Towards Correct Legion

George Stelle
Los Alamos National Laboratories
stelleg@lanl.gov

Noah Evans
Sandia National Laboratories
nevans@sandia.gov

Abstract

Legion is a programming model that uses logical regions of memory to reason about locality and concurrency. We formalize the Legion semantics in a mechanized proof assistant, Coq, and show how this enables one to reason formally and prove theorems about the semantics. We also discuss how to extend this work to prove that serial semantics are preserved and how given enough time, one could extend the effort to prove an implementation is correct.

1 Intro

High performance computing, as the name implies, has long focused on performance. Expensive machines are purchased to run increasingly complex simulations, and every optimization can save huge amounts of money, both due to the initial cost of the machine and the cost of power to run it.

That said, what good is a fast computation if we aren't certain that the values it computes are correct? Presumably, these computations inform important decisions in areas ranging from climate to nuclear safety. We should strive to be as confident as possible that we aren't basing these decisions on buggy code.

The strongest guarantees of correctness come from the area of formal specifications and proofs. A specification is a description of how a program should behave, while proofs are used to show that an implementation abides by a specification. *Formal* specifications and proofs are defined as specifications and proofs that have been embedded in a machine-checked formal logic [**? ?**]. This process of formalizing specifications and proofs in machine checked logics has been shown to catch common errors in previously informal proofs.

In addition, recent work has shown that modular, composable formal specifications enable formal correctness proofs for large software systems that were previously unattainable [**?**]. By integrating a formal specification tightly with an implementation, one can achieve the highest levels of confidence in code currently possible [**?**]. For example, the Compcert project implements an optimizing c compiler and proves it correct. The success of Compcert in removing compiler bugs has been well documented. We will discuss later how Compcert could be used in building verifiable HPC systems.

In HPC, programming models have become an increasingly important way to ensure that code stays maintanable, performant, and portable [**?**]. We claim that these programming models provide a good target for formalization: any correctness arguments can be leveraged by every program using the model. It is exactly this kind of modularization of proofs that has been so successful in proving programs correct [**?**].

For this paper, we take a recent programming model, Legion, and formalize its semantics and type system. We then prove a few supporting lemmas as examples of the kinds of things one can prove in an HPC settings. Finally, we discuss the some of the possible paths towards full, formally correct implementation. This is no small task, as we discuss later, and the ways forward are numerous and lengthy. But if we believe that HPC informs important decisions, we must strive for this pinnacle of software correctness!

2 Background

Legion is a programming model for high performance computing that enables reasoning about locality and with the attractive property that it is guaranteed to preserve serial semantics. It achieves this by dividing memory into logical regions. This allows programmers to avoid the burden of reasoning about the non-determinism that arises from concurrent computation. For this paper, we'll be basing our formalization work on the OOPSLA 2013 by Triechler et. al. While there has been much work extending Legion since then, the core semantics has stayed relatively stable.

Improvements to proof assistants over recent decades have made them a powerful tool for implementing and reasoning about code. From provably correct compilers to provably correct operating systems, proof assistants enable extremely high certainty in correctness even for large scale software projects.

Coq is a dependently typed proof assistant. At its core is the Calculus of Inductive Constructions, a small dependently typed programming language and logic, capable of powerful abstractions for formal proofs. In the last 10 years great strides have been made in using it to both build systems and prove useful properties about them [**? ? ?**].

Correctness'17, November 2017, Denver Colorado, USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

Specifications come in a wide range of formats. Many, like the OpenMP specification, are very informal, primarily consisting of english descriptions of how different constructs should behave. Others, like Legion's, are written by hand in the using standard notation for logical relations, etc. For this paper, when we refer to a *formal* specification, we are referring to a specification defined in a formal logic, like Coq.

While semantics defined by hand as logical relations are a huge improvement over natural language specifications, there is still much room for improvement. [**?**] has shown that even hand-written mathematical specifications and proofs in computer science papers almost always have errors that can be caught by a formal proof. This is not to say, of course, that formal specifications and proofs are infallible. On the contrary, it is quite easy to get a specification wrong. That said, often, if a there is a mistake in a spec , the process of proving it formally will expose that mistake, allowing for its correction.

3 A Formal Specification of Legion

"A program without a specification cannot be incorrect, it can only be surprising." [**?**] One of the most important and challenging aspects of proving correctness is defining a formal specification of how a program should behave. Indeed, there is currently a large NSF effort involving many of the best minds in programming languages dedicated to improving our ability to do just this [**?**]. The choice of specification has far-reaching consequences in how easy it is to later prove correctness properties.

One thing that makes Legion unique among high performance runtime systems is that it provides a specification of its operational semantics [**?**]. We take this specification of and formalize it in Coq, enabling us to prove lemmas and theorems about it. See Figure **??** for the Coq implementation of the syntax and semantics.

The Legion semantics are defined as a small step operational semantics, which result in non-deterministic orderings of memory reads and writes. By defining a notion of `valid_interleaves`, we can reason formally about what evaluations can happen concurrently. This should allow us to prove formally that if the runtime makes scheduling decisions based on region permissions in a particular way, serial semantics are preserved with respect to the resulting heap state and return value. This is a desirable property: it means that any code using Legion can reason, either formally or informally, about its code with respect to a simple serial semantics, and have those semantics be preserved for parallel execution. This is in contrast to programming models like Cilk, which claim a weaker property: a serial execution is one of many valid executions.

4 Discussion

One common theme in formalizing systems is the discovery of holes in the corresponding informal specification. For example, when formalizing the C language for the CompCert project, Leroy et. al. found many cases where the specification was incomplete. Note that this is seperate notion from undefined behavior. Undefined behavior in a specification is well defined. A little bit like known unknowns vs. unknown unknowns, gaps in a specification can make it impossible to prove desired properties of a system.

In formalizing Legion, we discovered that while there is an impressive theory ensuring that the side effects of function calls running concurrently preserve serial semantics, there is no theory for what function calls can run concurrently based on value dependencies. For example, consider the infamous Fibonacci example, which naively computes the n'th Fibonacci number:

```
fib(n) = if n < 2 then 1 else fib(n-1) + fib(n-2)
```

4.1 Performance

Returning to the "P" in "HPC", an important area of research is formal reasoning about performance. One of the challenges for Legion is that the scheduler has a non-trivial amount of work to do. Generally at least one core per node is dedicated entirely to runtime overheads. This is one area where Cilk, though fairly informally, shines. They prove a number of bounds on work that the scheduler has to do.

Ideally, we'd like to make similar claims about Legion runtime overheads. Currently, due to the complexity of the runtime implementation, this is infeasible. Like some of the desired correctness properties, a significant simplification of the runtime, and perhaps the semantics, would likely be required to achieve such a goal.

5 Conclusion

We hope to have convinced the reader of the importance and viability of proving software systems in HPC correct. While we've only taken small steps on the path to provably correct Legion, our hope is that it will inspire support for such ventures by providing a convincing story on how to move forward. There is ever- increasing evidence that this class of deep specifications and certified software is an incredibly powerful tool for proving desirable properties about our large software systems. We'd like to see more focus on correctness in HPC, and we think deep specifications and certified proofs are the way forward.

References