Towards Efficient Search: Leveraging Relational Interpreters and Partial Deduction Techniques

by

Ekaterina Verbitskaia

PhD Thesis

Dissertation committee:

Submission: May 21, 2024

Prof. Dr. Anton Podkopaev
Prof. Dr. Jürgen Schönwälder
Dr. William E. Byrd

Constructor University | School of Computer Science and Engineering

Abstract

There is a duality between the problems of verification and search. It becomes evident in the context of relational, or pure logic, programming. Since any program in this paradigm can be executed in different modes, one interpreter can serve as both a verifier and a solver. One disadvantage of this method is its often poor performance. Several specialization techniques can be employed to mitigate the issue based on a mode and partially known arguments, i.e. the information known prior to execution. The goal of this work is to leverage partial deduction techniques to improve the performance of relational interpreters within the verifier-to-solver approach.

Acknowledgements

Contents

Chapter 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 [Lozov et al.(2019)]. 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 [Abramov and Glück(2000), Abramov and Glück(2002), Aman et al.(2020)]. 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 is 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 `[1,2]`, and it 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 [Lozov et al.(2018)], 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 [Friedman et al.(2005), Byrd et al.(2017)]. 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 [Kiselyov et al.(2005)]. The last feature distinguishes KANREN from PROLOG and other well-known logic languages, which have not been designed with search completeness in mind. In addition to this, KANREN discourages the use of cuts and non-relational constructions such as `copy-term` that are prevalent in other logic languages, and for that reason, every program writen in pure KANREN can be safely run 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 [Rozplokhas and Boulytchev(2022)]. 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 optimization techniques called *specialization*, or *partial evaluation*, are capable of mitigating some of the listed sources of inefficiency [De Schreye et al.(1999), Verbitskaia et al.(2021)]. Specialization 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 $\texttt{f}\ \texttt{x}_{static}\ \texttt{y}_{dynamic}$, 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: $\texttt{f}\ \texttt{x}_{static}\ \texttt{y}_{dynamic}\ \texttt{==}\ \texttt{f\_x}_{static}\ \texttt{y}_{dynamic}$.

In the field of logic programming, specialization is generally known as partial deduction [Komorowski(1992)]. 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 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 [Verbitskaia et al.(2021)]. Then we combined it with the functional conversion in an effort to get even greater performance increase [Verbitskaia et al.(2024)].

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

Chapter 2

Background

Contents

2.1 Logic Programming Languages

Over the years, multiple logic programming languages have been developed, with Prolog [Battani and Meloni(1973)] being the most widespread. It was the first successful attempt to enable declarative programming by means of writing programs in a subset of formal logic. At its core, Prolog uses Horn clauses, a semidecidable subset of first-order predicate logic. Each program formulates a set of facts and predicates that connect these facts. The evaluation of a program is done by an Selective Linear Definite clause resolution [Robinson(1965)] (SLD resolution) of a query, often following depth-first approach.

For years, logic programming was highly limited by hardware capabilities, leading to necessary compromises. One of them was an early removal of occurs-check from the unification algorithm [Cohen(1988)]. This means that running a query "`?  f(X, a(X)).`", given a program "`f(X, X).`", produces a nonsensical result "`X` $\mapsto$ `a(X)`". It is up to the user to ensure that a variable never occurs in a term it is unified with. Fortunatelly, a special sound unification predicate such as `unify_with_occurs_check` can be used to prevent such results.

Another compromise is linked to the implementation details and has more significant consequences. Logic languages are inherently nondeterministic, and evaluation on a deterministic computer requires decisions about how to explore the search space. Prolog was first designed for automatic theorem proving, an area in which a single solution to a query is generally sufficient. Thus, most Prolog implementations feature depth-first search, which often results in either non-termination or the generation of infinitely many similar answers to a query when an infinite branch of the search tree is explored. Additionally, non-relational constructs such as a cut and `copy-term` have been adopted for efficiency reason. Unfortunately, these two aspects often limit a relation to a single mode and directly contradict the main idea of declarative programming: a program can no longer be written with disregard of the peculiarities of the language.

Recently, there has been a resurgence of the logic programming paradigm with the emergence of new languages, including MERCURY[1], CURRY[2], MINIKANREN[3], and others. Additionally, a prominent video games developer Epic Games invested into designing a new functional-logic programming language [Augustsson et al.(2023)]. This new generation of logic languages combines the paradigms of logic and more mainstream fuctional programming. MERCURY and CURRY are standalone logic-functional programming languages with dedicated compilers that makes it difficult to interoperate with bigger systems typically written in a general-purpose language.

