MoonBit: Explore the Design of an AI-Friendly Programming Language

Haoxiang Fei
International Digital Economy Academy
Shenzhen, China
feihaoxiang@idea.edu.cn

Yu Zhang
International Digital Economy Academy
Shenzhen, China
zhangyu@idea.edu.cn

Hongbo Zhang
International Digital Economy Academy
Shenzhen, China
bob.hongbo.zhang@idea.edu.cn

Yanlin Wang
Sun Yat-sen University
Guangzhou, China
wangylin36@mail.sysu.edu.cn

Qing Liu
International Digital Economy Academy
Shenzhen, China
liuqing@idea.edu.cn

ABSTRACT

MoonBit, a new general-purpose programming language designed for cloud and edge computing, was initiated in late 2022, coinciding with the announcement of ChatGPT. Language models like GPT, capable of producing practical programs, are revolutionizing the way we write programs and interact with computers. However, significant challenges persist, such as the models' inability to understand the global context of a whole project with its dependencies, the need for human verification and correction of generated code, and the lack of assurance in meeting basic requirements like syntactic correctness.

In this paper, we explore the design of the MoonBit language highlighting its AI integration, emphasizing the synergy between traditional code intelligence and large language model capabilities. We also introduce a real-time, semantics-based sampler to guide the inference process of language models. This approach ensures the generated programs are both syntactically correct and free from obvious semantic flaws, such as type errors. Crucially, this has been achieved with minimal impact on overall performance. Our evaluation demonstrates a notable improvement in code quality, achieved without sacrificing the models' responsiveness.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

KEYWORDS

Large Language Model, Program Synthesize, Static Analysis

ACM Reference Format:
Haoxiang Fei, Yu Zhang, Hongbo Zhang, Yanlin Wang, and Qing Liu. 2024. MoonBit: Explore the Design of an AI-Friendly Programming Language. In *2024 International Workshop on Large Language Models for Code (LLM4Code '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643795.3648376

1 INTRODUCTION

MoonBit is a general-purpose programming language with its development platform for cloud and edge computing [13]. It compiles to WebAssembly, and aims to be *fast*, *compact* and *user-friendly*. For a modern programming language to be user-friendly, its integration with AI-powered assistants is indispensable.

In the realm of code intelligence, the development of Large Language Models (LLMs) stands as a significant milestone, marking a new era in the seamless fusion of contemporary programming languages and AI-assisted technologies. LLMs have shown remarkable capabilities in code understanding and code generation tasks. Many code assistants powered by LLMs, such as GitHub Copilot[1] (which is powered by Codex [4]), have been widely used by programmers to improve their coding experience.

Despite the advanced capabilities of LLMs, they still exhibit several weaknesses. One of the primary limitations is their considerable operational costs, making their deployment an expensive endeavor for many organizations. Furthermore, LLMs occasionally yield inaccurate or misleading responses, necessitating manual review and correction to ensure reliability. Even worse, the integration of LLMs with emerging programming languages presents a significant challenge, a phenomenon particularly pronounced in the context of low-resource programming languages [3]. This is because LLMs tend to inaccurately blend syntax and features from various languages in code generation for these languages [1].

In this paper, we use MoonBit as a case study to explore approaches to integrating LLMs with emerging programming languages. The design of MoonBit's syntax and frontend is specifically engineered to enhance the efficiency of LLMs, aiming to reduce both response latency and operation costs. On the one hand, MoonBit's code features a flat and minimally nested structure. This enhances the efficiency of *key-value caching*, which is a key mechanism to accelerate inference by storing previous model states to circumvent redundant computations [8]. On the other, the type definitions

[1]https://github.com/features/copilot

and function signatures in MoonBit are explicitly present at the toplevel, thereby allowing a more flexible *retrieval-based prompt augmentation* [10].

Additionally, we have developed a semantics-based sampler that works closely with static program analyzers. This approach combines the conventional code intelligence with the power of LLMs to eliminate syntactic and semantic errors, significantly improving the accuracy of code completion. Meanwhile, the system is designed for incremental and real-time updates, where adding new knowledge and revising prompts does not require re-evaluating existing ones. This design ensures that the sampler keeps pace with the code generation process of LLMs, effectively avoiding any slowdowns.

In this paper, we make the following contributions:

- We present the design of MoonBit with an emphasis on features that improve the efficiency of the key-value cache (§2.1), and the flexibility of prompt augmentation (§2.2).
- We design a semantics-based language sampler that integrates the strengths of static analyzers and LLMs to increase the accuracy of generated code (§3).

In the following part of this paper, we first discuss in more details the design of MoonBit (§2), and the implementation of the semantics-based sampler (§3). We then evaluate its performance on some example programs and demonstrate that it meets the expectation above (§3.4). Finally, we discuss related work in §4, and possible future work in §5.

