Compiling Logic Programs

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

November 2024

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

1 Introduction

1.1 Problem

Logic Programming is a programming paradigm used for knowledge representation and automated reasoning tasks [6]. Prolog is a well-known example of a Logic Programming language [5]. Because Logic Programs declaratively represent the specification for their solution rather than imperatively implement a procedure to construct it, Logic Programs have a search tree semantics that can quickly become intractable. The performance of a Logic Program is therefore highly dependent on the "encoding" of a problem, as this determines the branching factor of the search tree.

Canonical [12] is a Logic Programming engine designed to prove theorems in mathematics. Given an arbitrary mathematical statement formalized as a dependent type theory logic program, Canonical can execute it to find a proof. Canonical is intended to be used my mathematicians doing formalization work [7] in the Lean [10] interactive theorem prover, which, like Coq [2] and Agda [3], has its foundations in dependent type theory. A user wishing to prove a lemma or fill in a step of a larger proof may invoke Canonical as a Lean tactic to exhaustively search for a solution.

Scaling up Logic Programming to handle fully general mathematical reasoning is a challenge. Even simple problems generate a massive search space. This problem is compounded when the encoding is not hand-authored but is instead directly translated from Lean's internal representation, as it is in the case of Canonical's Lean tactic. For these reasons, it is essential that we perform as much pre-processing on input programs as possible to ensure efficient search.

Imperative	Logic Program
Instruction	Goal
Basic Block	Premise
CFG Edge	Rule
Branch Condition	Unification
Program Stack	Context

Figure 1: Correspondence

```
e : E
i : (G → G) → F
h : (E, F) → D
g1 : D → B
g2 : C → B
f : B → A
-----------------
main_goal : A := ?
```

Figure 2: A Logic Program

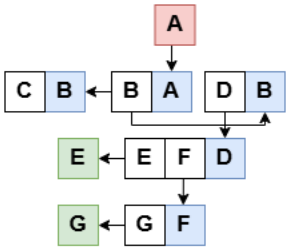

Figure 3: CFG

1.2 Approach

While imperative programs and logic programs appear quite different in their semantics, there is a surprisingly close correspondence that allows the methods from optimizing compilers to be applied analogously. Figure 1 shows this correspondence, and the terminology which we will use interchangeably. Following course concepts, we built a data structure for the control flow graph of a logic program, populated it with basic blocks and edges from the input program, and implemented a system for dataflow passes to be written on the CFG.

Using this dataflow system, we implemented passes for reachability and liveness, which proceed nearly identically to their imperative counterparts. With these passes, we completed Dead Code Elimination. We are then further able to refine the edges in the control flow graph using flow-sensitive analysis powered by Canonical itself. Using the analysis from earlier passes and bootstrapping with a small query to the Canonical theorem prover, we can ascertain that certain branches can never be taken.

With information gathered through the passes, we are able to implement several impactful optimizations at runtime. Compilation has accelerated a number of essential operations, like checking branch conditions and managing program state. These optimizations culminate in a path-sensitive analysis (again bootstrapped by Canonical) to determine pairs of edges that can never be executed in sequence.

An example of how dead code elimination can help optimize a CFG can be shown in the example shown below, where the problem is defined in Figure 4. Figure 5 shows the CFG representation of the same problem, where main goal is marked with red, final premises are marked green, and arrows represent the rules. After applying DCE, the CFG size is reduced significantly.

