Relational Interpreters for Search Problems*

PETR LOZOV, EKATERINA VERBITSKAIA, and DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We address the problem of constructing a solver for a certain search problem from its solution verifier. The main idea behind the approach we advocate is to consider a verifier as an interpreter which takes a data structure to search in as a program and a candidate solution as this program's input. As a result the interpreter returns "*true*" if the candidate solution satisfies all constraints and "*false*" otherwise. Being implemented in a relational language, a verifier becomes capable of finding a solution as well. We apply two techniques to make this scenario realistic: *relational conversion* and *supercompilation*. Relational conversion makes it possible to convert a first-order functional program into relational form, while supercompilation (in the form of conjunctive partial deduction (CPD)) — to optimize out redundant computations. We demonstrate our approach on a number of examples using a prototype tool for OCanren — an implementation of miniKanren for OCaml, — and discuss the results of evaluation.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; **Source code generation**;

Additional Key Words and Phrases: relational programming, relational interpreters, search problems

1 INTRODUCTION

Verifying a solution for a problem is much easier than finding one — this common wisdom can be confirmed by anyone who used both to learn and to teach. This observation can be justified by its theoretical applications, thus being more than informal knowledge. For example, let us have a language $\mathcal{L}$. If there is a predicate $V_{\mathcal{L}}$ such that

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

(with p_ω being of size, polynomial on ω) and we can recognize $V_{\mathcal{L}}$ in a polynomial time, then we call $\mathcal{L}$ to be in the class NP [Garey and Johnson 1990]. Here p_ω plays role of a justification (or proof) for the fact $\omega \in \mathcal{L}$. For example, if $\mathcal{L}$ is a language of all hamiltonian graphs, then $V_{\mathcal{L}}$ is a predicate which takes a graph ω and some path p_ω and verifies that p_ω is indeed a hamiltonial path in ω. The implementation of the predicate $V_{\mathcal{L}}$, however, tells us very little about the *search procedure* which would calculate p_ω as a function of ω. For the whole class of NP-complete problems no polynomial search procedures are known, and their existence at all is a long-standing problem in the complexity theory.

There is, however, a whole research area of *relational interpreters*, in which a very close problem is addressed. Given a language $\mathcal{L}$, its *interpreter* is a function $\texttt{eval}_{\mathcal{L}}$ which takes a program $p^{\mathcal{L}}$ in the language $\mathcal{L}$ and an input i and calculates some output such that

$$\texttt{eval}_{\mathcal{L}}(p^{\mathcal{L}}, i) = [\![p^{\mathcal{L}}]\!]_{\mathcal{L}}\,(i)$$

*This work was partially suppored by the grant 18-01-00380 from The Russian Foundation for Basic Research

Authors' address: Petr Lozov, lozov.peter@gmail.com; Ekaterina Verbitskaia, kajigor@gmail.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University, Russia , JetBrains Research, Russia.

miniKanren.org/workshop/2019/8-ART3

where $[\![\bullet]\!]_{\mathcal{L}}$ is the semantics of the language $\mathcal{L}$. In these terms, a verification predicate $V_{\mathcal{L}}$ can be considered as an interpreter which takes a program ω, its input p_ω and returns *true* or *false*. A *relational* interpreter is an interpreter which is implemented not as a function $\texttt{eval}_{\mathcal{L}}$, which calculates the output from a program and its input, but as a relation $\texttt{eval}^o_{\mathcal{L}}$ which connects a program with its input and output. This alone would not have much sense, but if we allow the arguments of $\texttt{eval}^o_{\mathcal{L}}$ to contain *variables* we can consider relational interpreter as a generic search procedure which determines the values for these variables making the relation hold. Thus, with relational interpreter it is possible not only to calculate the output from an input, but also to run a program in an opposite "direction", or to synthesize a program from an input-output pair, etc. In other words, relational verification predicate is capable (in theory) to both *verify* a solution and *search* for it.

