Towards Efficient Search
Leveraging Relational Interpreters and Partial Deduction Techniques

by

Ekaterina Verbitskaia

a Thesis submitted in partial fulfillment
of the requirements for the degree of

**Doctor of Philosophy
in Computer Science**

Approved Dissertation committee

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

Submission: May 21, 2024

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Verbitskaia, Ekaterina
Matriculation number	20333664
What kind of thesis are you submitting	PhD-Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however. The Thesis has been written independently and has not been submitted at any other university for the conferral of a PhD degree; neither has the thesis been previously published infull.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung. Diese Arbeit wurde in der vorliegenden Form weder einer anderen Prüfungsbehörde vorgelegt noch wurde das Gesamtdokument bisher veröffentlicht.

. . . .

Date, Signature

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 [23]. 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 [1, 2, 3]. 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 [24], 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 [11, 7]. 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 [15]. 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 [31]. 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 [9, 36]. 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 [18]. 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 [36]. Then we combined it with the functional conversion in an effort to get even greater performance increase [37].

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.

1.1 The Goal of the Research

1.2 Tasks

Chapter 2

Background

2.1 Logic Programming Languages

Over the years, multiple logic programming languages have been developed, with PROLOG [5] 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 [30] (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 [8]. 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 [4]. This new generation of logic languages combines the paradigms of logic and more mainstream fuctional programming. MERCURY and CURRY are stand-alone 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 [15] 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 [33]. It was designed for the Refal programming language [35], 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 [16, 25]. Numerous modifications have also emerged, featuring alternative termination strategies, generalization, and splitting techniques [19, 32, 34].

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 [10] and was mostly aimed at imperative languages. A lot of effort has been extended to partial evaluation [14, 13] 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 [29, 28]. 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 [17] and formalized by Lloyd and Shepherdson [21]. 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.

[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/`

The main drawback of partial deduction is addressed by Leuschel with conjunctive partial deduction [9] in the ECCE system. This method makes use of the interaction between conjuncts for specialization, removing 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 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 [20]. 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 [27, 26]. Mode analysis embodies data-flow analysis that makes it possible to compile the same definition into several functions specialized for the given direction.

2.3 Relational Programming and miniKanren

Chapter 3

Relational Interpreters for Search

Many programming problems can be broadly categorized as either verification tasks or solution-finding tasks. The former involves checking whether a given solution meats certain criteria, which is often straightforward to implement and inexpensive to run. On the other hand, the latter requires discovering a solution that satisfies the problem's constraints, which can be significantly more complex and resource-intensive. To illustrate the idea, let us consider the verifier-based definition of the NP complexity class [12].

Definition 1: NP complexity class

We say that a language $\mathcal{L}$ is in the complexity class NP (Nondeterministic Polynomial time) if there is a predicate $V_{\mathcal{L}}$ such that

$$\forall \omega : \omega \in \mathcal{L} \Leftrightarrow \exists p_\omega : V_{\mathcal{L}}(\omega, p_\omega),$$

where p_ω is of size polynomial on ω, and we can recognize $V_{\mathcal{L}}$ in polynomial time.

In this definition, p_ω serves as a proof of the fact that $\omega \in \mathcal{L}$. For example, if $\mathcal{L}$ is the set of all Hamiltonian graphs ω, then $V_{\mathcal{L}}$ is a predicate that, checks whether a path p_ω is Hamiltonian in the graph ω. Implementing $V_{\mathcal{L}}$ demands little effort: one only needs to make sure that the path forms a sequence of vertices with no repetitions. However, the implementation of the predicate does not reveal anything regarding the *search procedure* which will compute p_ω by ω. Notably, a related problem is studied in the area of *relational interpreters.*

Definition 2: Relational interpreter

A *relational interpreter* for a language $\mathcal{L}$ is a relation $eval^o_{\mathcal{L}}$ that connects a program $p^{\mathcal{L}}$ written in the language $\mathcal{L}$, its input i, and its output o, which corresponds to the semantics $[\![\cdot]\!]_{\mathcal{L}}$ of the program $p^{\mathcal{L}}$ applied to the input:

$$eval^o_{\mathcal{L}}(p^{\mathcal{L}}, i, o), \text{such that } o = [\![p^{\mathcal{L}}]\!]_{\mathcal{L}}(i)$$

Using this terminology, we can view a verification predicate $V_{\mathcal{L}}$ as a relational interpreter that connects a program ω and its input p_ω with either $true$ or $false$.