1.3 Related Work

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 pro-

```
cy1 : D → E
cy2 : E → D
p : D → A
f : ((a : A) → B) → A
g : C → B
c : C
-----------------
main_goal : A := ?
```

Figure 4: A Logic Program

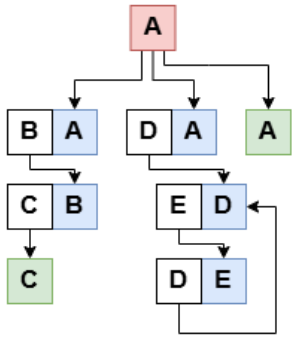

Figure 5: CFG (Original)

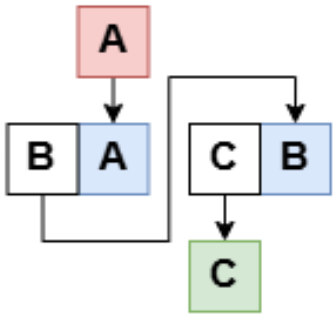

Figure 6: CFG (DCE Applied)

Figure 7: Comparison of Logic Program Representations and CFGs

grams with a small instruction set [1]. Improvements in optimizing this intermediate representation follow in subsequent works [15, 9, 17, 11, 16]. At a lower level, `wamcc` is a compiler for WAM instructions to C programs that execute the semantics [4].

Our work does not involve an intermediate representation of the logic program for efficient execution. Instead, the Canonical logic programming engine stays roughly the same. Our performance gains come from compilation passes that remove edges from the control flow graph, effectively changing the encoding of the input program and reducing the branching factor during search. Furthermore, many of the contributions of the WAM and others exclusively apply to constructs in Prolog, which is not expressive enough to perform general mathematics. All of our techniques apply to automated thoerem proving in depdendent type theory, which is an undecidable problem involving higher order unification [8] and heuristic-guided order of executing instructions.

To our knowledge, compilation for logic programs in dependent type theory has only been attempted by Prof. Frank Pfenning and his students on the TWELF logic programming system [13]. We take inspiration from TWELF's "Compiled Unifiers" technique [14] which, although unpublished, has been brought to our attention by Prof. Pfenning directly.

1.4 Contributions

Our work contributes to the field of logic programming by providing a comprehensive compiler infrastructure that adapts traditional compiler optimizations to this novel domain. The primary contributions of our project include:

- An adaptation of control flow graphs and dataflow system for use with logic programs.
- An implementation of Dead Code Elimination using liveness and reachability passes.

- A flow-sensitive pass which uses Canonical to eliminate inaccessible rules from the CFG.
- Several improvements to the performance of unification and metavariable domain computation, driven by analysis passes and path-sensitive analysis.

The optimizations and analyses we developed enhance the scalability and efficiency of logic programs, which is crucial for applications like automated theorem proving, where large, complex programs are automatically generated without regard for performance. By introducing automated compiler optimizations, we bridge the gap between abstract logic representations and practical, performant execution.

2 Background

2.1 Logic Programming

Logic programming is a declarative programming paradigm used extensively for knowledge representation, automated reasoning, and symbolic computation. Unlike imperative programming, where the programmer defines specific sequences of operations, logic programming focuses on defining relationships and rules that describe the problem.

In logic programming, computation is carried out by performing logical inference on a set of rules and facts. The core mechanism involves finding solutions to a given query by attempting to unify it with known rules and facts, using a process called unification. This non-deterministic search process, combined with backtracking, is what gives logic programming its flexibility, but it also presents challenges for efficient execution.

A logic program begins execution with the statement you are looking to prove. We call this the *main goal*. To achieve this goal, a number of premises are provided, some of which may apply. To apply a premise, you must supply the input parameters that the premise requires. These parameters are themselves goals that are filled recursively. Furthermore, these goals may be accompanied with premises of their own, which are added to the set of premises available to that goal in a process known as *context extension*. This pattern of goals with premises and premises with goals continues in a tree structure, defining the entire logic program, with the main goal as the root.

Logic programs have unique characteristics that differentiate them from imperative programs. These include the use of non-determinism, backtracking, and complex relationships between goals and premises. Due to these properties, the execution of logic programs often involves frequent branching, resulting in a combinatorial explosion of execution paths. This explosion in the number of possible paths that a program can take presents a significant barrier to scalability.

2.2 Dataflow Analysis for Logic Programming

Dataflow analysis is a method used in compiler theory to gather information about the possible set of values calculated at various points in a program. It forms the basis of many compiler optimizations. For our project, we extended dataflow analysis to logic programming by implementing reachability and liveness analyses on the CFG.

