EPIC: Formalizing a Parallel Lambda Calculus

CS4991 Capstone Report, 2025

Jamie Fulford
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
ase6gh@virginia.edu

ABSTRACT

Large language models have become central to modern scripting languages but require complex manual parallelization to achieve good performance. Finding ways to automatically parallelize these calls while preserving program semantics has proven challenging and error prone. To solve this, my team and I developed EPIC, a lambda calculus with an opportunistic evaluation strategy that is parallel by default. I formalized this calculus in the Rocq proof assistant using mutual induction principles and reduction relations to handle external calls and streaming data through Church encodings. Through this formalization, I proved key properties like confluence and preservation of well-formedness. Future work includes extending the formalization with primitive values adding types to the language.

1. INTRODUCTION

Scripts that compose external calls, particularly to large language models, have become increasingly central to modern software development. These scripts often require complex orchestration of API calls to remote LLM services, which can take several seconds each, and a single script might make dozens of such calls. While parallelizing these calls can dramatically improve performance, doing so manually is challenging and error-prone.

The EPIC (Opportunistically Parallel Lambda Calculus) project addresses this challenge by providing a core calculus that automatically exploits parallelism in programs with external calls. However, automatically parallelizing programs requires rigorous guarantees that the parallel execution preserves the program's meaning. To help establish these guarantees, I worked with researchers at the University of Pennsylvania to formalize EPIC's semantics in the Rocq (formerly known as Coq) proof assistant.

A programming language's formal semantics provides precise mathematical rules for how programs execute. For EPIC, these rules must specify both how external calls behave and when they can safely run in parallel. The foundation is a precise model of external function calls within the language's operational semantics, along with proofs that parallel execution always produces equivalent results.

2. RELATED WORKS

Mell, et al. [1] develop EPIC, a lambda calculus for automatic parallelization of external calls in scripting languages. Their calculus represents control flow constructs like loops and conditionals through Church encodings, allowing these constructs to be treated uniformly as functions whose evaluation can be parallelized. The language's opportunistic evaluation strategy identifies implicit parallelization opportunities that would typically require explicit annotations. The implementation demonstrates performance improvements across several

LLM applications, achieving up to 6.2x speedup compared to sequential execution. The authors show how their evaluation strategy naturally handles streaming data by allowing computation to proceed on partial results. My formalization work builds directly on their calculus, providing machine-checked proofs of the metatheoretical properties that establish correctness, particularly confluence and well-formedness.

Recent work has developed several domain-specific languages to address various aspects of LLM programming, though none achieve automatic parallelization. LangChain [2] enables users to construct pipelines of LLM operations and arbitrary functions but relies on explicit parallelization primitives. SGLang [3] provides advanced prompting capabilities and supports fork/join parallelism through explicit annotations. While its compiler mode can construct dataflow graphs, it lacks support for control flow operations, and its interpreter mode relies on Python's sequential evaluation. While these DSLs focus on different aspects of LLM programming than EPIC, their limitations in handling parallel execution help motivate the need for formal foundations in automatically parallelizing LLM-based scripts.

3. PROJECT DESIGN

The formalization of EPIC in the Rocq proof assistant presented several interesting challenges, from representing the language's syntax and semantics to proving key properties that establish the correctness of its opportunistic evaluation strategy. In this section, I describe the design of the formalization, focusing on its structure, core definitions, operational semantics and proof techniques.

3.1 Formalization Structure

The formalization is structured as a single Rocq development file that captures both the static and dynamic aspects of the EPIC language. Following a standard approach for formalizing programming languages, I organized the development into clearly delineated sections: syntax definitions, well-formedness conditions, operational semantics and proofs of key properties.

My first design decision was to represent the core language using mutually recursive inductive types for terms and lets-bindings, reflecting EPIC's distinction between function definitions and bindings. This mutual recursion required careful handling, especially when defining induction principles for proofs.

For variable binding, my team and I used de Bruijn indices, a numerical representation of variable references that simplifies substitution operations but adds complexity to context management. This approach required defining helper functions for shifting indices when moving terms across binding boundaries.

The development focuses on proving two key properties: confluence (different evaluation orders lead to equivalent results) and well-formedness preservation (evaluation maintains the language's scoping rules). These properties establish the soundness of EPIC's parallelization approach and guarantee that automatic parallelization preserves program semantics.

3.2 Language Definition

EPIC's syntax is represented using two mutually recursive inductive types: term for function definitions and lets for bindings and function applications. This mutual recursion captures the language's essential structure, where functions contain binding sequences and bindings can include function definitions.

The core constructs of EPIC include function definitions, binding a function to a name, function application, tuple construction and projection, variable substitution and returning a value.

External calls are represented as a special case of function application with primitive

values. This representation allows the formalization to treat external calls uniformly with other function applications at the syntactic level while distinguishing them during evaluation.

Well-formedness is defined using a context-based approach where a context represents the set of variables in scope. The judgment ensures variables are used only after definition and that scope is properly maintained across function boundaries. This enables efficient checking of variable scope and is essential for proving that evaluation preserves well-formedness.

3.3 Operational Semantics

The operational semantics of EPIC is defined through small-step reduction relations that specify how programs evaluate one step at a time. I structured these relations into three layers: the core semantics, task semantics, and opportunistic evaluation strategy.

The core semantics handles the deterministic fragment of the language, including function inlining, variable substitution and tuple manipulations. Each rule precisely defines how a specific language construct steps to its reduced form. For example, the rule for function application replaces the call site with the function body after appropriate variable substitution.

