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, and for good reason. 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 relatively certain that the answers it returns are correct? Presumably, these computations inform important decisions about topics ranging from climate to nuclear safety. We should be as confident as possible that we aren't basing these decisions on unexpected semantics.

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

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.

Correctness'17, November 2017, Denver Colorado, USA

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

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 [**? ? ?**].

Specifications come in a wide range of formats. Many, like the OpenMP specification, are very informal, primarily consisting of english. Others, like Legion's previously, 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 in papers are a huge improvement over natural language specifications, there is still room for improvement. [**?**] has shown that even mathematical specifications and proofs in computer science papers almost always have errors that would 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." TODO: Find quote source in recent Appel paper. One of the most important and challenging aspects of proving correctness is defining a formal specification of what a program *should* do. Indeed, there is currently a large NSF effort dedicated to improving our ability to do just this [**?**].

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.

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

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)
```

References