Using these analyses, we implemented Dead Code Elimination (DCE) to remove all unreachable or irrelevant components of the program. This not only reduces the branching factor in the CFG but also simplifies the logic program, improving performance by reducing the workload during runtime.

3 Implementation Details

3.1 CFG

The implementation of our compiler begins with the construction of a Control Flow Graph (CFG) to represent the logic program. Our CFG contains a list of all of the goals and premises in the program, along with their parent-child relationship as described in Section 2.

The edges for the control flow graph do not point from basic block to basic block. Instead, they point from instruction to basic block. Conceptually, this is because all instructions in a basic block are executed in parallel, and so are all the final instruction in their block and will execute a branch to another basic block. So, while a CFG in an imperative context can abstract instructions with basic blocks, the duality between goals and premises is fundamental in our CFG implementation.

Each goal in the CFG stores its parent premise, its child premises, and the edges connecting it to premises. Symmetrically, each premise stores its parent goal, its child goals, and the edges connecting other goals to it.

A key challenge in the creation of the CFG is populating the edges. In imperative code, this can be done directly through an analysis of jump and branch instructions to find their targets. For Logic Programming, determining whether a premise can apply to a goal is called the *unification problem* and it is undecidable in general. The issue is that the success or failure of a unification is dependent on the assignment to earlier goals (hence *dependent* type theory). We must be conservative and allow all edges that may possibly be taken with success. To address this, we implemented multiple layers of increasingly precise filters on the set of edges.

First, we attempt an exact match on the premise and goal type. These types are expressions in an abstract syntax tree, and so proceed by comparison of each of the nodes. If they disagree at any position

of the tree, we can be certain they will never unify. This happens most frequently. However, if each of the branches either succeeds or reaches a dependency on an earlier goal (called a *metavariable*) we must conservatively assume the constraint must be satisfiable. This algorithm is complete for simple type theory but misses a number of impossible edges where dependency arises.

Say, for instance, we have some premise which is a generic function, that for any input type T returns an element of type T. If our goal is to produce a proof of the proposition $x = y$, can we use this premise? If we assume that propositions are not data types, it is clear that T and $0 \leq 1$ will never match. However, since the value of T is a user-supplied goal, our naively assume the constraint is satisfiable.

To solve this issue, we observe that for any unification problem matching a and b, their types A and B must also unify. In type theory, each expression has a type, including types themselves. In the above example, the type of T is `Type` and the type of $0 \leq 1$ is `Prop`. Since `Type` $\neq$ `Prop`, we can exclude this edge.

Our second version of this algorithm begins much like the first. However, if the first algorithm generates a constraint, we recurse on the types of the two sides. Eventually, there will either be an exact match or a failure. This improvement comes with a significant reduction in rules. Further refinements on this algorithm will be discussed later, but they require analysis passes to be performed first.

3.2 Dataflow Framework

By design, our dataflow framework precisely matches the pseudocode given in lecture. The user provides functions that (1) specify the order that nodes are visited, (2) specify the predecessors/successors of a node, (3) transfer across a node, (4) meet two output values together, (5) create a value at the initial condition, and (6) create a value at the boundary condition. Running the pass then populates a map with the values at each of the nodes.

Our dataflow framework handles forwards and backwards passes through genericism. Simply changing function (2) to output either predecessors or successors changes the direction. More surprisingly, our dataflow framework is generic over the type of the nodes. This is due to our goal and premise duality. We found that some passes are best implemented on goals/instructions and others are best implemented on premises/basic blocks.

3.2.1 Reachability

A premise can only be applied to a goal if it is in the context of that goal. The main goal, for instance, only has access to its own premises, and not the local (child) premises of any other goal in the problem. We implement a forward pass called Reachability that for each goal determines which premises could be

accessible to it in any path.