2 AI-DRIVEN LANGUAGE DESIGN

MoonBit is designed with an emphasis on clarity and simplicity. It particularly emphasizes a clear distinction between *toplevel* and *local* definitions with mandatory type signatures for defitions at the toplevel. MoonBit also adopts *structural interface implementation*, where a type implements an interface by implementing its methods, thus eliminating the necessity for extra nested code blocks. This section explains the rationale behind these design choices and demonstrates their importance in facilitating MoonBit's integration with LLMs.

2.1 KV Cache Affinity

The utilization of key-value caches (KV cache) plays a crucial role in enhancing the efficiency of LLMs' inference processes. In the context of an autoregressive language model, the KV cache stores the key and value tensors produced during each step of the autoregressive generation [8]. This means a KV cache can be modeled as a *linear* array of past intermediate results.

However, a variety of programming languages, including Java, C++ and Rust, provide syntax and semantic structures that allow for code blocks with multiple levels of nesting. Although this feature facilitates the organization of code, it poses challenges for programmers and LLMs in building up the codebase *linearly*.

In the example depicted in Figure 1a, a programmer is implementing the method `think` of the trait `Agent` for the type `Llama`. They discover that type `Llama` lacks the `generate` method, which is defined in the trait `LLM`. Given their position within a nested code block, they are required to move out to the toplevel to implement the trait `LLM` for the type `Llama`. However, in the context of LLMs, altering a prompt several functions back leads to invalidation of the relevant KV cache segment, highlighted in red in the figure. Following such a modification, all these functions undergo re-evaluation, leading to waste of computational resources, and more critically, an extended delay in output.

```
struct Llama { ... }
trait Agent { ... }
impl LLM for Llama ...
impl Agent for Llama {
  fn act(&self) ...
  fn info(...) ...
  fn ...
  fn think(&mut self) {
    self.generate
```

(a) Rust

```
struct Llama { ... }
trait Agent { ... }
fn act(self: Self) ...
fn info(...) ...
fn ...
fn generate(...) ...
fn think(self: Self) {
  self.generate
```

(b) MoonBit

Figure 1: Linearity of Rust and MoonBit

By contrast, as shown in Figure 1b, MoonBit enables programmers and LLMs to develop their programs *linearly*, without frequent back-and-forth navigation. With the structural interface, functions implementing an interface are not confined to a specific code block. This allows a nearly linear generation of interfaces and their respective implementations, thereby effectively minimizing the KV cache misses.

2.2 Context Modularity

The purpose of requiring full annotation on toplevel functions is twofold. First, with these type signatures, MoonBit enjoys the advantage of providing LLMs with prompts that are not only smaller but also more contextually relevant. It is well known that even though LLMs are getting larger and larger context windows [9], evaluating a large prompt remains costly (51.346 s to evaluate 9516 tokens on a M2 Ultra). In MoonBit, the primary focus of the context is on the signatures of toplevel functions, rather than their full function bodies. This practice enables a more flexible retrieval-based prompt augmentation, like the one suggested by Shrivastava et al. [10] or by Ding et al. [5]. Moreover, by using static analysis and heuristics, it is feasible to provide LLMs with a more concise yet relevant part of the toplevel context. This method substantially lowers the response latency while maintaining the accuracy of the generated output.

Second, the process of code generation in MoonBit is executed in a two-stage approach. In the first stage, LLMs are tasked with generating the *signatures* of toplevel definitions as complete as possible. Subsequently, LLMs proceed to generate the body for each toplevel structure. The explicit signatures of toplevel definitions permits the independent synthesis and type-checking of each function body. This top-down approach enhances the flexibility in scheduling and reorganizing these specific tasks.

3 SEMANTICS-BASED SAMPLING

In this section, we explore the development of the language sampler for MoonBit. Unlike traditional methods, MoonBit's sampler

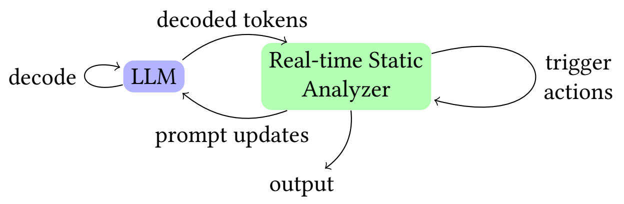

Figure 2: LLM and static analyzer cooperation architecture

collects and utilizes information from both local and global context, and collaborates closely with both a parser and a static analyzer. As the LLM generates a new token, it utilizes real-time feedback from the parser and static analyzer, thereby ensuring that each generated token is not only syntactically correct but also free from obvious type errors. The high-level architecture of the cooperation between the LLM and the analyzer is shown in Figure 2. Traditional approaches, which lack this collaboration and dump decoded tokens directly, often fall short, leading to the generation of erroneous programs. In §3.4, we provide an evaluation that compares the results of our MoonBit sampler with those of the raw decoder.

