Formal verification of Polyhedral Compilation

Siddharth Bhat
4th semester
20161105

May 25, 2018

Contents

1 Introuction

We wish to explore the current state-of-the-art when it comes to verified compilation, and then understand how this boundary can be pushed, with respect to polyhedral compilation techniques.

Currently, CompCert, Compcert-SSA, etc. are all extending how much of compiler technology has been formally verified. However, none of these are optimising compilers. They do perform several peephole-style optimisations, but they do not have the rich optimisations that conventional optimising compiler frameworks such as LLVM has.

We work with the polyhedral compilation framework, which is a collection of mathematical methods to represent and prove properties of loop transformations. This set of techniques has both unified many disparate loop transformations that used to exist (loop fusion, fission, tiling, strip-mining, etc). It has also given a nice mathematical framework to reason about loop transformations.

We have experience working with loop optimisations in Polly, which is a loop and data locality optimiser that is integrated within LLVM. It's been run on a large collection of benchmarks and real world code (notable, AOSP (the android open source project)). It is quite "industrial strength", in that it deals with the many pathological cases that show up in real-world programs while also providing high-performance transformations.

We would like to integrate this set of ideas that have been refined in the course of working on Polly into a verified compiler such as CompCert. We believe this has potential, because the theory is elegant and is amenable to being formalised, unlike the wide variety of ad-hoc peephole style loop optimisations that used to be present before it. It also provides impressive speedups on many benchmarks, which is always nice to have in a compiler.

It also forms an interesting research problem: how does one integrate the currently present coq tooling surrounding polyhedral libraries (such as VPL) and compilers (such as CompCert) to obtain provably correct loop optimisations.

Therefore, this independent study aims tp:

- Learn Coq, a tool for creating developments of programs and proofs which is widely used by the specification and formal verification community.
- Gain experience and tinker with CompCert, a verified C compiler developed by INRIA and Airbus International.
- Try to implement transformations within CompCert to understand its' model of compilation.
- Perform a similiar exercise with the VE-LLVM codebase, to understand their similarities and differences with respect to CompCert.

2 Learning Coq

With regard to learning Coq, Software Foundations is widely considered a standard book on the theory and practise of Coq. Therefore, I have solved the entireity of the first volume and much of the second volume. The solutions to the exercises are posted on my github account (Link here).

To learn CoInduction, I referred to Coq'Art, another slightly older book which takes a more formal and theory-based approach. My solutions of the coinductive chapter are also posted on github (Link here).

To learn the finer aspects of much of the theory, mostly revolving around dependent types and many of the problems they pose during proof development, I referred to CPDT (Certified programming with dependent types) (Link here).

Stack overflow was also invaluable, with members of the Coq community helping me with the more practical aspects of dealing with a theorem prover. Again, questions and answers are available against my profile (Link here).

To get a feel for the problem at hand, I have a development of the theory of (a naive view of) memory, loops, and arrays.

I first proved statement switching: That is, two statements which do not alias can be switched in the program order without affecting program semantics. That particular proof development is available on github (Link here).

After that, I attempted to scale this up to a proof about loops on my toy language. This too has an associated proof development (Link here). Note that with this proof development, I have left in the original comments. I have also added new comments, where the benifit of hindsight and greater knowledge allows me answer `TODO` and `WHY` that I had left for myself. There is clear and marked growth in my understanding of Coq, as witnessed by the new comments.

3 CompCert

3.1 Overview of CompCert

CompCert is a verified compiler for a large subset of the C language. It is used in production by AirBus Internatioal, and is developed by the INRIA group, at the Paris-Rocquencourt research center by the Gallium team. The PI of the project is Xavier Leroy, a computer scientist who is most well known for his development of the OCaml compiler, along with the CompCert project.

CompCert proves the semantic preservation of C semantics by constructing a bisimulation between the original C program's operational semantics and that of the target processor's instruction set semantics.

CompCert comes with a sophesticated memory model of the C language. It also has large body of theory implemented within it, to reason about semantics, simulation diagrams, lattices, finite maps and sets, and syntax-based transformations.