Since we are looking for accessible premises in any path, we use union as our meet function. The transfer function over a goal simply adds the local (child) premises of that goal. The predecessors of a goal are all of the goals with a rule to its parent premise. Because this transfer function is distributive, we obtain a set of premises for each goal that is the union of all contexts that will be observed for that goal over the course of the algorithm.

Using this pass, we can remove rules in the following way: for each rule between a goal and a premise, if the premise cannot be in the context of that goal, we are safe to remove the rule. We can also remove a rule if it does not contain the premises of the main goal, as this means it is inaccessible from the main goal. Much like imperative unreachable code, this does not improve the runtime performance on its own, as such rules would never be attempted. Rather, it is useful for improving the precision and performance of future optimizations to be performed, especially those that iterate over rules or require the contexts of goals to be known.

While the above implementation proceeds much like reachability analysis in imperative programming, there is an improvement that applies only in this setting, which we refer to as Reachability+. When taking the predecessors of a node, we enforce not only that those predecessors have a rule to the parent premise of this goal, but that our parent premise is also reachable to that goal, so that this rule may be invoked. Essentially, we take reachability of rules into account during reachability.

This improves the pass significantly, but also introduces a dependence on the output of the dataflow pass in the successor computation function. This ostensibly breaks the termination guarantee for dataflow. However, since rules are strictly added as reachable over the course of the algorithm, and there are finitely many rules, the pass must continue to converge. Each pass from this point forward only operates on reachable edges.

3.2.2 Liveness

The Liveness pass is a backwards pass that determines for each premise which goals are live at that point. We consider a goal to be live if there is some live premise that matches with it, and we consider a premise to be live if all of its goals are live (such that it can be applied). The base case is a premise that has no goals, which can be applied unconditionally.

The successors of a premise are all of the premises that apply to each of the its goals, unioned together. The transfer function for a premise checks whether all the goals of the premise are live and if so adds all goals that unify with this premise to the set of live goals. The meet function is union and the boundary and initial sets are empty. This pass is quite similar in form to the Liveness pass from lecture.

Using the result from this pass, we can complete Dead Code Eliminaton. For any rule between a goal and a premise, if the goal is not live at that premise, remove the rule. This has the impact of deleting all rules involving dead premises or goals. Because we assume at the start of the algorithm that all goals are dead, we do not consider zombie code to be live.

3.3 Flow-Sensitive Analysis

As mentioned before, determining whether an edge in the CFG can be traversed is an undecidable problem, because it requires finding assignments for the dependent metavariable goals that satisfy the unification constraints. For a typical compiler, this would be the end of the story. However, since Canonical is a tool designed to exhaustively search for assignments to these metavariables, there is an opportunity to have Canonical improve compilation, not just the other way around.

Imagine we have a premise for reflexivity stating that for all x, $x = x$. Can we use this rule to prove $0 = 1$? Doing so generates two constraints on the metavariable x. One says that it is equal to 0 and the other says it is equal to 1. Clearly these constraints are inconsistent, but our earlier algorithms do not attempt any consistency checks on these equations.

For each rule surviving Dead Code Elimination, we attempt to validate with Canonical whether applying it causes inconsistency in constraints. To ensure that our application of this rule is sufficiently general to subsume all possible applications of the rule, we must ensure that the goal we apply it on has access to all of the premises in its context that it could possibly have access to. If we assume the goal has access to all the premises in the problem, our result will be imprecise.

Thankfully, we have already solved this problem with the earlier reachability pass, defining for each goal a set of premises which is the meet-over-paths for what will be accessible in any execution. In this sense, our pass will be sensitive to the flow of the program. Initializing a goal with this maximal context and applying the rule in question, we obtain constraints. We then continue executing Canonical on a low depth setting, such that the query will terminate quickly.

If Canonical reports that the constraints are unsatisfiable, we can safely remove the rule. If instead it is satisfiable we know the rule must stay. Often, Canonical will timeout, in which case we must conservatively assume the rule is consistent.