Implementing relational interpreters amounts to writing it in a relational language. In principle, any conventional language for logic programming (Prolog [Clocksin and Mellish 2003], Mercury [Somogyi et al. 1996], etc.) would make the job. However, the abundance of extra-logical features and the incompleteness of default search strategy put a number of obstacles on the way. There is, however, a language specifically designed for pure relational programming, and, in a narrow sense, for implementing relational interpreters — miniKanren [Friedman et al. 2005]. Relational interpreters, implemented in miniKanren, demonstrate all their expected potential: they can synthesize programs by example, search for errors in partially defined programs [Byrd et al. 2017], produce self-evaluated programs [Byrd et al. 2012], etc. However, all these results are obtained for a family of closely related Scheme-like languages and require a careful implementation and even some *ad-hoc* optimizations in the relational engine.

From a theoretical standpoint a single relational interpreter for a Turing-complete language is sufficient: indeed, any other interpreter can be turned into a relational one just by implementing it in a language, for which relational interpreter already exists. However, the overhead of additional interpretation level can easily make this solution impractical. The standard way to tackle the problem is partial evaluation or specialization [Jones et al. 1993]. A *specializer* $\texttt{spec}_{\mathcal{M}}$ for a language $\mathcal{M}$ for any program $p^{\mathcal{M}}$ in this language and its partial input i returns some program which, being applied to the residual input x, works exactly as the original program on both i and x:

$$\forall x \ : \ [\![\texttt{spec}_{\mathcal{M}}\,(p^{\mathcal{M}}, i)]\!]_{\mathcal{M}}\,(x) = [\![p^{\mathcal{M}}]\!]_{\mathcal{M}}\,(i, x).$$

If we apply a specializer to an interpreter and a source program, we obtain what is called *the first Futamura projection* [Futamura 1971]:

$$\forall i \ : \ [\![\texttt{spec}_{\mathcal{M}}\,(\texttt{eval}^{\mathcal{M}}_{\mathcal{L}}, p^{\mathcal{L}})]\!]_{\mathcal{M}}\,(i) = [\![\texttt{eval}^{\mathcal{M}}_{\mathcal{L}}]\!]_{\mathcal{M}}\,(p^{\mathcal{L}}, i).$$

Here we added an upper index $\mathcal{M}$ to $\texttt{eval}_{\mathcal{L}}$ to indicate that we consider it as a program in the language $\mathcal{M}$. In other words, the first Futamura projection specializes an interpreter for a concrete program, delivering the implementation of this program in the language of interpreter implementation. An important property of a specializer is *Jones-optimality* [Jones et al. 1993], which holds when it is capable to completely eliminate the interpretation overhead in the first Futamura projection. In our case $\mathcal{M}$ = miniKanren, from which we can conclude that in order to eliminate the interpretation overhead we need a Jones-optimal specializer for miniKanren. Although implementing a Jones-optimal specializer is not an easy task even for simple functional languages, there is a Jones-optimal specializer for a logical language [Leuschel et al. 2004], but not for miniKanren.

The contribution of this paper is as follows:

- We demonstrate the applicability of relational programming and, in particular, relational interpreters for the task of turning verifiers into solvers.
- To obtain a relational verifier from a functional specification we apply *relational conversion* [Byrd 2009; Lozov et al. 2018] — a technique which for a first-order functional program directly constructs its relational counterpart. Thus, we introduce a number of new relational interpreters for concrete search problems.

- We employ supercompilation in the form of conjunctive partial deduction (CPD) [De Schreye et al. 1999] to eliminate the redundancy of a generic search algorithm caused by partial knowledge of its input.
- We give a number of examples and perform an evaluation of various solutions for the approach we address.

Both relational conversion and conjunctive partial deduction are done in an automatic manner. The only thing one needs to specify is the known arguments or the execution direction of a relation.

As concrete implementation of miniKanren we use OCanren [Kosarev and Boulytchev 2016] — its embedding in OCaml; we use OCaml to write functional verifiers; our prototype implementation of conjunctive partial deduction is written in Haskell.

The paper is organized as follows. In Section 2 we give a complete example of solving a concrete problem — searching for a path in a graph, — with relational verifier. Section 3 recalls the cornerstones of relational programming in miniKanren and the relational conversion technique. In Section 4 we describe how conjunctive partial deduction was adapted for relational programming. Section 5 presents the evaluation results for concrete solvers built using the technique in question. The final section concludes.