This makes using CompCert difficult, since weaving a new theory within the already existing theory is difficult. On the other hand, the pre-existing in-

frastructure makes many theorems slight extensions or adaptations of theorems that are already proven in CompCert.

3.2 Our transformations in CompCert

We chose to perform transformation on CMinor, a language within CompCert, which is roughly C-with-explicit-exits. We felt that this language was at the right level of abstraction for us to work at, due to the presence of loops and conditionals still being present at this level of the language. If we had chosen some lower-level language within CompCert, we would have had to construct techniques to analyze and recover loop information from CompCert.

Since we were interested in polyhedral compilation, we know that the target programs which we would compile would be SCoPs (Static control parts): That is, a single-entry-single-exit region of a program with natural loops and reducible control flow. Moreover, all conditions are not data-dependent, and all loop bounds and array accesses are affine functions of the induction variables.

Such a program setting can be described entirely by the polyhedral model, which we exploit to analyze and transform the program.

To model this precise subset of programs, we construct *our own IR (internal representation)*, called `PolyIR`.

We show a bijection of semantics between CMinor programs and PolyIR. We carry out our proofs of semantic preservation of loops within PolyIR, exploiting the bijection of semantics to CMinor to lift programs to PolyIR, carry out the transformation, and then lower the program back to CMinor.

3.3 Semantic preservation proofs in CompCert

The aim of exploring CompCert was to have proofs of semantic preservation of store switching and loop reversal.

We managed to perform proofs of *memory preservation* within compcert for our loop reversal and statement switching passes. This requires a lot of reasoning around CompCert's complicated memory model, which models the full behaviour of the C specification.

However, we did not prove full semantic preservation of either of these transformations. Indeed, in general, showing semantic preservation of passes which mess with instructions is difficult in CompCert. There is a PLDI'16 paper, "Peephole optimisations for CompCert", which required the development of a general framework of program transformations.

4 VE-LLVM

VE-LLVM is a fomalization of LLVM's semantics within Coq. They provide an interpreter for LLVM. More importantly, they formalize the semantics of LLVM, and hence one can implement transformations over the IR whose semantic preservation proofs can be supplied.

This is of interest to us, since we better understand LLVM semantics than that of C. Also, Polly was written for LLVM, so we also understand how to perform polyhedral compilation within LLVM. Hence, we feel that is also a good candidate for our proof development.

I have recently started exploring VE-LLVM. In this case, since their proof development is quite new, I had to build up a lot of infrastructure around VE-LLVM before I could start working on anything. Hence, the raw code is present, but a well-documented proof development is missing, both from the original developers and from me.

I can, however, describe what I have done with regards to VE-LLVM.

- Create a generalised infrastructure around passes, available as Pass.v
- Conjecture and prove lemmas around the main monad within VE-LLVM, which is used to model IO, available at Trace.v
- Implement and verify large parts of the proof of semantic preservation of a simple pass which rewrites `2 * x` into `x + x`. This is a more delicate proof than it first seems due LLVM's semantics and their interactions with integer wrapping.

In particular, VE-LLVM's main challenge is that the program's effects are recorded against a Coinductively defined, possible infinite `Trace` object. Coinductive reasoning is famously tricky in Coq, due a syntactic check that Coq performs on coinductive proofs. In general, Coinductive proofs compose poorly due to the presence of this syntactic.

This is complicated by the fact that VE-LLVM does not really reason about the traces themselves. Rather, it reasons over equivalence classes of traces, with the equivalence relation, termed `EquivUpToTau`, *also a Coinductive proposition.*

We currently have a decent understanding of the mechanics of proving within VE-LLVM. However, there seems to be much basic infrastructure that is lacking. Hence, we have e-mailed the principal investigator, Steve Zdancewic, with questions and possible collaboration efforts.

5 Conclusion

I believe that we have fulfilled the expected goals of the independent study: Namely, that of developing an understanding of theorem provers, formal verification, and verified compilation. We got a decent amount of headway into the polyhedral portion of this as well, which we will continue to work on over the summer.