Task semantics extends the core semantics to handle external calls, which introduce nondeterminism since they interact with the outside world. External calls are represented as opaque "tasks" that can be resolved to arbitrary values according to an external semantics parameter. This approach allows the formalization to reason about external effects without modeling them explicitly.

The opportunistic evaluation strategy defines how the language exploits parallelism. Unlike traditional sequential evaluation strategies, opportunistic evaluation steps all statements in a program simultaneously, maximizing parallelism while preserving dependencies. This strategy is formalized as a relation that applies the core and task semantics to all reducible expressions in a program in a single evaluation step.

3.4 Proof Techniques

Proving properties about EPIC required specialized techniques due to its mutually recursive definitions and parallel evaluation strategy. A significant challenge was defining appropriate induction principles that work with mutual recursion between terms and bindings. I developed a custom mutual induction principle that allows for simultaneous reasoning about both syntactic categories.

For confluence proofs, I used a technique based on local confluence, showing that if a term can step to two different terms, these terms can eventually reach a common result. Since external calls introduce nondeterminism, I extended this technique to reason about sets of terms rather than individual terms, defining confluence in terms of possible outcomes from different evaluation paths.

4. ANTICIPATED RESULTS

The formalization of EPIC in Rocq is expected to provide a strong theoretical foundation for the language's claims of safe automatic parallelization. While time constraints prevented a complete proof of full confluence, the partial results achieved still contribute valuable insights into the language's properties.

The primary anticipated outcome is a formal verification of core properties that would guarantee the soundness of EPIC's parallelization approach. Specifically, the confluence property for the core deterministic semantics demonstrates that different evaluation orders of the same deterministic program fragment lead to equivalent results. This property is crucial as it establishes that automatic parallelization of deterministic code

preserves program meaning. For the extension to nondeterministic external calls, we anticipated showing that any nondeterminism in the final result would arise solely from the inherent nondeterminism of external calls, not from the parallelization strategy itself.

Another expected result is the formal verification of well-formedness preservation. This property ensures that if a program starts in a well-formed state, it remains well-formed throughout evaluation, regardless of which evaluation path is taken. The formalization proves that variable scoping is maintained correctly even when execution order is changed through parallelization. This result is particularly important for establishing the safety of the opportunistic evaluation strategy, which aggressively reorders operations to maximize parallelism. Based on similar formalizations in the literature, we expect that completing the remaining proof obligations would confirm that EPIC's approach to automatic parallelization is both correct and comprehensive, providing developers with a sound foundation for parallel scripting without requiring explicit parallelization primitives.

5. CONCLUSION

The formalization of EPIC in the Rocq proof assistant represents a significant step toward rigorous guarantees for automatically parallelized LLM-based scripts. By encoding EPIC's semantics and proving key properties like confluence and well-formedness preservation, this work provides mathematical assurance that EPIC's opportunistic evaluation strategy safely parallelizes programs without changing their meaning. The mutual induction principles developed to handle EPIC's recursive structure offer a robust framework for reasoning about languages with complex binding patterns and external effects.

This formalization addresses a critical need in modern software development, where LLM-based scripts face increasing performance demands but lack formal foundations for safe parallelization. By establishing that EPIC's parallelization approach preserves program semantics, developers can confidently leverage automatic parallelization without sacrificing correctness. The benefits extend beyond LLM applications to any domain where external calls create performance bottlenecks, offering a principled approach to exploiting implicit parallelism while maintaining program behavior.

6. FUTURE WORK

Several extensions to this formalization would enhance its applicability and theoretical foundations. First, incorporating primitive values like integers, strings, and booleans into the calculus would bring the formalization closer to practical implementations. This extension would require defining appropriate reduction rules for operations on these primitives and proving that they maintain the language's key properties. Adding a static type system represents another important direction. A properly designed type system could provide additional guarantees about program behavior and potentially enable more aggressive parallelization optimizations.

Extending the formalization to handle more complex control flow patterns, such as exceptions and continuations, would broaden its applicability to realistic programming scenarios. This would require careful consideration of how these features interact with parallelization and may necessitate refinements to the confluence property. Integrating effects typing or capabilities could provide additional fine-grained control over which operations may execute in parallel, allowing for more nuanced reasoning about when parallelization is safe and beneficial.

7. ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the research team at the University of Pennsylvania, particularly Steve Zdancewic, Stephen Mell, and Joey Velez-Ginorio, for

their guidance and collaboration throughout this project. This work was supported by the REPL (Research Experiences for Undergraduates in Programming Languages) summer program run by Joey Velez-Ginorio, which provided me with invaluable exposure to programming languages research.

REFERENCES

[1] Stephen Mell, Konstantinos Kallas, Steve Zdancewic, and Osbert Bastani. 2025. *Opportunistically Parallel Lambda Calculus. Or, Lambda: The Ultimate LLM Scripting Language*. Retrieved from https://arxiv.org/abs/2405.11361

[2] Harrison Chase. 2022. LangChain. https://github.com/langchain-ai/langchain

[3] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. *SGLang: Efficient Execution of Structured Language Model Programs*. Retrieved from https://arxiv.org/abs/2312.07104

INSTRUCTOR NOTE: More detailed instructions for each assignment and element of the Capstone report can be found in the *CS4991 Writing Guide,* posted in the Files section of Canvas.