3.1 Real-time Analysis

To provide real-time feedback for MoonBit's sampler, we have specially developed an *incremental parser* and a *semantic parser*. The incremental parser is designed to efficiently build a partial Abstract Syntax Tree (AST) out of any valid program prefixes, and can resume parsing when new tokens arrive. On the other hand, the semantic parser offers semantic information, such as types and variables in scope, by performing static analysis on the partial AST. This process is made possible because of MoonBit's *fault tolerant* type system, which ensures that type inference is not impeded by prior type errors. Thus, the incremental parser and the semantic parser can continuously supply useful information to the sampler.

3.2 Local Sampling

The sampling process from local context involves repeated activation of the incremental parser, which works as follows. Given a program prefix p, the incremental parser processes it to produce a partial AST t, a parsing state s, and a partial token w. Upon receiving an additional code snippet Δp, the incremental parser resumes from the state s, using the combination of Δp and the partial token w as its input. As a result, the incremental parser generates a new triple (t', s', w'), which is stored as the current state.

The local sampler improves its accuracy by exploiting the incremental parser to verify the validity of the tokens generated from the LLM's vocabulary. Given a prompt p, the LLM first produces a list of candidate tokens L from its vocabulary. To validate L, the incremental parser is then applied to p to produce a triple t, s, w. Subsequently, each token $l \in L$ is processed by the parser: if l is accepted, it is considered valid; otherwise, its sampling probability is reduced to zero. Thus, the generated code snippet is guaranteed to be syntactically correct. This approach, similarly applied with the semantic parser, ensures the variables are used within their scope and reduces the likelihood of type errors in the generated code snippets, enhancing their overall quality.

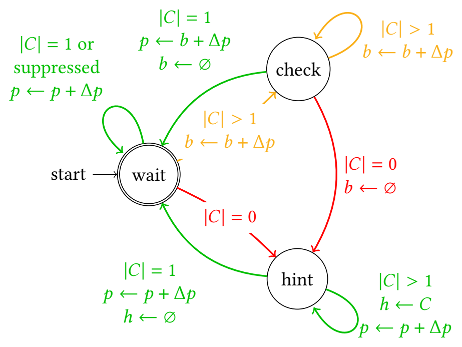

Figure 3: State transition diagram for the completion engine

3.3 Global Sampling

In contrast to local sampling, incorporating global context into sampling poses greater challenges. The large size of global context can be detrimental to the efficiency of real-time code generation. Our method for global sampling achieves a balance between comprehensive understanding of the global context and optimal performance by utilizing a *completion engine*.

To fulfill this objective, our model is designed to speculatively select a token at the beginning, permitting potentially incorrect ones. In cases where the token does not match with any of the completion suggestions from the completion engine, the function of which will be explained shortly, the prompt is augmented with a list of completion candidates as in Listing 1. These augmentations are placed near the end of generation (caret) to minimize KV cache invalidation.

```
...
// person has the following methods/fields:
// age, employer
person.
```

Listing 1: Example Augmentation

The completion engine operates in three states: *waiting*, *checking*, and *hinting*. The set of candidates at caret position is denoted as C, with $|C|$ representing the size of the set. The decoding process involves a prompt that is comprised of three elements: the speculation buffer b, the hint h, and the prompt p. The token generated by the LLM is denoted as Δp. The state transitions of the completion engine are shown in Figure 3.

The state of the completion engine is initialize to be *waiting*. When the number of completion candidates $|C|$ exceeds one, the state transitions to *checking*. In this state, generation continues until only one matching candidate remains. During *checking*, all generated tokens Δp_i are stored in the speculation buffer b. If only one candidate matches, the engine appends the content of the speculation buffer b to the prompt p, and clears the buffer b. In the absence of matching candidates, the engine moves to *hinting*, informing the LLM about the available candidates by assigning the string representation of C to h. In *hinting*, every LLM-generated token Δp is appended to the prompt p, as it is sampled to match the completion candidates. An example edit sequence is shown in Listing 2.

```
let x = p1.          // by LLM
let x = p1.sq        // by LLM, in speculation
let x = p1.~~sq~~        // |C| = 0, reset speculation
// p1 has the following methods: take_sqrt, ...
let x = p1.take_sqrt // by LLM
let x = p1.take_sqrt // by LLM or auto-completion
```

Listing 2: Possible editing sequence of program

3.4 Performance Evaluation

We select CodeLlama-34B [9] as our base model. Inference is performed on a 5-bit quantized version of the model in the GGUF format, on a M2 Ultra CPU with 192 GB unified memory, using llama.cpp [6].

