Towards Efficient Search: Leveraging Relational Interpreters and Partial Deduction Techniques

Ekaterina Verbitskaia[1,2][0000−0002−6828−3698]

[1] Constructor University, Bremen, Germany
[2] JetBrains Research, Amsterdam, the Netherlands
kajigor@gmail.com

1 Introduction

Verifying a solution to a problem is much easier than finding one — this common wisdom is known to anyone who has ever had the opportunity to both teach and learn [?]. Consider the Tower of Hanoi, a well-known mathematical puzzle. In it, you have three rods and a sequence of disks of various diameters stacked on one rod so that no disk lies on top of a smaller one, forming a pyramid. The task is then to move all disks on a different rod in such a way that:

- Only one disk can be moved at a time.
- A move consists of taking a topmost disk from one stack and placing it on top of an empty rod or a different stack.
- No bigger disk can be placed on top of a smaller disk.

It is trivial to verify that a sequence of moves is legal, namely, that it does not break the pyramid invariant. Searching for such a sequence is more convoluted, and writing a solver for this problem necessitates understanding of recursion and mathematical induction. The same parallels can be drawn between other related tasks: interpretation of a program is less involved than program synthesis; type checking is much simpler than type inhabitation problem. And in these cases, the first problem can be viewed as a case of verification, while the other is search. Luckily, there is a not-so-obvious duality between the two tasks. The process of finding a solution can be seen as an inversion of verification.

There are many ways one can invert a program [?,?,?]. One of them achieves the goal by using logic programming. In this paradigm, each program is a specification based on formal logic. The central point of the approach is that one specification can solve multiple problems by running appropriate queries, which is also known by running a program in different *directions* or modes.

For example, a program `append xs ys zs` relates two lists `xs` and `ys` with their concatenation `zs`. We can supply the program with two concrete lists and run the program in the forward direction to find the result of concatenation: `run q (append [1,2] [3] q)`, which results in a list `q = [1,2,3]`. Moreover, we can run the program backwards by giving it only the value of the last argument: `run p, q (append p q [1,2])`. In this direction, the program searches for every pair of lists that can be concatenated to evaluates to three possible answers: `{<p = [], q = [1, 2]>; <p = [1], q = [2]>; <p = [1,2], q = []>}`.

Now, consider a verifier written in a logic programming language for the Tower of Hanoi puzzle `verify moves isLegal`. Given a specific sequence of moves, it will compute `isLegal = True` or `isLegal = False` based on whether the sequence is admissible. However, if we execute the same verifier backwards, say `run q (verify q True)`, then it will find all possible legal sequences of moves, thus serving as a solver. One neat feature is that one can generate a logic verifier from its functional implementation by relational conversion [?], or unnesting. Thus, one can implement a simple, often trivial, program that checks that a candidate is indeed a solution and then get a solver almost for free.

This verifier-to-solver approach is widely known in the pure logic (also called relational) programming community gathered around the KANREN language family [?,?]. These are light-weight, easily extendible, embedded languages aimed to bring the power of logic programming into general purpose languages. They also implement the complete search strategy that is capable of finding every answer to a query, given enough time [?]. The last feature is what makes the programming style in KANREN pure and distinguishes it from the one in PROLOG and other well-known logic

languages. No cuts or non-relational constructions are needed to make sure a relational program finds all answers, and for that reason, every program can be safely in any direction.

The caveat of the framework is its often poor performance when done in the naive way. Firstly, execution time of a relational program highly depends on its direction. The verifiers created by unnesting inherently work fast only in the forward direction, not when they are run as solvers. Secondly, there are associated costs of relational programming itself: from expensive unifications to the scheduling complexity [**?**]. Lastly, when a program is run as a solver, we often know some of its arguments. For example, the solver for the Tower of Hanoi will always be executed with the argument `isLegal = True`.

A family of techniques called *partial evaluation* are capable of mitigating some of the listed sources of inefficiency [**?**,**?**]. In this research, we have adapted several well-known partial evaluation algorithms for logic programming to work with miniKanren — a minimal core relational language. We have also developed a novel partial evaluation method called Conservative Partial Deduction.

The goal of the research is to figure out what combination of partial evaluation techniques is capable of making the verifier-to-solver approach a reality.

2 Specialization Efforts for `miniKanren`

Specialization, or partial evaluation, is an optimization technique that precomputes parts of program execution based on information known about a program before execution. For example, consider a function `exp n x = if n == 0 then 1 else x * (exp (n - 1) x)` and imagine that we know from some context that it is always being called with the argument `n` equal to `4`. In this case, we can partially evaluate the function to `exp_4 x = x * x * x * x * 1` that is more efficient than the original function called with `n = 4`. Note, that a smart enough specializer can also be able to generate a function of form `exp_4 x = let sqr = x * x in sqr * sqr` that makes even less multiplications.

