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 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 [**?**], 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 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 [**?**]. 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 [**?**, **?**]. 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 `f` $\texttt{x}_{static}$ $\texttt{y}_{dynamic}$, it can be transformed into a more efficient function `f_`$\texttt{x}_{static}$ 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` $\texttt{x}_{static}$ $\texttt{y}_{dynamic}$ `==` `f_`$\texttt{x}_{static}$ $\texttt{y}_{dynamic}$.

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 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 [**?**]. Then we combined it with the func-

tional conversion in an effort to get even greater performance increase [**?**].

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.