Compiling Logic Programs

Yichen Ni (yichenn@andrew.cmu.edu)
Chase Norman (chasen@cmu.edu)

October 2024

Project Webpage: `chasenorman.com/compiler.html`

1 Project Description

Logic Programming is a programming paradigm used for knowledge representation and automated reasoning tasks [4]. Prolog is a well-known example of a Logic Programming language [3]. The performance of Logic Programs is very sensitive to small code transformations, yielding great opportunity for compilation. **We intend to build a compiler to compile logic programs to more efficient but semantically equivalent logic programs.**

Logic Programs have many features in common with imperative programs, allowing standard compiler passes to be adapted to this new paradigm. For instance, Logic Programs have basic blocks (called clauses), control flow edges (called rules), and expressions (called literals). The uniquely challenging aspect of the control flow of a Logic Program is that it exhibits non-determinism, so execution frequently *forks* into multiple parallel copies of the original, each selecting a different rule. As forks continue, this quickly leads to a combinatorial explosion that severely limits scalability. Reducing the branching factor of this execution fork tree can therefore provide super-exponential improvements in performance.

We believe the compilation passes we learned in class each have an analog for Logic Programs. The most impactful optimization will be Dead Code Elimination, as it will remove clauses and rules that can never be executed or can never lead to a relevant computation, leading to a sizable reduction in branching factor and work done at each program step. Our 75% goal is to complete this pass, including all the tooling required to apply Dataflow passes to this novel use case (Section 2.1).

The next compilation of interest is in optimizing the process used to determine which rules can be applied at a given program point. This is called *unification* and accounts for the majority of logic programming runtime. Computing specialized unification procedures for the rules of the input program can dramatically speed this up in common cases. Furthermore, a flow-sensitive account of unification can allow the logic programming engine to determine which rules can be traversed as a function of the control flow leading up to that point. This marks our 100% goal.

The final compilation pass we wish to attempt is Conditional Constant Propagation. If an instruction does not exhibit non-determinism, we can precompute its value and propagate this change. This specialization both improves runtime performance and would potentially allow the other compiler passes to more aggressively reduce the rules and clauses in the program. As we envision it, this pass would require significant structural modification to the program (adding parameters to functions, modifying the call sites, introducing auxiliary variables) and the ability to undo these modifications once an logic program finishes executing, to recover a solution for the original. For this reason, this is our 125% goal.

We will evaluate our compiler passes on a suite of logic programs for automated theorem proving applications. These programs represent mathematical statements, where the output of the program is a formal proof in Dependent Type Theory. We will measure the runtime of these programs with and without our optimizations, the compilation time, and the number of rules and clauses that are optimized or removed by our preprocessing. We will build our passes in the Canonical logic programming engine, discussed in Section 2.5.

For much of the history of logic programming, logic programs have been hand-authored. Given a problem, the author manually implements an *encoding* with good performance. For automated theorem proving

applications, logic programs come directly from interactive theorem provers like Lean, without regard for performant encoding. As for imperative programs, this added level of abstraction between the user and their program makes compilation essential.

2 Logistics

2.1 Plan of Attack and Schedule

We will proceed much as we proceeded through the course assignments. We will define the CFG from a given logic program, implement a dataflow framework for passes, and implement a number of passes that facilitate analogues to optimizations taught in the course.

Week	Yichen	Chase
10/21	Build CFG Data Structure	Implement filtering rules to CFG edges
10/28	Dataflow Pass System	Filtering type-incorrect CFG rules
11/4	Liveness Pass	Dead Code Elimination
11/11	Reachability Pass	"Compiled Unification" à la TWELF
11/18	Conditional Constant Propagation	Flow-sensitive CFG edge filtering
11/25	Benchmarking and begin report	Benchmarking and begin report

2.2 Milestone

The major milestone we expect to complete is Dead Code Elimination. This involves the implementation of the dataflow framework and CFG generation. We expect Dead Code Elimination to significantly reduce the branching factor of most logic programs, with a performance improvement at least 2x on average.

2.3 Literature Search

There is existing literature on compilation for logic programming. The first major advancement was the Warren Abstract Machine (WAM), which is an intermediate representation for executing Prolog logic programs with a small instruction set [1]. Improvements in optimizing this intermediate representation follow in subsequent works [10, 5, 12, 6, 11]. At a lower level, `wamcc` is a compiler for WAM instructions to C programs that execute the semantics [2].