2 SEARCHING FOR PATHS IN A GRAPH WITH A RELATIONAL VERIFIER

In this section we demonstrate how to solve a concrete problem of searching for paths in a directed graph with a relational verifier. A directed graph is a tuple $(N, E, start, end)$, where N is a finite set of *nodes*, E is a finite set of *edges*, functions $start, end : E \to N$ return a start and an end nodes for a given edge respectively. A path in a directed graph is a sequence:

$$\langle n_0, e_0, n_1, e_1, \ldots, n_k, e_k, n_{k+1} \rangle$$

such that

$$\forall i \in \{0 \ldots k\} \ : \ n_i = start\,(e_i) \text{ and } n_{i+1} = end\,(e_i).$$

The problem of searching for paths in a graph is to find a set $\{p \mid p \text{ is a path in } g\}$, where g is a graph. There are many concrete algorithms which search for paths in a graph. Implementing any of them involves determining in which way to traverse the graph, how to ensure one does not get stuck exploring a cycle in the graph (a cycle is a path in the graph of form $\langle n_0, e_0, \ldots, n_k, e_k, n_0 \rangle$), how to ensure one path is not processed multiple times, and so on. A much easier task is to implement a simple verifier, which checks if a sequence is indeed a path in a graph, and generate the path searching routine from it by the relational conversion.

Below is the implementation of the verifier "isPath". This function takes as an input a list of nodes "ns" and a graph "g". We represent the graph as a list of edges, stipulating there are no parallel edges. Each edge e is represented as a pair of nodes (n, m), where $n = start(e)$, $m = end(e)$. Given $ns = [n_0, \ldots, n_{k+1}]$ and a graph $g = [e_0, \ldots, e_l]$, the function returns true, if $\exists i_0 \ldots i_k$ such that $\langle n_0, e_{i_0}, n_1, e_{i_1}, \ldots, e_{i_k}, n_{k+1} \rangle$ is a path in g.

```
1  let rec isPath ns g =
2    match ns with
3    | x1 :: x2 :: xs  → elem (x1, x2) g && isPath (x2 :: xs) g
4    | [_]             → true
```

The function "elem" checks if an edge "e" exists in the graph "g". We omit the definition of equality check for edges "eq", since it is trivial to implement and is not relevant for the example.

```
let rec elem e g =
  match g with
  | []      → false
  | x :: xs → if eq e x then true else elem e xs
```

We stipulate that a path must include at least two nodes, since searching for shorter paths is trivial. Line 3 of the "isPath" definition checks that the first two nodes of the list form an edge of the graph. Then it checks that what is left after deleting the first node from the list is still a path in the graph. Line 4 may come off a little counterintuitive, since it states that a path which includes a single arbitrary node is in the input graph. However we only execute this branch by a recursive call of "isPath", which only happens after we have already ensured with the call to the "elem" function that the said node is in the graph.

The relational conversion of the verifier function "isPath" generates a relation "$\texttt{isPath}^o$" defined for a path "ns", a graph "g" and a boolean value "res", which is true if "ns" is a path in the graph "g" and false otherwise. The function "elem" is transformed into a relation "$\texttt{elem}^o$" defined for an edge "e", a graph "g" and a boolean value "res", which is true if "e" is an edge in the graph "g" and false otherwise. The result of the relational conversion of the functions "isPath" and "elem" is presented below.

```
5  let rec elem^o e g res = conde [
6    (g ≡ nil () ∧ res ≡ ↑false);
7    (fresh (x xs resEq) (
8      (g ≡ x % xs) ∧
9      (eq^o e x resEq) ∧
10     (conde [
11       (resEq ≡ ↑true  ∧ res ≡ ↑true);
12       (resEq ≡ ↑false ∧ elem^o e xs res)])))]
13
14 let rec isPath^o ns g res = conde [
15   (fresh (el) (
16     (ns ≡ el % nil ()) ∧
17     (res ≡ ↑true));
18  (fresh (x1 x2 xs resElem resIsPath) (
19     (ns ≡ x1 % (x2 % xs)) ∧
20     (elem^o (pair x1 x2) g resElem) ∧
21     (isPath^o (x2 % xs) g resIsPath) ∧
22     (conde [
23       (resElem ≡ ↑false ∧ res ≡ ↑false);
24       (resElem ≡ ↑true  ∧ res ≡ resIsPath)])))]
```

Here we use the syntax of OCanren. A new relation is defined as a recursive function with the keywords "**let rec**". The body of the relation is a goal created with the following goal constructors.

