\quad \mid \langle \mathit{Variable} \rangle \texttt{[} \langle \mathit{Expr} \rangle \texttt{:} \langle \mathit{Expr} \rangle \texttt{]} \\
\langle \mathit{Prog} \rangle &::= \texttt{pass} \mid \langle \mathit{Variable} \rangle \texttt{ = } \langle \mathit{Expr} \rangle \mid \langle \mathit{Prog} \rangle \texttt{; } \langle \mathit{Prog} \rangle \\
&\quad \mid \texttt{assert}\langle \mathit{Expr} \rangle \\
&\quad \mid \texttt{while (}\langle \mathit{Expr} \rangle\texttt{): } \langle \mathit{Prog} \rangle \texttt{ endwhile} \\
&\quad \mid \texttt{if (}\langle \mathit{Expr} \rangle\texttt{): } \langle \mathit{Prog} \rangle \texttt{ else } \langle \mathit{Prog} \rangle \texttt{ endif} \\
&\quad \mid \texttt{def } \langle \mathit{Name} \rangle \texttt{ (}\langle \mathit{Name} \rangle \texttt{ : } \langle \mathit{Type} \rangle\texttt{)*:}\langle \mathit{Prog} \rangle \texttt{ enddef}
\end{aligned}
$$

Table 4. Syntax of the program language L_{mix}, following constructs defined in Tables 1 and 3.

$$
\begin{aligned}
\langle \mathit{Mix} \rangle &::= \langle \mathit{Spec} \rangle \mid \langle \mathit{Prog} \rangle \\
\langle \mathit{MixProg} \rangle &::= \texttt{pass} \mid \langle \mathit{Variable} \rangle \texttt{ = } \langle \mathit{Expr} \rangle \mid \langle \mathit{Mix} \rangle \texttt{; } \langle \mathit{Mix} \rangle \\
&\quad \mid \texttt{assert(}\langle \mathit{Expr} \rangle\texttt{)} \\
&\quad \mid \texttt{while (}\langle \mathit{Expr} \rangle\texttt{): } \langle \mathit{Mix} \rangle \texttt{ endwhile} \\
&\quad \mid \texttt{if (}\langle \mathit{Expr} \rangle\texttt{): } \langle \mathit{Mix} \rangle \texttt{ else } \langle \mathit{Mix} \rangle \texttt{ endif} \\
&\quad \mid \texttt{def } \langle \mathit{Name} \rangle \texttt{ (}\langle \mathit{Name} \rangle \texttt{ : } \langle \mathit{Type} \rangle\texttt{)*:}\langle \mathit{Mix} \rangle \texttt{ enddef}
\end{aligned}
$$

5.2 The Program Language

Our program language is mainly based on the While language and Guarded Command Language [25]. The formal semantics follows the semantics of the While language with regard to the standard interpretation of validity [51]. The language is kept simple to make it easier for the LLM to understand and generate. The complete syntax of our program language is given in Table 3. Our programming language is imperative and has data types for booleans, natural numbers, integers, float, characters, and arrays. We include the extension of Array and Assert statements. The array has a natural number index type and the reading, updating, and slicing operations. To control the size and structure of programming, we also incorporate the use of procedures. The procedure is declared by a name, some parameters, and an associated program. This core language can be implemented as one of the most common programming languages, such as C and Python.

Lastly, we define the *mixed programming language* L_{mix} for the program refinement procedure, which is a mixture of L_{spec} and L_{pl}. Formally, its main construct is a variant of $\langle \mathit{Prog} \rangle$ where part of the program may be a specification, as defined in Table 4. This "intermediate" language is used in the middle of the refinement process, where parts of the specifications are refined into programs, and the other parts are still specifications. We may loosely call such a program a "mixed program".

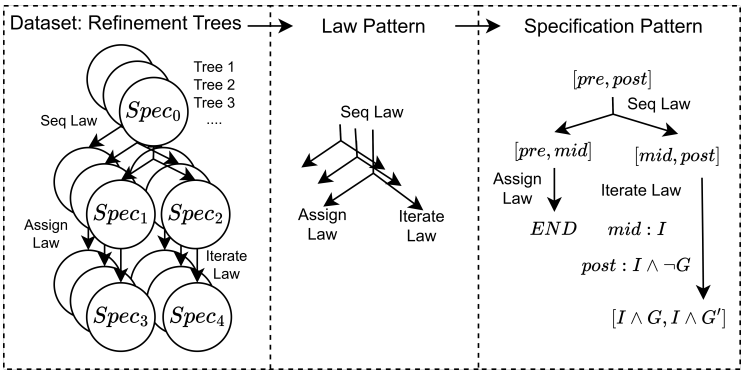

Fig. 7. The Framework of Law Learning Algorithm. The refinement trees come from the examples that have been refined to program. The token END means there are no more specifications.

6 Refine4LLM's Refinement Calculus

This section introduces the law learning algorithm to extend the refinement laws and Refine4LLM's extended refinement calculus.

6.1 Law Learning Algorithm

The learning algorithm first inputs a small dataset of problems and the core laws defined in Section 2. The core laws should be low-level but expressive to refine the specifications in the dataset. The algorithm grows the library of the refinement calculus by reviewing the examples from the dataset, finding common law patterns from the refinement trees, and abstracting common specification patterns into new laws.

Law Pattern. We first apply the Refine4LLM to refine all the problems in the dataset with the core laws. As shown in Figure 7, the refinement process is organized in a tree structure called refinement trees, where the node represents a specification, and the edge denotes a refinement law connecting these specifications. Given this constraint, we systematically traverse the tree via the edges, converting this traversal into a sequential representation of the refinement process. The algorithm identifies common sub-sequences within these sequences derived from the refinement tree. By analyzing these sub-sequences, we can extract patterns and repetitive refinement laws.

Specification Pattern. After extracting the law pattern, we follow its frequency to extract the specification pattern. The specification language is defined in Section 5. We build the E-graph [24] to manage the specification patterns and find equivalences among different expressions efficiently from the high-level abstraction to the low-level element. To reduce the complexity, we first define some rewriting rules with high-level abstraction based on the semantics of the base laws:

$$
\begin{aligned}
&Strengthen - postconditon : [pre, post] \rightarrow [pre, post'] \\
&Weaken - preconditon : [pre, post] \rightarrow [pre', post] \\
&Skip : [pre, post] \rightarrow END \\
&Assign : [pre, post] \rightarrow END \\
&Seq : [pre, post] \rightarrow [pre, mid]; [mid, post] \\
&Alternation : [pre, mid] \rightarrow [pre \wedge G1, post]; [pre \wedge G2, post]; ... \\
&Iteration : [I, I \wedge \neg G] \rightarrow [I \wedge G, I \wedge G']
\end{aligned}
\tag{5}
$$

where *END* means that there are no more specifications, and the refinement process ends.

The E-graph data structure allows us to represent and reason about multiple equivalent expressions simultaneously, facilitating the application of rewriting rules in a structured and systematic way. We create an initial E-graph with nodes representing each unique specification. Then, we identify nodes that can be merged based on the rewriting rules and merge them. Finally, systematically apply the rewriting rules to the E-graph, expanding and merging nodes as needed. After applying all the rules, we extract the most common sub-components from the E-graph. Then, we expand the specification tree to the deeper level. The process of constructing and managing an E-graph continues iterative from high level to low level.

For example, in Figure 7, Refine4LLM will show that some examples in the dataset can be refined by sequential composition law, assignment law, and iteration law. Intuitively, the refinement process first initializes the variable to satisfy the invariant and then builds the iteration structure with the invariant. The refinement trees contain two sub-trees: the first part will be refined using assignment law, while the second part will be refined using iteration law. Based on the iteration law, the mid-condition is the invariant, and the postcondition can be regarded as a conjunction of an invariant and a guard condition. We conclude the pattern and derive the initialized iteration law in Theorem 6.6. Generally, the base laws could complete the equivalent refinements, but those produced by the final learned laws are much shorter and reduce the verification effort. Besides, we only derive frequently used laws for efficiency, not for completeness. The detailed pseudo-code of the learning algorithm is presented in [3].

6.2 Extended Laws

Skip. The new skip law gives the variant an initial value, utilizing the fact that the initial and final variables have the same value.

Lemma 6.1 (Initialised Skip Law). *Let x_0 denote the initial value of variant x, if $(x = x_0) \wedge P \Rrightarrow Q$, then the specification $x : [P, Q] \sqsubseteq$ Skip.*

Proof. Directly from the skip law in Lemma 2.3 as $P \Rrightarrow Q$. □

Seq. We extend a new sequential composition law to flexibly divide one specification into two parts based on the Strengthen-Postcondition and Weaken-Precondition laws.