$$V_{\mathcal{L}}(\omega, p_\omega) = b \Leftrightarrow eval^o_{\mathcal{L}}(\omega, p_\omega, b)$$

This analogy sheds light on how the multimodal nature of relational programming can be employed to turn a verifier into a solver. The interpreter can run in the verification mode if we query for the output and pass ground values for ω and p_ω.

$$run\ q\ eval^o_{\mathcal{L}}(\omega, p_\omega, q)$$

Conversely, by passing only ω and the output b, a witness p_ω can be computed. In this way, the relational interpreter functions as a solver implementing a search procedure.

$$run\ q\ eval^o_{\mathcal{L}}(\omega, q, b)$$

In general, there is no limitation on the output being of type Boolean: it may take any form that suits the problem the best. For example, we can observe the duality between program interpretation and program synthesis by running a relational interpreter in the appropriate directions. Similar parallels can be drawn when considering related problems such as type checking, type inference, and type inhabitation, as well as many other problems.

3.1 Relational Conversion

Writing a relational interpreter can be done from scratch by carefully considering the semantics of the language being evaluated. However, in the majority of cases its structure follows the one of an interpreter implemented in a functional language. Developers often choose this approach because programming in the functional paradigm feels more intuitive to them than designing programs in the relational paradigm. It is also facilitated by the existence of relational conversion — an automatic procedure capable of producing a relational program from its functional counterpart. Let us illustrate the conversion on a small example.

Consider evaluating a propositional formula, which is constructed from Boolean literals, integer-named variables, and other formulas using Boolean connectives, namely conjunction, disjunction, and negation: see figure 3.1. There might be other ways to construct a formula, such as implication and exclusive or, but their treatment does not differ significantly; thus we do not address them. We use integers as variable names to make it easier to represent variable substitutions as lists of Boolean constants. The value of variable `n` is the `n`-th element of the substitution list. The example formula $\neg v_0 \wedge (v_1 \vee False)$ is

```
data Formula
  = Lit Bool
  | Var Int
  | Neg Formula
  | Conj Formula Formula
  | Disj Formula Formula
```

Figure 3.1: The data type representation for a propositional formula

encoded as `Conj (Neg (Var 0)) (Disj (Var 1) (Lit False))`.

To determine the value of a formula, the functional interpreter presented in listing 3.2 deconstructs the formula by pattern matching and calculates the result according to its structure. Evaluating a Boolean literal is straightforward. When a formula is a variable, we look up its value in the substitution using the function `elem`. Note that `elem` is partial as we assume every variable in the formula has some value in the substitution. The three other cases necessitate recursive calls to the interpreter and combining their results according to the type of the formula. Evaluating the example formula with the substitution list `[False, True]` results in `True`.

```
eval :: [Bool] -> Formula -> Bool
eval subst (Lit b) = b
eval subst (Var v) = elem subst v
eval subst (Neg z) = not $ eval subst z
eval subst (Conj x y) = eval subst x && eval subst y
eval subst (Disj x y) = eval subst x || eval subst y

elem :: [a] -> Int -> a
elem (h : t) 0 = h
elem (_ : t) n = elem t (n - 1)
```

Figure 3.2: The interpreter for a propositional formula in a functional language

The simplest way to transform a function into a relational programming language is described in [6]. It is done is several steps. First, every nested function call is unnested and its result is bound to a variable. Second, an extra parameter `res` is added to the relational counterpart of the function being translated to associate its result with. Third, a pattern matching is transformed into a disjunction. Each disjunct unifies the scrutinee with the pattern in conjunction with the result of translating the body of the branch. Finally, each function call is replaced with the corresponding relation call, the result is unified with the extra argument `res`, and the calls and unifications are combined into conjunctions.

This conversion applied to the functional interpreter of the proposition language results in the code presented in figure 3.3. Note that Boolean operators, namely `&&`, `||`, and `not`, are converted into their relational counterparts `ando`, `oro`, and `noto`. We assume that these operators have some implementation accessible to the converter, which means that they are written by the user, their built-in implementations can be inspected, or their relational counterparts are hard-coded. In this particular example, we suppose that the Boolean operators have table-based implementations instead of more efficient short-circuit implementations for no particular reason except for better readability.