In contrast, MINIKANREN is implemented as a lightweight embedded domain-specific language, enabling the power of logic programming in any general purpose language. MINIKANREN features interleaving search [Kiselyov et al.(2005)] that guarantees that every solution to a query will be found, given enough time. Moreover, its extendible architecture allows for easy experimentation and addition of new features. The main design philosophy of MINIKANREN is to adhere to the pure logic programming as much as possible, so any program can be called in any direction. Taking all these considerations into account, we chose MINIKANREN as the main language for this research.

2.2 Specialization

The first specialization method, called supercompilation, was introduced by Turchin in 1986 [Turchin(1986)]. It was designed for the Refal programming language [Turchin(1989)], which was significantly different from the mainstream languages of the time. Since then, supercompilation has been adapted for various languages, expanding its utility across various programming paradigms [Klyuchnikov(2009), Mitchell(2010)]. Numerous modifications have also emerged, featuring alternative termination strategies, generalization, and splitting techniques [Leuschel(2002), Sørensen and Glück(1995), Turchin(1988)].

Several optimizations rely on the information about program arguments known statically. These optimizations precompute the parts of the program that depend on the known arguments and produce a more efficient residual program. Such transformations are generally known as mixed computations, specialization, or partial evaluation. It was first introduced by Ershov [Ershov(1982)] and was mostly aimed at imperative languages. A lot of effort has been extended to partial evaluation [Jones et al.(1993), Jones(1996)] since its first appearance, including the development of self-applicable partial evaluators.

In logic programming, a general framework called rules + strategies, or fold/unfold transformations, was introduced by Pettorossi and Proietti [Pettorossi and Proietti(1996), Pettorossi and Proietti(1994)]. It serves as a foundational theory for many semantics-preserving transformations, including tupling, specialization, compiling control, and partial deduction. Unfortunately, this approach relies on user guidance for control decisions, its termination is not always guaranteed, and because of it its automation is complicated.

Specialization in logic programming is commonly referred to as partial deduction. It was introduced by Komorowski [Komorowski(1982)] and formalized by Lloyd and Shepherdson [Lloyd and Shepherdson(1991)]. Comparing to fold/unfold transformations, partial deduction is less powerful, because it considers every atom on its own and does not track dependencies between variables. However, it is significantly easier to control and can be automated.

The main drawback of partial deduction is addressed by Leuschel with conjunctive partial deduction [De Schreye et al.(1999)] in the ECCE system. This method makes use of the interaction between conjuncts for specialization, removing some repeating traversals of data structures as a

[1]The website of the MERCURY programming language `https://mercurylang.org/`

[2]The website of the CURRY programming language `https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/`

[3]The website of the MINIKANREN programming language `http://minikanren.org/`

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 performed worse than the original ones.

Partial evaluators are categorized into offline and online methods, depending on whether control decisions are made before or during the specialization stage. LOGEN is the implementation of the offline approach for logic programming, developed by Leuschel [Leuschel et al.(2004)]. It includes an automatic binding-time analysis to derive annotations used to guide the specialization process. Offline specialization usually takes less time than online, and is capable to generate shorter and more efficient programs.

The fact that majority of PROLOG implementations do not impose a type system may be seen as a disadvantage when it comes to optimizations. MERCURY developed a strong static type and mode system that can be used in compilation [Overton et al.(2002), Overton(2003)]. Mode analysis embodies data-flow analysis that makes it possible to compile the same definition into several functions specialized for the given direction.

Chapter 3

Method A

Chapter 4

Conclusion

This concludes the thesis.

Bibliography

[Abramov and Glück(2000)] Sergei Abramov and Robert Glück. 2000. Combining Semantics with Non-standard Interpreter Hierarchies. In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, Sanjiv Kapoor and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–213. `https://doi.org/10.1007/3-540-44450-5_16`

[Abramov and Glück(2002)] Sergei Abramov and Robert Glück. 2002. *Principles of Inverse Computation and the Universal Resolving Algorithm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 269–295. `https://doi.org/10.1007/3-540-36377-7_13`

[Aman et al.(2020)] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan, et al. 2020. Foundations of Reversible Computation. *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405 12* (2020), 1–40. `https://doi.org/10.1007/978-3-030-47361-7_1`

[Augustsson et al.(2023)] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 203 (aug 2023), 31 pages. `https://doi.org/10.1145/3607845`

[Battani and Meloni(1973)] G Battani and H Meloni. 1973. Interpreteur du langage de programmation Prolog. *Grouped'Intelligence Artifieielle, Marseille-Luminy* (1973).