This technique can be used to solve the reflexivity example earlier, as well as other simple cases of inconsistent assignments to metavariables. In fact, this technique is also capable of noticing some metavariables that have no possible assignment even without equations, allowing rules to be eliminated in this case as well. Importantly, as Canonical improves in capability, especially in its capability to refute mathematical

statements, these improvements will also be seen in the compilation of logic programs.

3.4 Path-Sensitive analysis

All optimizations thus far have concerned the removal of rules unconditionally. It can be helpful for the performance of the logic program to find simple predicates that eliminate certain rules from consideration based on runtime information. For instance, we can determine that certain pairs of rules cannot be applied in sequence. At runtime, we can only consider rules that are available based on the previous rule that was applied.

We apply the same methodology as the flow-sensitive analysis. We create a goal with the maximal context, apply a rule, and then apply a second rule on one of the newly created goals. From here, we can run a short query to Canonical for whether the resulting state and constraints are satisfiable. There are frequently paths where a certain rule cannot be applied, and this allows us to pre-compute that information.

We limited ourselves to considering sequences of two rules because we believe this strikes a balance between performance and precision. In a graph with n vertices and a constant number of edges per node, this gives a runtime of $O(n^2)$. This is not out of line with the other passes we have implemented. At runtime, we use a table which indexes the available rules by the previous rule that was applied and the current goal. This allows us to avoid the paths that have been proven inaccessible during this step.

3.5 Accelerating Unification

All of the above refinements on the CFG are not useful if they do not drive improvements in performance. A sizable majority of the runtime of Canonical is spent testing candidate assignments for metavariables to see if the unification constraints are violated. Using information from the CFG and dataflow passes, we can drastically reduce the number of tests that need to be performed.

Previously, to find the set of candidate assignments to a metavariable, we would iterate through the context of the metavariable and attempt unification, filtering those that fail. As discussed previously, the majority of these fail immediately. After performing the passes on the CFG, we now pre-compute the available rules for each goal and, in the case of path-sensitive analysis, index these rules by the previously applied rule. At runtime, we iterate over these rules rather than the entire context of the metavariable and consider only the premises in the context that correspond with these rules. Essentially, we have pre-computed a filter on the set of options such that we only test premises that will succeed in at least some circumstances.

Our reachability, liveness, flow-sensitive, and path-sensitive analysis do not merely remove rules that immediately fail unification. Each of these passes also remove rules that provably cannot result in a successful

execution, either because the premise is unreachable, dead, or because the rule generates an inconsistent state as determined exhaustively by Canonical. For this reason, the reduction in rules not only improves the rate at which rules are performed, but also improves the quality of rules that are performed, potentially leading to an asymptotic improvement in performance.

When initially populating the CFG, we store an additional boolean for each of the rules. If the unification does not generate any constraints on metavariables (i.e. the two sides exactly match) we mark it as *always succeeds*. When testing an assignment to a metavariable using this rule, we can skip the unificationt test and generation of unification constraints, since we know the unification will be successful without constraints. This exchanges a recursive comparison of syntax trees, with β-reduction at each node, to a single boolean check where applicable.

Another clear inefficiency in this computation is in how we manage unification constraints on the metavariable. For each candidate assignment, we check to see if the constraint is violated and if so we remove it. It would be more efficient to analyze the constraints up front, and analytically determine which assignments will satisfy them. This is the idea behind what we call *masking*. For each constraint of the form $\texttt{?X} \equiv f(\cdots)$, we do not consider assignments to the metavariable `?X` that are an application to a variable other than f. In general, there are a number of cases that cannot be handled by this masking, but it suffices to remove many candidate assignments without expensive checks.