Another feature to note is that the order of calls in the conjunctions is determined by the structure of the expression in the body of the source function. This is why each disjunct starts with the unification generated from the pattern matching, and the calls to `evalo` are done before calling relations for boolean connectives `ando`, `oro`, and `noto`. This corresponds to the semantics of the functional language and allows computing the same answers when the constructed relation is run in the forward direction. Lastly, the order of disjuncts is also determined by the original program, following the sequence of pattern match clauses.

```
-- evalo :: [Bool] -> Formula -> Bool -> Goal
evalo subst fm u =
  fresh x, y, v, w, z in
    (fm == Lit u) |
    (fm == Var z & elemo z subst u) |
    (fm == Neg x & evalo subst x v & noto v u) |
    (fm == Conj x y & evalo subst x v & evalo subst y w & ando v w u) |
    (fm == Disj x y & evalo subst x v & evalo subst y w & oro v w u);

-- elemo :: [a] -> Int -> a -> Goal
elemo subst n res =
  fresh h, t, n1 in
    (s == (h :: t) & n == Zero & res == h) |
    (s == (h :: t) & n == Succ n1 & elemo t n1 res);

-- ando :: Bool -> Bool -> Bool -> Goal
ando x y b =
  (x == Trueo & y == Trueo & b == Trueo) |
  (x == Falso & y == Trueo & b == Falso) |
  (x == Trueo & y == Falso & b == Falso) |
  (x == Falso & y == Falso & b == Falso);

-- oro :: Bool -> Bool -> Bool -> Goal
oro x y b =
  (x == Trueo & y == Trueo & b == Trueo) |
  (x == Falso & y == Trueo & b == Trueo) |
  (x == Trueo & y == Falso & b == Trueo) |
  (x == Falso & y == Falso & b == Falso);

-- noto :: Bool -> Bool -> Goal
noto x b =
  (x == Trueo & b == Falso) |
  (x == Falso & b == Trueo);
```

Figure 3.3: The relational interpreter for a propositional formula

Querying `evalo` in different directions allows finding answers for various problems. For example, we can evaluate a formula in a given substitution, which associates `q` with the only possible answer `True`.

```
Q: run q (evalo [False, True]
                (Conj (Neg (Var 0)) (Disj (Var 1) (Lit False)))
                q)
A: q = True
```

In general, a query can have multiple free variables. For instance, we can leave both the substitution and the last argument to be free variables. In the following case, multiple possible answers exist, including `s` $\mapsto$ `[False, True]`, `q` $\mapsto$ `True`:

```
Q: run q, s (evalo s
                   (Conj (Neg (Var 0))
```

```
                               (Disj (Var 1) (Lit False)))
                        q)
A: s = [True, True]; q = False
A: s = [True, False]; q = False
A: s = [False, True]; q = True
A: s = [False, False]; q = False
```

Finally, querying a relation with the last argument `res` known makes the relational interpreter function as a solver. Consider the following query which produces an infinite number of formulas, that evaluate to `True` in the substitution `[False, True]`, including `(Conj (Neg (Var 0)) (Disj (Var 1) (Lit False)))`.

```
Q: run q (evalo [False, True]
                q
                True)
A: q = Var 1
A: q = Not (Var 0)
...
A: q = (Conj (Neg (Var 0)) (Disj (Var 1) (Lit False)))
...
```

The conversion based on unnesting demonstrates the fundamental principles of creating a relation from a function, but it is quite limited. In reality, functional programming often involves using higher-order functions which the described method does not handle. A more complicated typed relational conversion described in [24, 22] and implemented as a separate tool[1] supports this important feature by changing the target language to allow higher-order functions as arguments of relations. To learn more about how this conversion works, the reader is directed to the original papers.

3.2 Reasons of Poor Performance of Relational Interpreters

Relational conversion is designed in such a way that it preserves the semantics of the original function when the relation is run in the forward direction. It means that this direction is always deterministic, and the execution time experiences not more than a linear slowdown [22]. However, running a relation in any other direction can demonstrate unpredictable performance. We cannot compare it with the performance of the original program, since it can only work as a function. However, we can sometimes see that the relation can be modified to be more efficient. Let us consider some sources of inefficiency on the example introduced in the previous subsection.

