Dependent Types and the Future of Programming

As large language models (LLMs) become increasingly integrated into programming workflows, a new challenge arises: ensuring the correctness and reliability of AI-generated code. This essay explores the potential of dependent types as a formal method for expressing specifications and correctness conditions in future programming systems. It argues that the fusion of LLMs with dependently typed languages may enable a hybrid paradigm that combines statistical generation with symbolic verification.

1 Introduction

The emergence of large language models (LLMs), such as GPT-4 [1], Codex [2], and AlphaCode [3], is fundamentally transforming software development. These models generate code with minimal user input and are capable of solving programming problems across a wide range of domains. However, LLMs remain fundamentally statistical: they generate outputs that are plausible rather than guaranteed to be correct. This introduces significant risks, especially when deployed in safety-critical or security-sensitive systems.

2 Dependent Types and Formal Guarantees

Dependent types offer a path forward by enabling correctness to be encoded directly in the type system. In dependently typed languages such as *Coq* [6], *Agda* [5], *Idris* [4], and $F*$ [7], types can express propositions about values and behaviour. A program that type-checks in such a system is also a proof that it satisfies its specification. This aligns with the *Curry-Howard correspondence*, where types are propositions and programs are proofs.

Historically, the use of dependent types has been hindered by their steep learning curve and the labor-intensive nature of writing proofs. However, LLMs present an opportunity to automate the generation of such proofs, transforming what was once an esoteric method into a practical tool for everyday development.

3 LLMs and Verified Code Generation

We can envision a future workflow in which human developers write high-level specifications—expressed as dependent types—and LLMs generate candidate implementations. These implementations are then verified mechanically by the type checker. If the generated code fails to type-check, it is rejected outright; if it passes, the specification has been met *by construction*.

This model shifts the role of the programmer from writing algorithms to designing types and constraints. Programming becomes an exercise in intent: the human defines what must be true, and the LLM fills in the how.

Such an approach also fits naturally within the paradigm of *proof-carrying code* [8], where code is accompanied by formal proofs of its correctness. LLMs could act as synthesis engines,

guided and constrained by a dependently typed specification language, producing not just code but also evidence of its validity.

4 Symbolic–Statistical Hybrid Systems

The integration of LLMs with type theory represents a hybridisation of statistical and symbolic paradigms. Where LLMs are flexible, creative, and fluent in natural language, type systems are precise, rigid, and grounded in formal logic. This hybrid approach is already being explored in interactive theorem proving, where machine learning assists in tactic selection and proof search [9, 10].

Future systems might feature models trained not just on code but on proofs and type-correct derivations, capable of navigating large spaces of program behavior under strong correctness guarantees. Such systems could use LLMs as *assistants* within a dependently typed environment, handling the brute-force search while humans guide specification and structure.

5 Conclusion

Dependent types and large language models originate from very different traditions—symbolic logic and statistical inference—but their combination may define the next era of programming. As LLMs continue to automate more of the programming process, the demand for tools that ensure trust, safety, and correctness will only grow. Dependent types offer the formal apparatus to meet that demand, not by replacing LLMs, but by constraining and verifying them. This synergy could lead to a programming paradigm where correctness is a first-class concern, and programming is specification-driven by default.

References

[1] OpenAI. (2023). *GPT-4 Technical Report*. https://openai.com/research/gpt-4

[2] Chen, M., Tworek, J., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[3] Li, Y., et al. (2022). Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814*.

[4] Brady, E. (2017). *Type-Driven Development with Idris*. Manning Publications.

[5] Bove, A., Dybjer, P., & Norell, U. (2009). A brief overview of Agda. In *TPHOLs 2009*.

[6] The Coq Development Team. (2023). *The Coq Proof Assistant*. https://coq.inria.fr

[7] Swamy, N., Hriţcu, C., et al. (2016). Dependent types and multi-monadic effects in F*. In *POPL*.

[8] Necula, G. C. (1997). Proof-carrying code. In *Proceedings of POPL*.

[9] Gauthier, T., Kaliszyk, C., & Urban, J. (2017). Deep reinforcement learning for proof search in Coq. *arXiv preprint arXiv:1805.07563*.

[10] Avigad, J., et al. (2020). The Lean Theorem Prover. In *Intelligent Computer Mathematics (CICM)*.