The final performance improvement we made to Canonical was in the heuristics of which goal/metavariable to work on next. Without loss of generality, it is always advisable to work on a metavariable that has only 0 or 1 options for assignment. In the 0 case, we are immediately able to cut the branch and backtrack and in the 1 case, we simply gain more information without forking the tree. Determining the size of the domain of a metavariable is the most expensive operation performed by Canonical, and cannot be performed on every metavariable at every step. Using what we determined at compile time, however, we can find an upper-bound on this number. We can simply consider the set of premises in the context of the metavariable corresponding to rules present in our CFG. Since we have precomputed information about this set, we can determine the size in roughly constant time. Using this heuristic we can identify metavariables that are good candidates to work on next.

4 Experimental Setup

We benchmark Canonical on 20 evaluation problems written in the Lean interactive theorem prover, which are converted to logic programs in Canonical's internal representation by a custom conversion tool. These problems were hand-authored by Prof. Jeremy Avigad and ourselves to represent a set of type-theoretically

interesting and algorithmically challenging problems for Canonical. The problems involve reasoning about inductively defined datatypes like lists and natural numbers, proving properties by induction, with equality reasoning, and using provided lemmas. For example, one such problem is to prove that the `append` operation on lists is associative. These problems frequently take millions of steps for Canonical to find a solution. It is worth noting that despite the challenge of these benchmarks, logic programs tend to have many fewer basic blocks than imperative programs. Our examples tend to have fewer than 40. For this reason, we disregard compilation time as a small fixed cost.

Each of our benchmarks were performed on a 2020 Mac Mini with M1 and 16GB of RAM. Compiler benchmarks are deterministic and so are each performed once. Efficiency improvements are measured in steps taken by Canonical per second, and are performed in a sequential mode which uses a single thread. We measured total step count using parallelism, but averaged over 5 runs to decrease the variance.

5 Experimental Evaluation

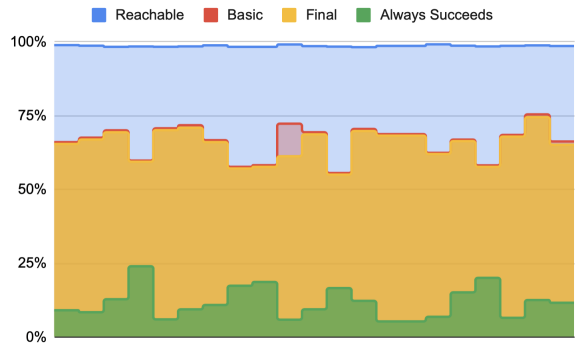

Figure 8: Number of Rules Optimized

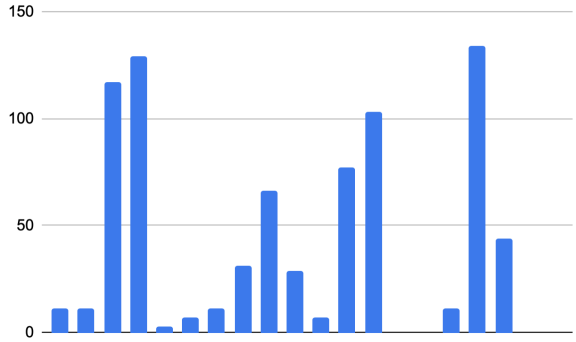

Figure 9: Path-Sensitive Optimizations by Example

5.1 CFG optimization

Figure 8 shows the breakdown of CFG rules before and after our optimizations on each of our 20 test problems. We count rules by percentage of the total possible number of edges in the CFG–that is the number of goals multiplied by the number of premises. While this may sound extremely pessimistic, our reachability analysis (in blue) found that over 98% of these potential rules are reachable and therefore were attempted by Canonical before our mitigations.

Next, we look at the result of the basic "exact match" algorithm for populating the CFG in red. This results in an average reduction by 33% of edges in the graph, which accounts for the vast majority of edges removed by our procedure. This is further refined to the final output of our algorithm, in yellow, which

consists of about 65% of all available rules. We have found that reachability and liveness, although they did contribute to further passes, did not have a significant impact on our test examples. This is because the script used to generate the logic programs from Lean uses a crawling procedure that defines only the notions that are required to solve the problem and only terminates when this has reached a fixedpoint. While this is slated to change in the future, this behavior essentially guarantees reachability and liveness of the resulting definitions, rendering these passes ineffective. The design of the translation from Lean to logic programming also meant there was a relatively limited number of distinct types mentioned in the problem, meaning there was less opportunity for our basic algorithm to differentiate.