Our evaluation involves 33 relatively simple coding tasks that do not trigger prompt augmentations, as all global definitions fall within the model's context window. The tasks are processed using both the raw decoder and the decoder augmented with MoonBit's semantics-based sampler. The outcomes are detailed in Table 1. Notably, we observed a significant improvement in compilation rate of the output programs, with a modest performance penalty of approximately 3%.

	Raw decoder	Semantics-based sampling
Performance	12.10 t/s	11.75 t/s
Compilation Rate	43.75%	56.25%

Table 1: Performance evaluation of semantics–based sampler

For tasks where prompt augmentation is involved, we observed an average delay of approximately 0.86 s per prompt augmentation. However, it is commonly observed that each distinct type of prompt augmentation generally triggers only once. This is because the model adapts and learns from the context after the augmentation.

Based on the aforementioned results, an equation can be formulated to calculate the time required for the model to generate k tokens. Given the length p of the prompt , the prompt evaluation rate r_p, the generation rate r_k, and count of global identifiers absent in the prompt m, the time t_k to compute the next k tokens can be determined as follows:

$$t_k = \frac{p}{r_p} + \frac{k}{0.97 \times r_k} + m \times 0.86\,\mathrm{s}$$

4 RELATED WORK

Prior works on combining static information with LLM inference have shown an effective increase in the accuracy of the generated code [1, 7, 11]. Many of these works suggest using completion engine [7, 11] or monitors [1] to guide the token generation process of large language models, and prune "invalid" tokens so that the output is type-consistent or satisfy the constraints generated by static analyzer.

However, these works are limited in their scopes. Synchromesh focuses on SQL-like languages [7], hence its insufficiency to handle the global-local duality presented in MoonBit. Monitor-Guided-Decoding [1] primarily addresses completions triggered by special characters, while others [11] will request for completion on *all* possible candidates suggested by LLM. These limitations underscore the need for a more general approach to combine static analysis with LLMs for general programming languages.

In addition to the inference-time approaches above, recent research also reveals the potential of using static analysis at global context to guide the code generation and testing process [2, 5, 10, 12]. For example, Eric Zelikman et al. use static analysis to generate a dependency graph between toplevel function signatures, and group these function signatures according to their dependencies to allow faster test and verification of the generated results [12]; Bairi et al. use static analysis to retrieve temporal and spatial context to guide LLMs to generate repository level code-edits. [2]

5 CONCLUSIONS AND FUTURE WORK

Our exploration into the MoonBit's AI-first design has yielded promising results, notably in generating accurate programs using the CodeLlama-34B [9] model combined with domain expertise. The standout achievement is the program's ability to decode at speeds comparable to the original decoder, while ensuring syntax accuracy and greatly enhancing semantic precision through live semantic analysis.

Furthermore, we are in the process of creating a package manager, which is instrumental in aggregating additional data to finetune the model. Future objectives include the enhancement of MoonBit's AI capabilities, incorporating functionalities such as code assistance, review, question and answer interfaces, test generation, and validation mechanisms. In parallel, we are planning a fast interpreter to provide real-time feedback during runtime. This will further increase the reliability of the AI-generated code.

REFERENCES

[1] Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Thirty-Seventh Conference on Neural Information Processing Systems.*

[2] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. CodePlan: Repository-level Coding Using LLMs and Planning. In *NeurIPS 2023 Foundation Models for Decision Making Workshop.*

[3] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2023. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. https://doi.org/10.48550/arXiv.2308.09895 arXiv:2308.09895 [cs]

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs]

[5] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. https://doi.org/10.48550/arXiv.2212.10007 arXiv:2212.10007 [cs]
[6] Georgi Gerganov. 2023. Llama.Cpp. ggml.ai.
[7] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2021. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*.
[8] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. *Proceedings of Machine Learning and Systems* 5 (March 2023).
[9] 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. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs]
[10] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-Level Prompt Generation for Large Language Models of Code. https://doi.org/10.48550/arXiv.2206.12839 arXiv:2206.12839 [cs]
[11] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *ESEC/FSE 2023*. ACM, San Francisco, CA, USA. https://doi.org/10.1145/3611643.3616271 arXiv:2309.00608 [cs]
[12] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic Reasoning with Language Models by Composing Decompositions. In *Thirty-Seventh Conference on Neural Information Processing Systems*.
[13] Hongbo Zhang. 2023. MoonBit: The Fast, Compact & User Friendly Language for WebAssembly. International Digital Economy Academy.