Lemma 6.2 (Flexible Sequential Composition Law). *Let P, Q, A, B, C, D be some formulate, if $(P \Rrightarrow A) \wedge (B \Rrightarrow C) \wedge (Q \Rrightarrow D)$, then the specification $x : [P, Q] \sqsubseteq x : [A, B]; x : [C, D]$.*

Proof. First, use the sequential composition law in Lemma 2.4, $x : [P, Q] \sqsubseteq x : [P, B]; x : [B, Q]$. Then refine the two parts with the weaken-precondition law in Lemma 2.2, $x : [P, B] \sqsubseteq x : [A, B]; x : [B, Q] \sqsubseteq x : [C, Q]$. Finally, refine the second part with the strengthen-postcondition law in Lemma 2.1, $x : [C, Q] \sqsubseteq x : [C, D]$. □

Assign. We extend two assignment laws. The initialized assignment law utilizes the initial values of the variants to simplify the further proof for $pre \Rrightarrow post\langle x := E\rangle$. The following assignment law allows any assignment in its second half, provided the changed variants.

Lemma 6.3 (Initialized Assignment Law). *Let E be any Expr in the programming language, $post\langle x := E\rangle$ replaces every x in the formula post with E. If $(x = x_0) \wedge (y = y_0) \wedge pre \Rrightarrow post\langle x := E\rangle$, then $x, y : [pre, post] \sqsubseteq \boldsymbol{x = E}$.*

Proof. Use the assignment law in Lemma 2.5 as $pre \Rrightarrow post\langle x := E\rangle$. □

Lemma 6.4 (Following Assignment Law). *Let E be any Expr in the programming language, $post\langle x := E\rangle$ replaces every x in the formula post with E.* $x : [pre, post] \sqsubseteq x : [pre, post\langle x := E\rangle]$ ***; x = E.***

Proof. First use the sequential composition law, $x : [pre, post] \sqsubseteq x : [pre, post\langle x := E\rangle]$**;** $x : [post\langle x := E\rangle, post]$. Then, refine the second part using the assignment law, $x : [post\langle x := E\rangle, post] \sqsubseteq \mathbf{x} = \mathbf{E}$. □

Alternate. The if-else alternation law is a simplified version of the original one without further proof of the guard condition.

Lemma 6.5 (If-else Alternation Law). *Let P, Q, and G be some formulae, then the specification* $x : [P,\ Q] \sqsubseteq if\ (G)\ (x : [P \wedge G,\ Q])\ else\ (x : [P \wedge \neg G,\ Q])$.

Proof. As $Pre \Rrightarrow G \vee \neg G$ based on the law of excluded middle, the lemma can be directly implied from the alternation law in Lemma 2.6. □

Iterate. The new initialized iteration law first assigns the initial value to the invariant, and the second specification preserves the invariant and changes the variant during iteration until the negated guard condition is reached. In practice, based on the convergence of monotonic sequences of real numbers, we replace the existing condition with the monotonic and bounded condition given in Lemma 6.7. To avoid infinite loops, we add the assertion to check whether the expression V decreases by at least the error bound of the floating-point precision.

Lemma 6.6 (Initialised Iteration Law). *Let P, I, and G be some formulae, V be any variant expression, and i and M are positive integers, then the specification* $x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]$ ***;*** $while(G)\ do\ (x : [I \wedge G,\ I \wedge (\exists i < M,\ V_i \rightarrow \neg G))]$.

Proof. First, using the sequential composition law in Lemma 2.4, $x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]$**;** $x : [I, I \wedge \neg G]$. Then, refine the second part with the iteration law in Lemma 2.7. Note that we replace the condition for integer-valued variants with any variant expression for scalability. To guarantee the termination of the iteration, a state of variant should exist to negate the guard condition after finite iterations. □

Lemma 6.7 (Assertion Iteration Law). *Let P, I, and G be some formulae, V be any variant expression, then the specification* $x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]$ ***;*** $while(G)\ do\ (x : [I \wedge G,\ I \wedge V < V_0]; assert\ V \neq V_0)$.

Proof. First, follow the initialized Iteration Law. Then, note that the float precision error is e, then we have $\exists i = \lceil \frac{V_0}{e} \rceil < M, V < 0 \rightarrow \neg G$. □

Traverse. We build a traverse law to facilitate the problems related to the array. The formula P contains the variants l and i, which can be equations that recursively define a sequence. Note that the following refinement should preserve the invariant $P(l, i)$ and make progress to $P(l, i + 1)$ following induction.

Lemma 6.8 (Traverse Law). *Let l be the list of type T, natural numbers m and n denote the range, pre and P be some formula,* $l : [pre, \forall i : nat \wedge m \leq i < n \rightarrow P(l, i)] \sqsubseteq l, i : [pre, l[m]];\ i = m$ ***;*** $while(i < n)\ do\ (l, i : [P(l, i), P(l, i + 1)]$**;** $i = i + 1)$.

Proof. First, using the expand law and sequential composition law in Lemma 2.4, $l, i : [pre, l[i] \wedge i = m]; l, i : [l[i] \wedge i = m, l[i] \wedge i = n]$. Then refine the second part with the initialised assignment law Lemma 6.3 and iteration law in Lemma 2.7, we have $i = m$ **;** $while(i < n)\ do\ (l, i : [P(l, i), P(l, i) \wedge 0 \leq n - i < n - i_0]$. Finally, using the following assignment law in Lemma 6.4 for the specification, $[P(l, i), P(l, i + 1) \wedge 0 \leq n - (i + 1) < n - i]$**;** $i = i + 1$ and can be simplified to the target. □

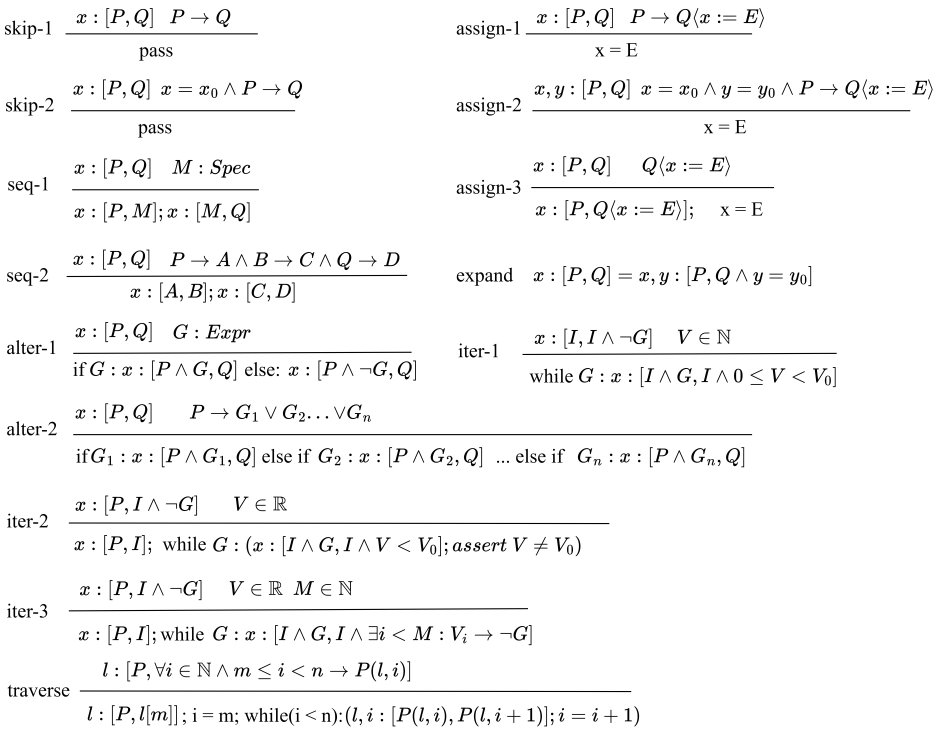

Fig. 8. Refinement laws w.r.t. the specification language and program language. The detailed notations are defined in Section 6.

The law learning algorithm can derive new laws not covered in this section. We transform the extended refinement calculus as background knowledge in the prompts to guide the LLM. The detailed instructions in the refinement laws can facilitate both LLM interaction and ATP verification. All the new laws are built upon the core refinement laws, and their correctness can also be derived. We summarize the above refinement laws in Figure 8.

Theorem 6.9 (Soundness of Derived Refinement Laws). *If $\vec{x} : [pre, post] \sqsubseteq prog$ is derivable from the laws in Section 2 and Section 6.1, then $\{pre\}prog\{post\}$ holds.*

Proof. Immediate from the proof of individual laws. □

7 Formal System: Refine and Verify

This section details the formal methods part of Refine4LLM, including the program refinement system and verification system.