Imagine running the relation to generate formulas that evaluate to `True` in some given substitution: `run q (evalo [...]  q True)`. Since miniKanren evaluates conjunctions from left to right by default, the unification of the second argument `fm` is done first. This unification does not provide any useful data leaving variables under constructors free leading to subsequent calls to the `evalo` and `elemo` relations with only the substitutions known. Having finished evaluating, these calls bind variables which are then passed to the

[1] The tool for automatic relational conversion `noCanren`: `https://github.com/PLTools/noCanren/`

Boolean connectives `ando`, `oro`, and `noto`. Because all their arguments are now known, the latter relations serve as predicates, filtering out those sub-formulas whose evaluation do not make the result `True`. This is recognized as "generate and test" behavior and often leads to poor performance.

Fortunately, reordering the conjuncts in such a way that the Boolean relations are executed first, gets rid of the undesired behavior. For instance, running `ando v w True` limits the possible values of `v` and `w` to `True`. With this extra information, the subsequent calls to `evalo` only generate the formulas that actually contribute to the answer the user is interested in. By doing this, the execution time is reduced this many times citation.

Consider adding the example with $f_1 x \&\& f_2 x$ from the relational interpreters for search paper.

As evidenced by these examples, there might not be a single optimal order of conjuncts that works well in each direction. Finding such an order is likely (citation needed) an undecidable problem. There are however several approaches that aim at finding if not the best, then a better order which suits the given direction. The approach we are taking in this dissertation includes mode analysis and specialization.

There are other sources of overhead that are deeply rooted in the nature of relational programming in miniKanren. One of them is *scheduling complexity* [31]. Its effect has been observed when comparing two possible implementations of the `appendo` relation that only differ in the order of two conjuncts but demonstrate asymptotically different performance. One of them exhibits performance linear with the size of the first list, while the other's performance is polynomial. The instinctive feeling that the cause of this discrepancy lays in a different number of unifications performed does not hold up. In fact, both relations make the same number of unifications, and there is another explanation of what is happening. Add a figure from Rozphlohos' paper.

Put simply, miniKanren maintains a lazy data structure that is used in the decomposition of goals into basic unifications, performing them in an order determined by the interleaving search, and threading them together to compute the final answers. This structure stays constant size in one of the relation, but grows linearly in the other. The paper [31] provides a way to estimate the scheduling complexity, but it necessitates using a human oracle which hinders the adoption of this metrics in specialization and other automatic optimization efforts.

Chapter 4

Specialization

4.1 Partial Deduction

4.2 Online vs Offline Approaches

4.3 Conservative Partial Deduction

4.4 Offline Specialization of miniKanren

Chapter 5

Functional Conversion

5.1 Mode Analysis

5.2 Intermediate Functional Language

5.3 Conversion to a Target Language

Chapter 6

Evaluation

6.1 Benchmark

6.2 Time Measurements

Chapter 7

Conclusion

This concludes the thesis.

Bibliography

[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] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. The verse calculus: A core calculus for deterministic functional logic programming. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023.

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

[6] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations.* PhD thesis, Indiana University, 2009.

[7] 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.

[8] Jacques Cohen. A view of the origins and development of prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[9] 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.

[10] A.P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18(1):41–67, 1982.

[11] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[13] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, sep 1996.

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

[15] 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.

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

[17] H Jan Komorowski. 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*, pages 255–267, 1982.

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

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

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

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

[22] Petr Lozov. *Automated Synthesis and Efficient Execution of Relational Programs*. PhD thesis, Saint Petersburg State University, 2022.

[23] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. Relational interpreters for search problems. In *miniKanren and Relational Programming Workshop*, page 43, 2019.

[24] 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.

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

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

[27] David Overton, Zoltan Somogyi, and Peter J Stuckey. Constraint-based mode analysis of mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 109–120, 2002.

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

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

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

[31] Dmitry Rozplokhas and Dmitry Boulytchev. Scheduling complexity of interleaving search. In *International Symposium on Functional and Logic Programming*, pages 152–170. Springer, 2022.

[32] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *ILPS*, volume 95, pages 465–479, 1995.

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

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

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

[36] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. An empirical study of partial deduction for minikanren. In Alexei Lisitsa and Andrei P. Nemytykh, editors, Proceedings of the 9th International Workshop on *Verification and Program Transformation,* Luxembourg, Luxembourg, 27th and 28th of March 2021, volume 341 of *Electronic Proceedings in Theoretical Computer Science*, pages 73–94. Open Publishing Association, 2021.

[37] Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 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*, PEPM 2024, page 107–118, New York, NY, USA, 2024. Association for Computing Machinery.