However, none of these works directly modify the input program. The performance gains obtained from these papers is exclusively from better definitions and implementations of intermediate representation instructions, as well as the ability to compile certain subproblems to concrete predicates. Our work assumes that the logic programming engine is fixed and seeks to find transformations that change the underlying work that needs to be done. Furthermore, none of these works can be applied for the use case of automated theorem proving, which requires Dependent Type Theory.

To our knowledge, this has only been attempted by Prof. Frank Pfenning and his students on the TWELF logic programming system [8]. We take inspiration from TWELF's "Compiled Unifiers" technique [9] which, although unpublished, has been brought to our attention by Prof. Pfenning directly.

2.4 Resources Needed

No special software, hardware, or compute is required for this project.

2.5 Getting Started

Through writing this proposal, we have outlined the analogous optimizations to those taught in the course and have a basic understanding of how they must be implemented. We intend to build these into the Canonical logic programming engine [7]. Chase Norman built Canonical over the past year to efficiently execute logic programs for use cases in automated theorem proving for mathematics. This will provide the basis on which our optimizations can be implemented and benchmarked.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Aug. 1991. ISBN: 9780262255585. DOI: 10.7551/mitpress/7160.001.0001. URL: https://doi.org/10.7551/mitpress/7160.001.0001.

[2] Philippe Codognet and Daniel Diaz. "wamcc: Compiling Prolog to C". In: *Logic Programming: The 12th International Conference*. The MIT Press, June 1995. ISBN: 9780262291439. DOI: 10.7551/mitpress/4298.003.0037. eprint: https://direct.mit.edu/book/chapter-pdf/2304255/9780262291439_cbf.pdf. URL: https://doi.org/10.7551/mitpress/4298.003.0037.

[3] Alain Colmerauer and Philippe Roussel. "The birth of Prolog". In: *SIGPLAN Not.* 28.3 (Mar. 1993), 37–52. ISSN: 0362-1340. DOI: 10.1145/155360.155362. URL: https://doi.org/10.1145/155360.155362.

[4] Michael R. Genesereth and Matthew L. Ginsberg. "Logic programming". In: *Commun. ACM* 28.9 (Sept. 1985), 933–941. ISSN: 0001-0782. DOI: 10.1145/4284.4287. URL: https://doi.org/10.1145/4284.4287.

[5] C.S. Mellish. "Some global optimizations for a PROLOG compiler". In: *The Journal of Logic Programming* 2.1 (1985), pp. 43–66. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(85)90004-4. URL: https://www.sciencedirect.com/science/article/pii/0743106685900044.

[6] K. Muthukumar et al. "Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism". In: *The Journal of Logic Programming* 38.2 (1999), pp. 165–218. ISSN: 0743-1066. DOI: https://doi.org/10.1016/S0743-1066(98)10022-5. URL: https://www.sciencedirect.com/science/article/pii/S0743106698100225.

[7] Chase Norman. URL: https://chasenorman.com.

[8] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.

[9] Brigitte Pientka. URL: https://github.com/standardml/twelf/blob/main/src/opsem/README.

[10] Anderson Faustino da Silva and Vítor Santos Costa. "The Design and Implementation of the YAP Compiler: An Optimizing Compiler for Logic Programming Languages". In: *Logic Programming*. Ed. by Sandro Etalle and Mirosław Truszczyński. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 461–462. ISBN: 978-3-540-36636-2.

[11] Peter Van Roy and Alvin M. Despain. "High-Performance Logic Programming with the Aquarius Prolog Compiler". In: *Computer* 25.1 (Jan. 1992), 54–68. ISSN: 0018-9162. DOI: 10.1109/2.108055. URL: https://doi.org/10.1109/2.108055.

[12] Neng-Fa Zhou. "Global Optimizations in a Prolog Compiler for the Toam". In: *The Journal of Logic Programming* 15.4 (1993), pp. 275–294. ISSN: 0743-1066. DOI: https://doi.org/10.1016/S0743-1066(14)80001-0. URL: https://www.sciencedirect.com/science/article/pii/S0743106614800010.