7.1 Program Refinement System

A formal program refinement system leverages LLMs and automatic theorem provers (ATPs). The system automatically derives the proof obligations required to show that the refined program satisfies the original specification. The process iterates: new proof obligations are generated, refined, and submitted to the ATP until all obligations are proven. Once all proof obligations are satisfied, the system outputs the verified, refined program along with its formal specification. We list the

Law	LLM	Refine4LLM
Skip	-	verify $P \Rrightarrow Q$
Sequence	M	new spec $[P, M]$; $[M, Q]$
Assignment	$x = Expr$	verify $P \Rrightarrow Q\langle x := Expr\rangle$
Alternation	G	new spec $if\ (G)\ (x : [P \wedge G, Q])$ else $(x : [P \wedge \neg G, Q])$
Iteration	I, G	new spec $x : [P, I]$; while(G) do($x : [I \wedge G, I \wedge (\exists i < M,\ V_i \rightarrow \neg G)]$)
Traverse	P	new spec $l : [pre, l[m]]$; i = m; while(i < n) do $(l, i : [P(l, i), P(l, i+1)];\ i = i+1)$

Table 5. The schemes of specifications and conditions that the LLM and Refine4LLM generate for further verification in ATP. For the former four laws, the origin specification is $x : [P, Q]$. For the iteration law, the origin specification is $x : [P, I]; x : [I, I \wedge \neg G]$.

schema of the relationships among the refinement laws, the LLM-generated code, and Refine4LLM's output in Table 5.

7.2 Verification System with ATPs

Passively Verify. After the LLM generates the choice of the law and the associated code, Refine4LLM will verify them using ATPs to justify whether the code satisfies the constraints based on the condition of the selected refinement law. If the ATP confirms that the constraints are met, Refine4LLM will apply the refinement law to the current specification, leading to the generation of a new formal specification and the verified code. If the verification fails, the LLM will receive the failure message and the possible counterexamples. It then attempts to generate an alternative set of code, and the retry process has a defined limit. If repeated attempts fail, Refine4LLM will revert to the previous successful refinement step and the specification that was valid before the failed attempt. The LLM receives detailed feedback about the failures and the counterexamples, along with the last valid specification. For proof obligations that the ATP cannot automatically resolve, the LLM can provide human-readable explanations, suggest lemmas, or even generate proof sketches to assist the user in manual intervention.

Modular Verification. Refine4LLM implements modular verification by dividing the program refinement into small steps and splitting the specification into smaller, independent modules. Each module, with its own constraints, is verified separately by the ATPs, allowing for more efficient identification of errors localized to specific parts of the system. This method also facilitates parallel verification processes, speeding up the overall verification time. If a module fails verification, the LLM receives feedback specific to that module, including failure reasons and counterexamples. If all modules pass their respective verifications, Refine4LLM then integrates them to form a complete and verified refinement step.

8 Informal System: Interaction with LLM

This section describes the part of our system that uses informal LLM instead of formal methods and how the LLM and formal methods interact.

8.1 Specification Formalization

The formalization process involves breaking down complex specifications into smaller sub-specifications and refining them bottom-up to build the high-level specification. We build a top-down algorithm to split the specification and a bottom-up algorithm to synthesize the program into functional

abstractions. The user needs to check the correctness of the formalized specification. We assume that all the formal specifications from the user are correct and meet the requirements.

Top-down Specification Split. We assume that all the formal specifications checked by the user are correct and meet the user's requirements. Refine4LLM will interact with the users to formalize the user's requirements. The algorithm starts with a comprehensive, high-level specification of the system or software to be developed. Some functions in the specification may have been refined and stored in the library in the past, and some may not. Each sub-specification has its own specification, and they will be split into smaller specifications until all elements defined in their specifications have been refined or atomic elements in the language defined in Section 5. We retrieve similar specifications from the library to match the refined and stored specifications. We first ask the LLM to retrieve the possible equivalent specification in the library and then verify it using the logic laws based on the strengthen-postcondition Theorem 2.1 and Theorem 2.2 law. Intuitively, weaker preconditions necessitate handling a broader range of potential inputs during implementation, thereby reducing the flexibility of the approach. Additionally, stronger postconditions further restrict the implementation's freedom, demanding a closer adherence to specified outputs. The retrieved similar specification will replace the origin sub-specification. Finally, Refine4LLM will share all the specifications, including the sub-specifications, with the user to ensure correctness.

Bottom-up Refinement. For those sub-specifications that are not refined, we formalize the sub-specifications and refine them from the bottom. Each sub-specification is refined and validated independently, ensuring that every module functions correctly before it is integrated into the larger system. The bottom-up refinement allows for deep focus on each component's details, ensuring high-quality development with rigorous refinement and validation at each stage. Refine4LLM will save the refinement steps and the associated programs in the refinement library. The refinement procedures can be reused when meeting the new specifications that contain the same precondition and postcondition. The top-down and bottom-up methods not only save development time by avoiding redundant specifications from scratch but also ensure consistency and reliability by using previously validated, refined specifications.

8.2 Code Generation with LLMs

Retrieval Augmented LLM. We provide the LLM with the refinement calculus as background knowledge so that the LLM can utilize the refinement laws with the retrieval augmented techniques. We also customize our LLM for program refinement tasks by crafting instructions based on the formal specification language L_{spec} and our program language L_{pl} defined above. Our LLM has been instruct-tuned with the examples in Morgan's book [44]. This tailoring ensures that the LLM is well-equipped to handle the complexities of program refinement, drawing from a robust foundation of relevant examples and formal definitions.

Actively Guide. A *prompt* for the LLM, like GPT-4, refers to the instruction given to the model to elicit a response. The traditional prompts design always follows static templates like *Program Refinement for the following specification*. Instead of relying on static, template-based prompts, this approach treats the LLM as a constraint solver that actively constructs prompts that include logical formulae representing the specifications. These formulae detail the constraints the LLM's outputs must satisfy, ensuring that the generated code or refinement steps are aligned with the specific requirements of the task. The LLM will select the refinement law and generate the associated code based on the given prompts. As each step has its generated specification, there is no need to add the previous refinement as history based on the congruence of *Hoare Logic*. This method enhances the

LLM's utility by focusing its computational efforts on solving well-defined problems, leveraging its capabilities to generate precise and accurate refinements.

Prompt Engineering. Prompt refers to the art of crafting queries or inputs that are optimized to elicit the best or most relevant responses from an AI model. This practice is particularly prevalent in the field of machine learning and AI-driven interactive systems. Our prompt is relatively straightforward yet comprehensive and includes all the details of the refinement rules and specifications:

Given the refinement rule ... The previous code ... is not correct since ...
Provide a correct code satisfying the specification [pre, post].

Or to decide which refinement laws to use:

Choose a proper rule to refine the current specification [pre, post].

where all the rules are included in the context. This approach minimizes ambiguity and maximizes the relevance and accuracy of the LLM's responses, thereby enhancing the overall efficiency of the program refinement process.

9 Evaluation

This section shows the quantitative analysis of the most popular benchmarks compared to the state-of-the-art LLMs and a program refinement baseline.

9.1 Experiment Setting

We evaluate Refine4LLM to answer the following research questions:

- RQ1: Whether Refine4LLM can generate more robust programs than the baselines?
- RQ2: Can the learning algorithm and the associated extended refinement calculus reduce the time and depth of the refinement process?
- RQ3: Whether the top-down splitting and bottom-up refining method for building a library of program refinement can enhance Refine4LLM for complex problems?
- RQ4: How does the performance of Refine4LLM vary with different components?

9.1.1 baselines.

GPT-4. Generative Pre-trained Transformer 4 (GPT-4) [47] is a multi-modal large language model created by OpenAI and the fourth in its GPT foundation models. As a transformer-based model, GPT-4 uses a paradigm where pre-training using both public data and "data licensed from third-party providers" is used to predict the next token. After this step, the model was fine-tuned with reinforcement learning feedback from humans.

CorC. CorC [39, 54] is a graphical and textual IDE to create programs in a simple while-language following the Correctness-by-Construction approach. Starting with a specification, the open-source tool supports CbC developers in refining a program through a sequence of refinement steps and verifying the correctness of these steps using the theorem prover KeY [34].

9.1.2 Benchmarks. We choose the example programs in the baseline [54] and 164 benchmarks in the HumanEval dataset, widely used in evaluating code generation [18, 47, 62]. Besides, to test the correctness and robustness of the generated code, we adopt the *EvalPlus* [41] dataset with the same 164 benchmarks but has an average of more than 80 times the number of origin test cases to test the robustness of the generated code. We adapted the formal specifications from the Bigcode Project [8], which includes 18 programming languages for the origin 164 problems in the HumanEval dataset. The dataset contains the *golden codes* to solve the problem, which does not contain complex program language features beyond our language L_{pl}. These golden codes

Model	LLama3	GPT-3.5	Claude-3	GPT-4		Refine4LLM
Input Specification	NL	NL	NL	NL	NL+ FS	FS
HumanEval Passed	125	126	136	145	148	**150**
EvalPlus Passed	116	116	126	128	142	**150**

Table 6. A comparison of Refine4LLM and popular LLMs on the number of generated programs that passed the test cases in HumanEval and EvalPlus 164 benchmarks. NL means natural language inputs, and FS means formal specification inputs.

can be used to validate the correctness of the formal specifications. We manually check all the specifications in the dataset, which is necessary for establishing correctness criteria in any case and should not be a hurdle for program verification.