One example stands out, `mul_even`, with a 15% reduction in rules between the basic algorithm and our final approach. This is not due to reachability and liveness for the reason above, but instead due to the flow-sensitive analysis. There are a number of goals (corresponding to typeclasses in Lean) which only have a single option for their assignment, significantly constraining the potential assignments to other dependent goals. Using Canonical as an oracle, the compiler is able to eliminate rules that violate the available typeclasses, for instance forbidding arithmetic addition on two types that are not numeric. As we believe typeclasses will become an important challenge for Canonical in terms of solving, we find this result promising.

Of these rules in our CFG, we found an average of 18% of them *always succeed* (i.e. they produce no unification constraints). These are shown in green and represent a subset of rules that do not trigger a unification in Canonical.

Figure 9 shows the number of path-sensitive optimizations that were found in each example. These are pairs of rules that are inconsistent to be applied in sequence. On average, there were 39 such optimizations.

5.2 Performance Optimization

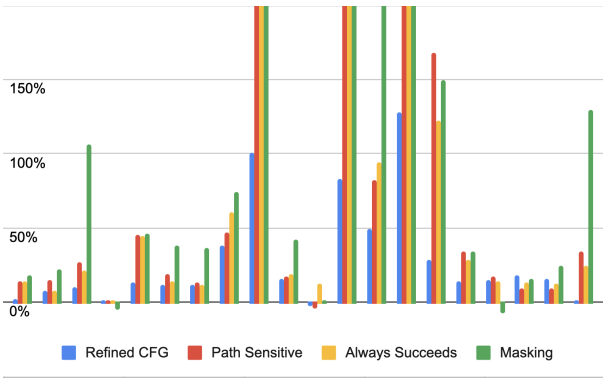

Figure 10: Efficiency Improvement

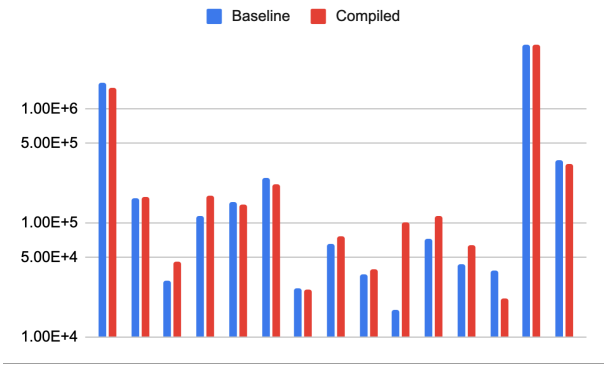

Figure 11: Step-count (log) applying compilation

Figure 10 shows the percentage improvement in runtime efficiency as we successively apply our opti-

mizations. We calculate efficiency as the number of steps (assignments to metavariables) completed by the Canonical per second until completion or timeout.

The first performance improvement comes from the reduction of rules in the CFG, in blue. We observe an average improvement of 27% in efficiency from no longer attempting these inconsistent edges. Adding our path-sensitive analysis (in red), we achieve an 80% improvement over the baseline on average. In some problems, especially those with short runtimes, we observe very significant performance improvements from path-sensitive analysis. This is because the short queries performed to Canonical are able to solve or nearly solve the problem in question. As a result, the path-sensitive information we generate is likely close to the absolute optimal, where only directions that can lead to a solution are considered. While we cut the graph off at 200%, these problems can saw larger performance gains, topping out at almost 400%.

Adding *always succeeds* to shortcut unification does not appear to reliably show improvement in performance. Our evaluation even shows a 1% degradation in performance, likely within the margin of error. We believe this is because most unifications that do not generate unification constraints are simple, like `Nat` $\equiv$ `Nat`, which can be checked nearly as fast as checking a boolean.