This pattern can be expressed in a way that if there is a function with some of its arguments statically known `f` $\texttt{x}_{static}$ $\texttt{y}_{dynamic}$, it can be transformed into a more efficient function $\texttt{f\_x}_{static}$ with its parts dependent on the static arguments precomputed. The resulting program must be equivalent to the original one, meaning that given the same dynamic arguments, it will return the same results: `f` $\texttt{x}_{static}$ $\texttt{y}_{dynamic}$ `==` $\texttt{f\_x}_{static}$ $\texttt{y}_{dynamic}$.

In the field of logic programming, specialization is generally known as partial deduction. Besides the values of static arguments, a partial deducer can also consider the information about a direction of a program or the interaction between logic variables in a conjunction of calls. In addition to specialization, a relation with a given direction can be converted into a function in which expensive logic operations are replaced with streamlined functional counterparts. In this project, we combine the two methods for the verifier-to-solver approach.

2.1 Conservative Partial Deduction

Conjunctive Partial Deduction was first developed by Michael Leuschel for `Prolog` in the ECCE system [**?**]. It makes use of the interaction between conjuncts for specialization, getting rid of some repeating traversals of data structures as a result. We implemented this algorithm as a proof-of-concept for `miniKanren`, and found out that some of the specialization results were subpar. In some cases, the specialized programs had worse performance than the original ones.

Then, we formulated a different specialization method, which we called Conservative Partial Deduction. The difference with CPD lies in the way conjunctions are treated. They are split more often and thus generate smaller programs. The method was able to achieve a 1.5-2 times performance increase on a propositional evaluator program and almost a 40 times performance increase on a type checker.

2.2 Functional Conversion

Even after the partial deduction has finished, there are some sources of inefficiency that can be addressed. The base of relational programming is unification, which is an expensive operation, especially if it runs occurs-check. When a direction of a relation is known, mode analysis [**?**,**?**] can

be run to determine data flow from the statically known variables. As a result, most unifications can be converted into assignments, pattern matching or equality checks. Besides that, we can reorder the calls within a conjunction to restrict the search space early.

In some cases, when the direction is deterministic, the relation can be transformed into a pure function with no overhead of the relational language. However, if the direction results in multiple possible answers, we still need to express non-determinism. This is possible in a functional programming language by using the `Stream` monad that is at the base of the relational programming. We implemented a functional conversion method that gave the propositional evaluation program a 2.5-fold increase in performance. For some programs dealing with arithmetic, it improved performance by up to two orders of magnitude.

3 Published Papers

- Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. *"Relational interpreters for search problems."* Relational Programming Workshop. 2019. The full text is available here.
  - This paper formalizes the verifier-to-solver approach and highlights its pitfalls.
- Ekaterina Verbitskaia, Daniil Berezun, Dmitry Boulytchev. *"An Empirical Study of Partial Deduction for miniKanren."* Verification and Program Transformation Workshop. 2021. The full text is available here.
  - This paper describes a novel partial evaluation approach developed for miniKanren.
- Ekaterina Verbitskaia, Igor Engel, Daniil Berezun. *"A Case Study in Functional Conversion and Mode Inference in miniKanren."* Partial Evaluation and Program Manipulation Workshop. 2024. The full text is available here.
  - This paper describes functional conversion that makes it possible to generate a functional implementation of a solver from the functional implementation of a verifier, thus getting rid of the inefficiencies innate to the relational programming itself.

References

1. Sergei Abramov and Robert Glück. Combining semantics with non-standard interpreter hierarchies. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, pages 201–213, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
2. Sergei Abramov and Robert Glück. *Principles of Inverse Computation and the Universal Resolving Algorithm*, pages 269–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
3. Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan, et al. Foundations of reversible computation. *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405 12*, pages 1–40, 2020.
4. William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–26, 2017.
5. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2-3):231–277, 1999.
6. Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.
7. Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 192—-203, New York, NY, USA, 2005. Association for Computing Machinery.
8. Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. Relational interpreters for search problems. In *miniKanren and Relational Programming Workshop*, page 43, 2019.
9. Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. Typed relational conversion. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 39–58, Cham, 2018. Springer International Publishing.
10. Dmitry Rozplokhas and Dmitry Boulytchev. Scheduling complexity of interleaving search. In *International Symposium on Functional and Logic Programming*, pages 152–170. Springer, 2022.

11. Jan-Georg Smaus, Patricia M Hill, and Andy King. Mode analysis domains for typed logic programs. In *Logic-Based Program Synthesis and Transformation: 9th International Workshop, LOPSTR'99, Venice, Italy, September 22-24, 1999 Selected Papers 9*, pages 82–101. Springer, 2000.
12. Zoltan Somogyi. A system of precise models for logic programs. In *ICLP*, pages 769–787. Citeseer, 1987.
13. Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. An empirical study of partial deduction for minikanren. *arXiv preprint arXiv:2109.02814*, 2021.