9.1.3 Implementation. We implement our approach based on Coq [6], CoqHammer [21], and GPT-4. The automatic verification is based on the open-source automated reasoning tool called CoqHammer. We follow the CoqHammer to incorporate four automated theorem provers, including cvc4 [7], vampire [40], Eprover [56] and Z3 [22] to facilitate the automation of verification. We input informal and formal specifications to the state-of-the-art LLM - GPT-4 to generate the code snippet and test in the test case. We choose GPT-4 as the base model of Refine4LLM and use instruction tuning [67] to customize the LLM with classic refinement examples from [44] and learn the extended version of the refinement calculus. Since our L_{pl} language only contains the most common language constructs in popular imperative languages, there's a straightforward translation from L_{pl} to, e.g., python. We then evaluate the converted Python program against the test cases following the Evalplus framework. The artifact is available online [3].

9.2 Experiment Results

In this section, we present the outcomes of our experiments, detailing the performance of Refine4LLM. The analysis aims to provide insights into the effectiveness and limitations of the approach under investigation.

9.2.1 RQ1: Performance of Program Generation. Table 6 shows the evaluation results on the Humaneval benchmarks. We choose the state-of-the-art LLMs include LLama3 [53], GPT-3.5, claude-3 [30] and GPT-4 as our baselines. The baselines' results follow the work [41]. The numbers in Table 6 indicate how many generated programs passed all the test cases in the datasets. EvalPlus has the same number of programs as HumanEval but more test cases for each program. Therefore, if a program has bugs, it may pass HumanEval but fail EvalPlus. Since LLM-generated code may contain bugs, EvalPlus generally reduces the pass rate on pure LLMs [41]. However, our method ensures correctness by design; thus, our pass rate does not drop, which is the key point of the comparison and represents a qualitative leap in improvement.

We add experiments to the GPT-4 that incorporate the formal specifications with the natural language descriptions because GPT-4 is the state-of-the-art model among all other LLMs. With only the natural language descriptions as input, GPT-4 shows the best overall performance of all the LLMs. We further analyze the error situation of the GPT-4 and Refine4LLM. GPT-4 may generate a program that passes all EvalPlus test cases, but our method can't because some code can't be verified. However, if we add more test cases, the LLM-generated programs may fail. By contrast, programs generated by our method can pass as many test cases as one can devise. In detail, GPT-4 sometimes ignores some exceptional cases like negative numbers or zero, which also interprets the experiment result that incorporating formal specifications enhances the GPT-4 since formal specifications contain helpful constraints.

Metrics	# Refinement Steps			Proof Time(s)		
Approach	CorC	Initial	Refine4LLM	CorC	Initial	Refine4LLM
Linear Search	5	5	4	0.4	0.2	0.1
Max Element	9	10	5	1.2	1.0	0.2
Pattern Matching	14	16	8	54.9	35.8	24.5
Exponentiation	7	7	5	15.2	14.4	10.4
Log Approximation	5	5	4	42.7	22.9	20.1
Dutch Flag Sort	8	9	5	5.7	4.3	4.1
Factorial	5	5	3	3.6	1.5	0.4

Table 7. A comparison of Refine4LLM and baseline *CorC* on classic program refinement problems. *initial* means the Refine4LLM system before the extension of the law. *# Refinement Steps* represents the number of steps required to refine the specification. *Proof Time(s)* measures the time taken to prove the correctness.

9.2.2 Failure Analysis. In the case of EvalPlus benchmarks that Refine4LLM failed to solve, we categorize the failures into three specific reasons with detailed explanations and promising direction for future improvements.

Complex Specifications. Eight problems have too complex specifications that Refine4LLM can not split and refine the specifications effectively. This complexity can arise from intricate mathematical conditions, nested logical expressions, or requirements that involve sophisticated relationships. For example, three problems in the Humaneval dataset involve parentheses matching, which may contain the subspecifications of a balanced pair of parentheses and non-overlapping parentheses. Some specifications are hard to refine without domain-specific knowledge requirements. One possible solution is to allow the user to define the domain-specific helper functions and include axioms or lemmas about their properties to guide the LLM and Refine4LLM.

Fail to Verify. Three problems fail to generate verified code due to the failures in the program verification. The automatic theorem prover may be unable to verify the generated code and the proof obligations due to limitations in its reasoning power or incomplete knowledge about the domain. We may use LLMs to suggest potential lemmas, auxiliary invariants, or transformations that could simplify the proof obligations. Besides, Refine4LLM also allows the user to check the proof failures and include axioms or lemmas to guide the theorem prover.

Lack of Laws for Some Data Structures. Three problems contain special data structures that Refine4LLM can not generate and verify the associated code. The system lacks built-in support for some special data structures, such as dictionaries or graphs, that are used in some problems. Our future work will extend the refinement calculus to support more complex data structures.

9.2.3 RQ2: Performance of Program Refinement and Verification. Table 7 shows the evaluation results on the example programs in the CorC baseline [54]. Since the baseline is not automated, we assume the user has completed all the refinement steps. Across all algorithms, Refine4LLM generally has fewer refinement steps and lower proof times than the CorC and Initial variants, which only contain the core refinement laws. The variability in refinement steps and proof times across different algorithms suggests differing complexity levels and optimization challenges. Due to their inherently complex nature, algorithms like pattern matching and log approximation may benefit more from advanced laws and optimizations. The experiment shows a significant difference in proof time for complex algorithms with Refine4LLM taking much less time than CorC and Initial.

9.2.4 RQ3: Performance of Refinement Library. Table 8 compares two scenarios: one our Refine4LLM and the other without any program-refined library based on the top-down splitting and bottom-up refining algorithm. We extract 50 examples from the HumanEval benchmarks

Model	No Library	Refine4LLM
#Refinement step	21.4	5.6
Refinement time(s)	~514	~275
Proof time(s)	~215	~87
Fall-Back rate(%)	26.87	9.45
# Program size	41.3	14.1
Pass rate(%)	52	82

Table 8. A comparison of Refine4LLM and a variant without the library of program refinement on EvalPlus benchmarks. *Fall-back rate* means the LLM fails to generate the code and falls back to the last specification. *# Program size* represents the number of lines generated.

that contain more than two sub-questions. We design the comparison with a maximum allowable duration of 600 seconds for the program refinement process. The top-down splitting of specifications has dramatically reduced the number of refinement steps required—from 21.4 to 5.6—and has decreased the refinement time from approximately 514 seconds to about 275 seconds. Additionally, the program size decreases with Refine4LLM, illustrating that bottom-up refining yields more compact programs in the libraries. Moreover, the library's modular verification not only reduces proof time but also decreases the fall-back rate of the LLM. This indicates that Refine4LLM effectively guides the LLM in generating accurate code that meets the constraints and refines the specification. Overall, Refine4LLM significantly enhances program refinement by incorporating the library, which simplifies the process and enhances both reliability and success rates.

9.2.5 RQ4: Ablation Study. We have conducted an ablation study to evaluate the impact of the extended laws and instruction fine-tuning on the performance of Refine4LLM by removing these components and measuring the effect on key metrics in the EvalPlus benchmarks. For the version of core law, we use the original version of GPT-4 without any extended refinement laws or instruction fine-tuning. In each interaction with GPT-4, we only give the core refinement laws to get the answer of the law selection and the code. For the extended law version, we further give all the refinement laws in each interaction. For the instruction tuning version, we apply instruction tuning to the GPT-4, which will be tuned on the dataset of refinement problems and relevant instructions about the core refinement laws.

The ablation study results shown in Table 9 indicate that both instruction fine-tuning and extended laws impact the performance and efficiency of Refine4LLM in distinct ways: Without instruction fine-tuning, the system's performance drops, and refinement time increases because the LLM is more likely to select inappropriate laws or require multiple iterations to identify the correct refinement approach. As a result, Refine4LLM struggles to generate correct code or to effectively refine some complex specifications, leading to more manual intervention and potential verification failure to complete the refinement. On the other hand, the refinement process takes longer in the absence of extended laws due to the need for additional refinement steps and iterations. Refine4LLM requires more interaction with the LLM and failure feedback loops before arriving at a valid solution.