Finally, we add equation masking in green, resulting in a cumulative average performance improvement of 97% over the baseline. Overall, this nearly matches our expected performance improvement from our project proposal at 2x.

Figure 11 shows the total number of steps Canonical took during solving before and after compilation for each example, on a log scale. We exclude examples that are solved during compilation and that timeout. Each of these figures is the average over 5 separate runs, as non-deterministic heuristic choices can affect the steps that are taken. From this graph, we can observe that there is no significant positive impact of compilation to the asymptotic performance of Canonical, despite the potential for this as mentioned earlier. We suspect this is again due to the lack of dead code in our examples. In general, any asymptotic benefits in performance are not expected of an optimizing compiler, but are a bonus if they are possible.

6 Surprises and Lessons Learned

When we started this project, the connection between compilation for imperative programs and logic programs was promising, but it was not until further in development that it become concrete and the depth of the correspondence started to show. It was surprising how many aspects that had been considered purely runtime considerations (unification, metavariable domains, heuristics) had so much that could be done ahead of time. In fact, a number of operations that were once done at compile time are now done at runtime, including a number of essential capabilities for working with inductive types. For this reason, the compiler is

now a required part of the Canonical toolchain, performing essential operations to convert the input problem into a form for solving. Just as imperative compilation has become an entire field of study, it is clear that we could not do justice to all of the possible directions with this course project.

Another surprise is perhaps the diminishing returns of attempts to filter out rules. A sizable majority of inconsistent rules can be found by naive methods, and there are many unification failures at runtime that have not been converted to a compile-time check. The application of course techniques on logic programming has cast the subject in a new light, and has generated new ideas about automated theorem proving.

7 Conclusions and Future Work

There is a great amount of potential for future work on this project, which is planned for eventual inclusion in Canonical. More advanced passes from optimizing compilers, like conditional constant propagation and loop invariant code motion, have analogues in the logic programming setting as well. However, as discussed in our project proposal, these optimizations would require an additional component known as *proof reconstruction*. The output of a logic program amounts to an execution trace of the successful branch of the search tree, which would not be preserved under these transformations. The ability to perform these transformations with the necessary information to recover the solution to the original problem is a topic for future work.

There were a number of other potential optimizations that we considered during the development of the compiler. We could identify cases of so-called *pattern unification* which have a unique, efficiently computable solution, allowing for a shortcut to be taken during runtime. The compiler could also be further involved in analysis used to define the heuristics for which instructions should be executed in which order, and which branch of the search tree should be explored further. In particular, analyzing the dependencies between instructions could be within the purview of the compiler. If the estimates for the size of the domain of a metavariable produced by our compiler are strong enough, they may even obviate the need to compute domains entirely, cutting the runtime at least in half. Finally, considering extensions to the concepts from the Warren Abstract Machine that do extend to the dependently typed setting could be another avenue for future research.

8 Distribution of Total Credit

Chase Norman will take majority of the credit, he implemented the CFG representation and the optimization passes. Yichen Ni worked on the evaluation and documentation of the project. The final distribution of work is 60%-40%.

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] Yves Bertot. "A Short Presentation of Coq". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 12–16. ISBN: 978-3-540-71067-7.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. ISBN: 978-3-642-03359-9.

[4] 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.

[5] 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.

[6] 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.

[7] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. North-Holland, 2014, pp. 135–214. DOI: https://doi.org/10.1016/B978-0-444-51624-4.50004-6. URL: https://www.sciencedirect.com/science/article/pii/B9780444516244500046.

[8] Gérard P. Huet. "Higher Order Unification 30 Years Later". In: *International Conference on Theorem Proving in Higher Order Logics*. 2002. URL: https://api.semanticscholar.org/CorpusID:20550274.

[9] 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.

[10] Leonardo Mendona de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26. URL: https://doi.org/10.1007/978-3-319-21401-6_26.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.