[Byrd et al.(2017)] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–26. `https://doi.org/10.1145/3110252`

[Cohen(1988)] Jacques Cohen. 1988. A view of the origins and development of Prolog. *Commun. ACM* 31, 1 (1988), 26–36.

[De Schreye et al.(1999)] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming* 41, 2-3 (1999), 231–277.

[Ershov(1982)] A.P. Ershov. 1982. Mixed computation: potential applications and problems for study. *Theoretical Computer Science* 18, 1 (1982), 41–67. `https://doi.org/10.1016/0304-3975(82)90111-6`

[Friedman et al.(2005)] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. The MIT Press. https://doi.org/10.7551/mitpress/5801.001.0001

[Jones(1996)] Neil D. Jones. 1996. An introduction to partial evaluation. *ACM Comput. Surv.* 28, 3 (sep 1996), 480–503. https://doi.org/10.1145/243439.243447

[Jones et al.(1993)] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[Kiselyov et al.(2005)] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 192—-203. https://doi.org/10.1145/1086365.1086390

[Klyuchnikov(2009)] Ilya G Klyuchnikov. 2009. Supercompiler HOSC 1.0: under the hood. *Keldysh Institute Preprints* 63 (2009), 1–28.

[Komorowski(1982)] H Jan Komorowski. 1982. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 255–267.

[Komorowski(1992)] Jan Komorowski. 1992. An introduction to partial deduction. In *Meta-Programming in Logic: Third International Workshop, META-92 Uppsala, Sweden, June 10–12, 1992 Proceedings 3*. Springer, 49–69.

[Leuschel(2002)] Michael Leuschel. 2002. Homeomorphic embedding for online termination of symbolic methods. *The essence of computation: complexity, analysis, transformation* (2002), 379–403.

[Leuschel et al.(2004)] Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. 2004. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming* 4, 1-2 (2004), 139–191.

[Lloyd and Shepherdson(1991)] John W. Lloyd and John C Shepherdson. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming* 11, 3-4 (1991), 217–242.

[Lozov et al.(2019)] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational Interpreters for Search Problems. In *miniKanren and Relational Programming Workshop*. 43. http://minikanren.org/workshop/2019/minikanren19-final3.pdf

[Lozov et al.(2018)] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58. https://doi.org/10.1007/978-3-319-89719-6_3

[Mitchell(2010)] Neil Mitchell. 2010. Rethinking supercompilation. *ACM Sigplan Notices* 45, 9 (2010), 309–320.

[Overton(2003)] David Overton. 2003. *Precise and expressive mode systems for typed logic programming languages*. Citeseer.

[Overton et al.(2002)] David Overton, Zoltan Somogyi, and Peter J Stuckey. 2002. Constraint-Based Mode Analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 109–120. https://doi.org/10.1145/571157.571169

[Pettorossi and Proietti(1994)] Alberto Pettorossi and Maurizio Proietti. 1994. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming* 19 (1994), 261–320.

[Pettorossi and Proietti(1996)] Alberto Pettorossi and Maurizio Proietti. 1996. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 360–414.

[Robinson(1965)] John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* 12, 1 (1965), 23–41.

[Rozplokhas and Boulytchev(2022)] Dmitry Rozplokhas and Dmitry Boulytchev. 2022. Scheduling Complexity of Interleaving Search. In *International Symposium on Functional and Logic Programming*. Springer, 152–170.

[Sørensen and Glück(1995)] Morten Heine Sørensen and Robert Glück. 1995. An Algorithm of Generalization in Positive Supercompilation.. In *ILPS*, Vol. 95. 465–479.

[Turchin(1986)] Valentin F Turchin. 1986. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 3 (1986), 292–325.

[Turchin(1988)] Valentin F Turchin. 1988. The algorithm of generalization in the supercompiler. *Partial Evaluation and Mixed Computation* 531 (1988), 549.

[Turchin(1989)] Valentin F Turchin. 1989. *REFAL-5: programming guide & reference manual.* New England Publ.

[Verbitskaia et al.(2021)] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2021. An Empirical Study of Partial Deduction for miniKanren. In Proceedings of the 9th International Workshop on *Verification and Program Transformation,* Luxembourg, Luxembourg, 27th and 28th of March 2021 *(Electronic Proceedings in Theoretical Computer Science, Vol. 341)*, Alexei Lisitsa and Andrei P. Nemytykh (Eds.). Open Publishing Association, 73–94. https://doi.org/10.4204/EPTCS.341.5

[Verbitskaia et al.(2024)] Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 2024. A Case Study in Functional Conversion and Mode Inference in miniKanren. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation* (, London, UK,) *(PEPM 2024)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3635800.3636966