Setting	Pass rate(%)	Refinement time(s)
Core Law	87.8	305
Extended Law	87.8	261
Instruct Tuning	91.5	234
Refine4LLM	91.5	215

Table 9. The Ablation Study Results.

```
// pre:  (N:float)(e: float) := N >= 0
     /\ e > 0
// post: (x:float)(y: float) := x*x <=
     N < y*y /\ y <= x+e
N, e = input()
# LLM selects Sequential Composition
    Law
// pre_1:= N >= 0 /\ e > 0
// post_1:= x*x <= N < y*y
# LLM selects Assignment law
x = 0
y = N+1
# verify pre_1 -> post_1(x := 0; y :=
    N+1)
// pre_2:= x*x <= N < y*y
// post_2:= x*x <= N < y*y /\ y <= x+e
# LLM selects Iteration law: Inv(pre_2
    ) Guard(~(y <= x+e))
while y > x+e:
  // pre_2_1:= pre_2 /\ y > x+e
  // post_2_1:= pre_2 /\ (...)
  # LLM selects Alternation law G((x+y)/2*(
      x+y)/2 > N)
  if (x+y)/2*(x+y)/2 > N:
    // pre_2_1_1:= pre_2_1 /\ (x+y)/2*(x+y)
       /2 > N
    // post_2_1_1:= post_2_1 /\ (...)
    # LLM selects Assignment law
    y = (x+y)/2
    # verify pre_2_1_1 -> post_2_1_1(y := (
      x+y)/2)
  else:
    // pre_2_1_2:=  pre_2_1 /\ (x+y)/2*(x+y
      )/2 <= N
    // post_2_1_2:= post_2_1 /\ (...)
    # LLM selects Assignment law
    x = (x+y)/2
    # verify pre_2_1_2 -> post_2_1_2(x := (
      x+y)/2)
```

Fig. 9. The generated code of the square root algorithm with program refinement. The statements tagged with a double slash // are the target specifications, tagged with *# LLM* are the LLM decisions, and tagged with *# verify* are the proof obligations. The type of the variables can be extracted from the specification.

10 Case Studies

This section illustrates four problems and their associated refinement procedure. First, we give the running code snippet of the motivation example - the square root algorithm to show how the Refine4LLM generates the verified code with proof obligations. Second, we illustrate the sorting problem's refinement procedure with the bubble sort algorithm and quick sort algorithm to show how the extended laws benefit program refinement. Thirdly, we present the program refinement of the prime factorization problem to show how Refine4LLM splits the complex specifications and involves functional abstraction and modular verification. We finally show a real-world example called mask distribution to illustrate how Refine4LLM formalizes a real-world problem and generates the program with the recurrence relation and the dynamic programming algorithm.

10.1 Square Root Algorithm

We show how Refine4LLM generates the program to solve the motivating example in Figure 9. The statements tagged with a double slash // are the target specifications, tagged with *# LLM* are the LLM decision, and tagged with *# verify* are the proof obligations. The proof obligations are the proviso condition to apply the associated refinement law. Note that we removed the iteration termination condition in (...) for a concise presentation. In detail, the LLM first sequentially splits the original specification into two parts. Informally, the first specification defines x, y such that $x^2 \leq N < y^2$, which can be implemented by assignment. Note that the assignment of y needs to satisfy the constraints in the postcondition of the specification, that is, $N < y^2$, eliminating the possibility of bugs of LLMs like $y = N$. The second specification preserves the invariant $x^2 \leq N < y^2$ and makes the variants x, y closer until $x + e \geq y$, which can be implemented with the iteration. The invariant, the guard condition, and the variant can be extracted from the specification. The LLM uses the alternation law to add another constraint and reduces the distance

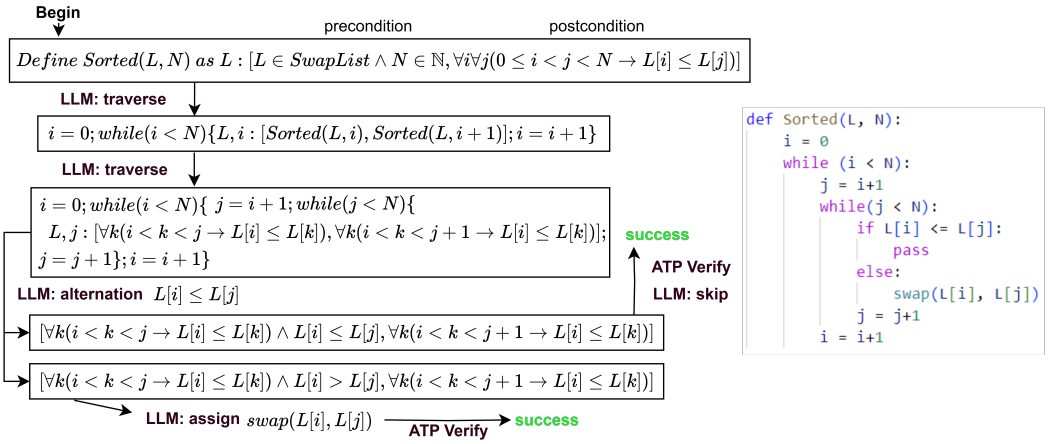

Fig. 10. Bubble sort algorithm with program refinement and the generated pseudo-code. The type of the variables can be extracted from the specification. The generated pseudo code can be translated to a domain-specific language with the formal specification.

between x and y to assign x or y with the mean of x and y. The proofs, including the invariants preserving and loop termination, are detailed in [3]. Compared to the standalone LLM-generated code, Refine4LLM verifies the code with its associated specifications. On the other hand, LLMs are prompted by the refinement laws and specifications. LLM selects the proper refinement law and generates the associated code automatically. The new specification will be formally generated based on the refinement laws by Refine4LLM after the ATPs verify the code.

10.2 Sorting Algorithm

The sorting algorithm is a classic problem in program refinement literature [14, 44]. We follow the definition in [14] to define a new type of array called *SwapList* that only allows swap operations:

$$swap : SwapList \rightarrow num \rightarrow num \rightarrow SwapList \tag{6}$$

Bubble Sort. We use the bubble sort and quick sort algorithms as examples to illustrate the extensibility of our refinement framework. For the bubble sort shown in Figure 10, the specification includes a *SwapList*, the length N of the *SwapList*, and the postcondition that the list elements should be sorted in ascending order. One refinement direction begins by adopting the traverse law that initializes variable $i = 0$, establishing that the sublist $L[0 : i]$ is a sorted list. Then, the refinement progresses by incrementally expanding this sorted list $L[0 : i]$ to $L[0 : i + 1]$, simultaneously decreasing the variant $N - i$. The traverse law can also be applied to refine the remaining specification, highlighting the iterative nature of both integer variables i and j, which iterates from a left bound to a right bound governed by constraints. Finally, LLM applies the alternation law to compare $L[i]$ and $L[j]$ to guarantee that $L[i]$ is always the smallest element in the remaining list. The traverse law, which combines the sequential composition law, iteration law, and assignment law, shows the advantages of deriving new advanced refinement laws based on the correctness-by-constructions [11]. The traverse law (detailed in Theorem 6.8) is derived for traversing the array elements from one index to another. It also simplifies the proof obligation by removing the requirement to decrease the variant. Note that the generated pseudo code can be translated to a domain-specific language with the formal specification.

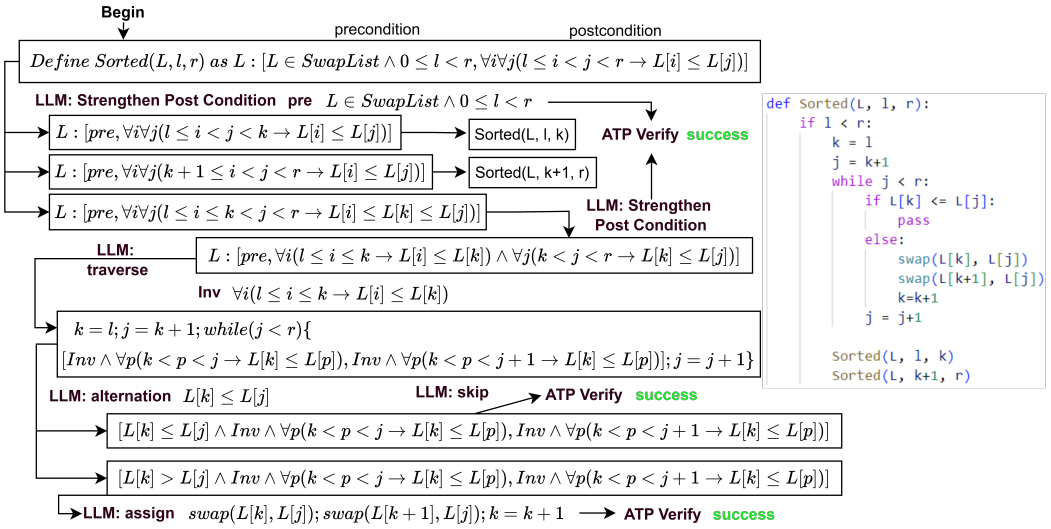

Fig. 11. Quick sort algorithm with program refinement and the generated pseudo-code.

Quick Sort. Another refinement direction involves decomposing the original problem into smaller but similar sub-problems, a method commonly recognized as recursion. As shown in Figure 11, the strengthen-post-condition law divides the *SwapList* into two parts, where the elements in the first part are less than or equal to $L[k]$ and those in the second part are greater or equal to $L[k]$. Each part is then sorted independently. The base situation is one-element sorting, which is trivial. The associated proof obligation for applying the strengthen-post-condition law involves demonstrating that these three new postconditions imply the original postcondition, which can be proved by structural induction. Further refinement seeks to disentangle the quantifiers i, j with the strengthen-post-condition law again and prove that the new postcondition can imply the original postcondition. The traverse laws can be applied with the invariant that $L[i] \leq L[k]$ and $L[k] \leq L[j]$ should be preserved when $L[k]$ changes. The algorithm will traverse the variable j with the invariant $\forall i(l \leq i \leq k \rightarrow L[i] \leq L[k]) \wedge \forall p(k < p < j \rightarrow L[k] \leq L[p])$. Then, the LLM generates the code to initialize $k = l; j = k + 1$ to satisfy the invariant and then assign j with $j + 1$ to decrease the variant $r - j$. Finally, the LLM uses the alternation law to compare the $L[k]$ and $L[j]$, similar to the bubble sort algorithm. The key difference is to reorder the sequence $L[k], L[k + 1], L[j]$ in the ascending order to preserve the invariant. If $L[k] > L[j]$, then after swapping the value of $L[k]$ *and* $L[j]$ and the value of $L[k+1]$ *and* $L[j]$, the new sequence will be in the ascending order. The swap operation for the list here is just a value swap, but we acknowledge that features such as pointers are important, and we will extend our method to consider such features, e.g., through separate logic, in future work.

10.3 Prime Factorization Problem

Functional abstractions are a common way for programmers to manage complexity. The program refinement specification naturally defines a functional abstraction's input and output. Besides, the sub-specifications in the program refinement procedure can also be regarded as small functional abstractions. For example, shown in Figure 12, the prime factorization problem may involve several sub-problems: prime number problem, factor number problem, and list product problem.

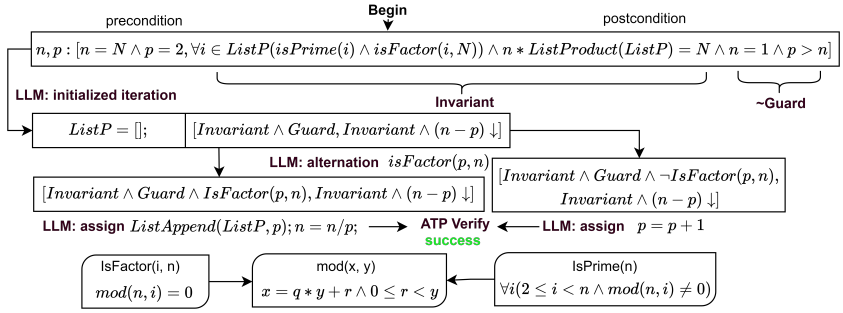

Fig. 12. Prime Factorization Problem with Program Refinement. It involves several sub-problems: prime number problem, factor number problem, modulo problem and list product problem.

Refine4LLM first interacts with the user to formalize the natural language specification. The description of the prime factorization problem is as follows:

Prime factorization is to break down the positive number into its factors until all are prime numbers.

The LLM auto-formalizes the problem with functional abstractions like *IsPrime*, *IsFactor*, and *ListProduct*. Some functions in the specification may have been refined and stored in the library in the past, and some may not. Each sub-specification has its precondition and postcondition, and they can be split into smaller specifications like *modulo* until all elements defined in their preconditions and postconditions have been refined to atomic elements in the defined language in Section 5. After formalizing the high-level specification, Refine4LLM will first search in the library to match the same specification and reuse the refined program if it exists. Then, a bottom-up algorithm will refine the remaining sub-specifications, synthesize the programs, and store them in the library from the smallest specification to the high-level specification. In this example, Refine4LLM will first refine the smallest specification of the problem *modulo* $mod(x, y)$, which can be used for refining the factor number problem and prime number problem. The refined programs with the specifications will be stored in the library and reused in future refinement.

For refining the prime factorization problem shown in Figure 12, the LLM first selects the initialized iteration law to initialize the variable n with N and $ListP$ with an empty array satisfying the condition $n \cdot ListProduct(ListP) = N$. Secondly, the LLM selects the alternation law with the condition *IsFactor(i,n)*. Although the condition ignores the *IsPrime(i,n)*, it can be proved that:

$$Invariant \wedge isFactor(i, n) \longrightarrow isPrime(i) \tag{7}$$

The intuitive proof is that if i is not a prime, there must exist k such that k is less than i and k is a factor of i. Because i is a factor of n, and n is a factor of N, k will also be a factor of N. This conclusion contradicts the invariant that all factors of N that are less than i have been considered and put into the $ListP$, which means i must be a prime number if it is a factor of n. It is a non-trivial resolution for the prime factorization algorithm to reduce the time of calculating the prime number.

Finally, the LLM assigns the n with n/p and appends the p to the $ListP$ to preserve the invariant. If p is not a factor of n, the LLM will increase the p to keep the variant $n - p$ decreasing. The generated pseudo-code is shown in Figure 13. Overall, this example illustrates that Refine4LLM can not only utilize the flexibility and rich knowledge of the LLM to generate better algorithms but also use the ATPs to generate verified code.

Prime Factorization Algorithm

```
N = input()
n = N
p = 2
ListP = []
while (p <= n):
    if n % p == 0:
        ListP.append(p)
        n = n/p
    else:
        p = p+1
```

Mask Distribution Algorithm

```
L, M = input()
i = 0
while(i<=length(L)):
    Q[i][0] = True
j = 0
while(j<=M):
    Q[j][0] = False

i = 1
while(i<=length(L)):
    j = 1
    while(j<=M):
        Q[i][j] = Q[i-1][j] or
          Q[i-1][j - L[i-1]]
```

Fig. 13. The pseudo-code of the prime factorization problem and the mask distribution problem.

10.4 Mask Distribution Problem

One challenge in applying program refinement to real-world scenarios is translating requirements, typically described in natural languages, into formal specifications. This translation process is often complex and can be a barrier to effectively using formal methods. However, the strength of LLMs sheds light on auto-formalization, which is the ability to interpret and analyze natural language. Leveraging this capability, Refine4LLM can minimize the effort required by users to engage with formal tools, making these processes more accessible and user-friendly. By utilizing the LLM's proficiency in natural language processing, Refine4LLM can bridge the gap between informal requirements and formal specifications, thereby streamlining the initial stages of program refinement. To demonstrate this approach, we illustrate a real-world example:

One company operates N factories, each with a specific production capacity for manufacturing masks. The company aims to distribute the masks equally between two countries, B and C, to ensure fairness. Each factory's production can only be assigned to one country for mask specification differences. We need to determine whether assigning the factories is possible so that each country receives an equal share of the total mask production.

With the description of the problem as the prompt, the GPT-4 [47] answers: *To determine if the masks from the factories can be evenly distributed to two countries, such that each factory's output is assigned to one country, we are effectively trying to see if the set of factory production capacities can be split into two subsets where the sum of each subset is equal. This is known as the "Partition Problem," which is a well-known problem in computer science.*

The GPT-4 automatically formalizes the question as follows:

$$Q(L, M) = \exists s(SubList(s, L) \land SumList(s) = M) \tag{8}$$

where L is a list containing each factory's output number, M is the target number for one country, and Q is a bool-type matrix to check whether M masks can be distributed to the front L factories. This statement asks whether a subset s of the list L exists such that the sum of the elements in s equals M. The recurrence relation is generated by the GPT-4 as follows:

$$Q[i][j] = Q[i-1][j] \lor Q[i-1][j - L[i-1]] \tag{9}$$

where $Q[i][j]$ denotes whether there exists a subset of the first i elements that sums to j and $L[k]$ represents the k^{th} element in the list L. The base cases are $\forall i, Q[i][0] = True$ and $\forall j, Q[0][j] = False$. The recurrence relation means that if either $Q[i-1][j]$ is true (the sum j can be achieved without the i^{th} element) or $Q[i-1][j - L[i-1]]$ is true (the sum j can be achieved by including the i^{th} element, then $Q[i][j]$ should be true.) To prove the recurrence relation is correct based on

the specification, we need to prove:

$$\begin{aligned} \exists s(SubList(s, L[0:i]) \land SumList(s) = j) \leftrightarrow \\ \exists s(SubList(s, L[0:i-1]) \land SumList(s) = j) \lor \\ \exists s(SubList(s, L[0:i-1]) \land SumList(s) = j - L[i-1]) \end{aligned} \tag{10}$$

The left-hand side states that there exists a subset s of the first i elements of the list L such that the sum of the elements in s equals j. The right-hand side states that there are two possible ways to achieve the sum j using the first i elements: (1) there exists a subset s of the first $i-1$ elements such that the sum of s is exactly j, or (2) there exists a subset s of the first $i-1$ elements such that the sum of s is exactly $j - L[i-1]$. The users need to check the specification and the base cases, while Refine4LLM can generate the program and perform the program refinement tasks automatically with the extended traverse law shown in Figure 13. Note that the pseudo-code is not completed and requires translation to a domain-specific language. The real-world example highlights the practical application of Refine4LLM and illustrates its potential to make formal methods more approachable.

11 Limitations

This section discusses the limitations of our current approach and potential future improvements.

11.1 Formal Specification

Refine4LLM is primarily designed to assist LLMs in generating robust code, not to develop formal specifications from scratch. It still relies on human input to provide formalized specifications, as is common with most formal methods tools. However, LLM significantly reduces the manual effort required for formalizing specifications and program refinement. While human involvement is necessary for defining correctness, Refine4LLM accelerates the refinement process and ensures the generated code adheres to the given specifications.

11.2 Law Selection and Code Generation

Refine4LLM heavily relies on the LLM's ability to the refinement law selection and the associated code generation. While Reinforcement learning (RL) techniques can enhance the law selection and proof search, a complete decoupling of law selection from code generation could result in inefficient or incorrect code generation. Besides, training a new LLM model for code generation with RL for law selection may be resource-intensive. Instead, constraints in the specifications need to guide the search, steering the synthesizer toward solutions that are not only correct but also easy to verify. Ensuring efficient law selection and code generation remains a challenge, and further improvements may also depend on the LLM's ability.

On the other hand, we focus on generating executable and verifiable code for this work. We mainly depend on the LLM's knowledge to generate optimized algorithms to pass the test cases in the specific time and space complexity requirements since synthesizing an optimized algorithm is challenging and could be our future research direction.

11.3 Verification

One limitation of the current system is the lack of proof for loop termination in some cases. If a refinement lacks such proof, we still consider the program *partially correct*. To address this, we propose additional iterative refinement laws in Section 6 to reduce cases where termination proofs are required since proving termination is a complex and often undecidable task. While this does not fully solve the problem, it alleviates the need for explicit termination proofs in many cases.

11.4 Scalability

The current implementation of Refine4LLM has scalability limitations due to the simplicity of its data structures and refinement laws. Its performance is tied closely to the capabilities of the underlying LLMs and automated theorem provers (ATPs). To address this, users are encouraged to interact with the LLM actively during program refinement by selecting appropriate laws, constructing proofs, and verifying the program's correctness. In future work, we will include more refinement laws like separation logic and other complex data structures. Developing more intuitive user interfaces for interacting with LLMs and ATPs during the refinement process could also enhance usability and user involvement, facilitating a more interactive and scalable refinement workflow.

12 Related Work

Our work is the first to combine ideas from two areas: program refinement and LLM. Refine4LLM builds on two core ideas from prior work: program refinement, which refines the high-level specification into executable code in several correctness-preserving steps, and mitigating LLM's hallucination, which involves different techniques to guide and verify LLM.

12.1 Program Refinement

As a long-standing task, program refinement can be traced back to [26, 31, 35]. The related theories [5, 44] are based on Hoare logic and the calculus of weakest preconditions. Some recent works propose a formalization of the refinement calculus with interactive theorem provers, such as [32] for Isabelle, and [2, 55] for Coq [6]. The goal of the development presented in [2] is to derive imperative programs by applying validated refinement rules in proof mode. Therefore, the final program design entangles the intermediate refinement steps and their proof of correctness. Users must specify loop invariants in [2]. In contrast, [55] allows the specification loop bodies as input-output relations and uses the weakest pre-specifications instead of the weakest preconditions to compute the proof obligations. Other works utilize refinement calculus on different applications, including [28] for compositional modeling and reasoning about reactive systems and [39] for the development of correct software product lines on object orientation and feature-oriented programming. However, Refine4LLM is an automated tool that leverages the creativity of LLMs and rigorous transformation of program refinement to provide an automatic and reliable code generator.

12.2 Mitigating LLM's hallucination

One of the critical challenges of LLMs is their tendency to *hallucinate*, which refers to generating information that is not only incorrect but often fabricated specious text. Prior works have proposed multiple ways to guide and verify LLMs for mitigating hallucinations. Prompting techniques like Chain-of-Thought [63], Tree-of-Thought [66], Graph-of-Thoughts [9] have been applied to provide example reasoning or relevant knowledge passing procedure for LLM. [37] sketches a self-monitoring and iterative prompting way that uses formal methods to detect the hallucination and steer the LLM to the correct specification. [16] builds the specialized prompt based on counterexamples provided by model checking and conducts code debugging and repairing based on LLMs. [10] builds Language Model Programming (LMP) to generalize prompting from pure text to combining text and scripting. Retrieval-based methods utilize external knowledge graphs or databases to help correct factual errors in LLMs' outputs [42, 69]. Some work uses multiple LLMs to solve one problem and relies on majority voting or consensus after a dialogue among the LLMs in natural language. Some studies use multiple LLMs to generate various data, including specifications and test cases, subsequently assessing the consistency [1, 43]. Other approaches involve using additional LLMs to review the output of a target LLM in a format akin to a debate [68]. Compared

to previous methods, Refine4LLM utilizes the formal verification and refinement calculus to avoid the hallucination formally.

12.3 Functional Synthesis

The program synthesis community has proposed many approaches for functional synthesis. Synthesis based on *verification* like [59] is an interplay of verification and synthesis in a program that is converted into some predicates that must hold for all inputs. *Counterexample* Guided Inductive Synthesis like Sketch [58] (aka CEGIS [57]) is a form of generate and check, where a synthesizer generates candidate programs that are checked by an off-the-shelf checking procedure. Synthesis with *refinement types* [49] specifies the complex properties of a program and catches errors early, even when the program has not been completed. The associated round-trip type checking prunes the search space efficiently at each step, and condition abduction incrementally generates programs. *Deductive synthesis* systems are generally challenging since rule selection is usually tricky. The Spiral system [50], designed for signal processing kernels, incorporates a lot of domain knowledge into the rules and the application strategies, enabling the fully automated transformation from high-level specifications to implementations. [23] introduces the Fiat system built on the Coq and leverages the automation that Coq's tactic language to provide a high degree of automation. The recent work [12, 29] synthesizes library functions that capture common functionality from a corpus of programs. In contrast, Refine4LLM builds high-level refinement laws of synthesizing the program to guide and verify the LLM since the LLM has great capability for code generation.

12.4 Reinforcement Learning

Several approaches have explored the intersection of reinforcement learning (RL) and formal methods, particularly in proof search and law selection. For instance, Chen et al. [17] applied RL to learn heuristics for applying rules in relational verification, demonstrating the potential for RL to guide complex decision-making processes in formal verification tasks. Additionally, traditional synthesis techniques have often been employed alongside RL techniques. [19] introduces Model Predictive Program Synthesis (MPPS), which utilizes program synthesis to generate guiding programs for program synthesis guided reinforcement learning automatically. Meanwhile, [65] formulates program synthesis as a reinforcement learning problem and proposes a novel variant of the policy gradient algorithm that incorporates feedback from a deductive reasoning engine. Such hybrid methods offer a way to avoid costly repeated interactions with a language model, as RL can learn efficient rule-application patterns through trial and error. However, the core challenge of program refinement is verifiable code generation. Whether the selected law is applicable depends on the code generation ability, but training a new LLM with RL is resource-intensive. Therefore, Refine4LLM that interacts with the state-of-the-art LLM offers a more approachable solution.

13 Conclusion

We have presented a tool called Refine4LLM for mostly *automated* generation of *verified* code with LLMs, Coq, and ATPs. We formally transform the specifications into code based on our refinement calculus and LLMs. Our approach extends the formal refinement calculus and builds active prompts to the LLMs. Refine4LLM uses ATPs to verify the refinement condition and the code based on the specification. Our experiments show that our method can more efficiently generate more robust and correct code than the state-of-the-art LLMs.

Acknowledgment

This research is supported by the National Research Foundation Singapore under its AI Singapore Programme (Award Number: AISG3-RP-2022-030).

References

[1] Pranjal Aggarwal, Aman Madaan, Yiming Yang, and Mausam. 2023. Let's Sample Step by Step: Adaptive-Consistency for Efficient Reasoning and Coding with LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 12375–12396. https://doi.org/10.18653/v1/2023.emnlp-main.761

[2] João Alpuim and Wouter Swierstra. 2018. Embedding the refinement calculus in Coq. *Science of Computer Programming* 164 (2018), 37–48. https://doi.org/10.1016/j.scico.2017.04.003 Special issue of selected papers from FLOPS 2016.

[3] Anonymous. 2024. LLM aided Program Refinement. https://sites.google.com/view/refine4llm.

[4] Ralph-Johan J. Back, Abo Akademi, J. Von Wright, F. B. Schneider, and D. Gries. 1998. *Refinement Calculus: A Systematic Introduction* (1st ed.). Springer-Verlag, Berlin, Heidelberg.

[5] Ralph-Johan R Back and Joakim von Wright. 1990. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing* 2 (1990), 247–272.

[6] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. 1999. The Coq proof assistant reference manual. *INRIA, version* 6, 11 (1999), 17–21.

[7] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 171–177.

[8] Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness.

[9] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michał Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16 (Mar 2024), 17682–17690. https://doi.org/10.1609/aaai.v38i16.29720

[10] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (jun 2023), 24 pages. https://doi.org/10.1145/3591300

[11] Tabea Bordis, Tobias Runge, Alexander Kittelmann, and Ina Schaefer. 2023. Correctness-by-Construction: An Overview of the CorC Ecosystem. *Ada Lett.* 42, 2 (apr 2023), 75–78. https://doi.org/10.1145/3591335.3591343

[12] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

[13] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 427–442.

[14] Michael Butler and Thomas Långbacka. 1996. Program derivation using the refinement calculator. In *Theorem Proving in Higher Order Logics*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–108.

[15] DAVID Carrington, I Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. 1998. A program refinement tool. *Formal Aspects of Computing* 10 (1998), 97–124.

[16] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C. Cordeiro. 2023. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. arXiv:2305.14752 [cs.SE]

[17] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational verification using reinforcement learning. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 141 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360567

[18] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[19] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*. Springer, 587–610.

[20] Koen Claessen, Reiner Hähnle, and Johan Mårtensson. 2002. Verification of hardware systems with first-order logic. In *Proceedings of the CADE-18 Workshop-Problem and Problem Sets for ATP*. Citeseer.

[21] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[23] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 689–700. https:

//doi.org/10.1145/2676726.2677006
[24] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (may 2005), 365–473. https://doi.org/10.1145/1066100.1066102
[25] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975
[26] Edsger W Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. 1976. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs.
[27] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. A Static Evaluation of Code Completion by Large Language Models. arXiv:2306.03203 [cs.CL]
[28] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. 2020. The refinement calculus of reactive systems toolset. *International Journal on Software Tools for Technology Transfer* 22 (2020), 689–708.
[29] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080
[30] Maxim Enis and Mark Hopkins. 2024. From LLM to NMT: Advancing Low-Resource Machine Translation with Claude. arXiv:2404.13813 [cs.CL]
[31] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81.
[32] Simon Foster, Jonathan Julián Huerta y Munive, and Georg Struth. 2020. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *Relational and Algebraic Methods in Computer Science: 18th International Conference, RAMiCS 2020, Palaiseau, France, October 26–29, 2020, Proceedings 18*. Springer, 169–186.
[33] GitHub. 2023. GitHub Copilot. https://github.com/features/copilot
[34] Reiner Hähnle. 2016. *Quo Vadis Formal Verification?* Springer International Publishing, Cham, 1–19. https://doi.org/10.1007/978-3-319-49812-6_1
[35] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
[36] Pei Huang, Haoze Wu, Yuting Yang, Ieva Daukantas, Min Wu, Yedi Zhang, and Clark Barrett. 2024. Towards Efficient Verification of Quantized Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 19 (Mar. 2024), 21152–21160. https://doi.org/10.1609/aaai.v38i19.30108
[37] Susmit Jha, Sumit Kumar Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, and Sandeep Neema. 2023. Dehallucinating Large Language Models Using Formal Methods Guided Iterative Prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*. 149–152. https://doi.org/10.1109/ICAA58325.2023.00029
[38] Samia Kabir, David N. Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2024. Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 935, 17 pages. https://doi.org/10.1145/3613904.3642596
[39] Maximilian Kodetzki, Tabea Bordis, Tobias Runge, and Ina Schaefer. 2024. Partial Proofs to Optimize Deductive Verification of Feature-Oriented Software Product Lines. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems* (Bern, Switzerland) *(VaMoS '24)*. Association for Computing Machinery, New York, NY, USA, 17–26. https://doi.org/10.1145/3634713.3634714
[40] Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*. Springer, 1–35.
[41] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=1qvx610Cu7
[42] Ariana Martino, Michael Iannelli, and Coleen Truong. 2023. Knowledge Injection to Counter Large Language Model (LLM) Hallucination. In *The Semantic Web: ESWC 2023 Satellite Events*, Catia Pesquita, Hala Skaf-Molli, Vasilis Efthymiou, Sabrina Kirrane, Axel Ngonga, Diego Collarana, Renato Cerqueira, Mehwish Alam, Cassia Trojahn, and Sven Hertling (Eds.). Springer Nature Switzerland, Cham, 182–185.
[43] Marcus J. Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2024. Beyond Accuracy: Evaluating Self-Consistency of Code Large Language Models with IdentityChain. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=caW7LdAALh
[44] Carroll Morgan. 1990. *Programming from Specifications*. Prentice-Hall, Inc., USA.
[45] C. Morgan, K. Robinson, and P. Gardiner. 1988. *On the Refinement Calculus*. Techllical Monograph. https://web.comlab.ox.ac.uk/files/3391/PRG70.pdf

[46] OpenAI. [n. d.]. Introducing GPTs. https://openai.com/blog/introducing-gpts.
[47] OpenAI and co. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[48] Ruchika Pandey, Prabhat Singh, Raymond Wei, and Shaila Shankar. 2024. Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects. arXiv:2406.17910 [cs.SE] https://arxiv.org/abs/2406.17910
[49] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (June 2016), 522–538. https://doi.org/10.1145/2980983.2908093
[50] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306
[51] Bernhard Reus. 2016. *The WHILE-Language.* Springer International Publishing, Cham, 29–45. https://doi.org/10.1007/978-3-319-27889-6_3
[52] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. 2023. Mathematical discoveries from program search with large language models. *Nature* (2023), 1–3.
[53] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[54] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick Kourie, and Bruce W. Watson. 2019. Tool Support for Correctness-by-Construction. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 25–42.
[55] Boubacar Demba Sall, Frédéric Peschanski, and Emmanuel Chailloux. 2019. A Mechanized Theory of Program Refinement. In *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019, Proceedings* (Shenzhen, China). Springer-Verlag, Berlin, Heidelberg, 305–321. https://doi.org/10.1007/978-3-030-32409-4_19
[56] Stephan Schulz. 2002. E - a Brainiac Theorem Prover. *AI Commun.* 15, 2,3 (aug 2002), 111–126.
[57] Armando Solar-Lezama. 2018. Lecture 10: Introduction to Functional Synthesis. Available at https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture10.htm.
[58] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* 136–148. https://doi.org/10.1145/1375581.1375599
[59] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* 313–326. https://doi.org/10.1145/1706299.1706337
[60] Wouter Swierstra and Joao Alpuim. 2016. From Proposition to Program: Embedding the Refinement Calculus in Coq. In *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13.* Springer, 29–44.
[61] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts.* 1–7.
[62] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs.CL]
[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
[64] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is Inevitable: An Innate Limitation of Large Language Models. arXiv:2401.11817 [cs.CL] https://arxiv.org/abs/2401.11817
[65] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. 2021. Program synthesis guided reinforcement learning for partially observed environments. *Advances in neural information processing systems* 34 (2021), 29669–29683.
[66] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *Advances in Neural Information Processing Systems* 36 (2023). Publisher Copyright: © 2023 Neural information processing systems foundation. All rights reserved.; 37th Conference on Neural Information Processing Systems, NeurIPS 2023 ; Conference date: 10-12-2023 Through 16-12-2023.

[67] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2024. Instruction Tuning for Large Language Models: A Survey. arXiv:2308.10792 [cs.CL] https://arxiv.org/abs/2308.10792

[68] Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. 2023. Cumulative Reasoning With Large Language Models. *arXiv preprint arXiv:2308.04371* (2023).

[69] Ruochen Zhao, Xingxuan Li, Shafiq Joty, Chengwei Qin, and Lidong Bing. 2023. Verify-and-Edit: A Knowledge-Enhanced Chain-of-Thought Framework. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 5823–5840. https://doi.org/10.18653/v1/2023.acl-long.320

[70] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]