- Disjunction $g_1 \vee g_2$, where g_1, g_2 — some goals. The two goals are evaluated independently and their results are combined.
- Disjunction of goal list **conde** $[g_1; \ldots; g_n]$, where $g_1; \ldots; g_n$ — some goals.
- Conjunction $g_1 \wedge g_2$, where g_1, g_2 — some goals. The goal g_2 is evaluated only if the evaluation of g_1 succeeded; the evaluation of g_2 uses the results of g_1.
- Syntactic unification $t_1 \equiv t_2$, where t_1, t_2 — some terms. Unification is a basic goal constructor. If t_1 and t_2 can be unified, the goal is considered successful and failed otherwise.
- Relation call $r^n t_1 \ldots t_n$ where r^n is a name of some n-ary relation, and t_i are terms.
- To introduce fresh variables into scope, one should use **fresh** $(\overline{x})$ g, where $\overline{x}$ is a list of variable names.

Besides goal constructors we use some syntactic sugar for values and lists. "↑" is used to transform a value into a logic value. The empty list is represented as "`nil ()`", and to construct a new list from a value "`h`" and a list "`t`" we use "`h % t`". A tuple of "`x`" and "`y`" is created with "`pair x y`".

Regrettably, this relational interpreter suffers from poor performance. Query "`isPath`^{o} `q <graph>` **true**" for path searching takes more than ten minutes even for graphs of 5 nodes. This is somewhat expected, considering that the relational conversion generates a relation which can be used for many different queries, which is excessive when any particular query is in question. This is, of course, not a desirable behaviour. Fortunately, further transformation of the relation can improve the performance.

For example, if we consider a query "`isPath`^{o} `q <graph>` ↑**true**", we can simplify lines 18 through 24 of its definition. First, we notice that, having "`res`" be equal to "↑**true**", we can safely remove the disjunct in line 23, after what the whole "**conde**" becomes unnecessary and can be removed. After moving the unifications for "`resElem`" and "`resIsPath`" to the top level, we get the following equivalent definition of the "`isPath`^{o}" relation. Note, that the call to the "`elem`^{o}" relation is done with the last argument being unified with "↑**true**", so further specialization is still possible.

```
25 let rec isPath^o ns g res = conde [
26   (fresh (el) (
27     (ns ≡ el % nil ()) ∧
28     (res ≡ ↑true)));
29   (fresh (x1 x2 xs resElem resIsPath) (
30     (resElem ≡ ↑true) ∧
31     (resIsPath ≡ ↑true) ∧
32     (ns ≡ x1 % (x2 % xs)) ∧
33     (elem^o (pair x1 x2) g resElem) ∧
34     (isPath^o (x2 % xs) g resIsPath)))]
```

The specialized version of the relation is much more performant than the original one. Before, searching paths of length 5 took more than 10 minutes while the specialized version finds paths of length 10 in the graph with 100 edges in a few seconds.

This transformation can be performed automatically with conjunctive partial deduction. The result of partially deducing the "`isPath`^{o} `q p` ↑**true**", where "`p`" and "`q`" are fresh variables is about 40 lines of code long and it has the same performance as the manually transformed relation. We omit the transformed program because of the space concerns, but it can be found in the repository[1].

3 RELATIONAL CONVERSION

In this section we describe how the relational conversion in the form of *unnesting* [Byrd 2009] is done. Unnesting constructs a relational program by a first-order functional program.

First, a new variable for every subexpression is introduced with the **let**-expression. Then, all pattern matching and if-expressions are translated into disjunctions, in which unifications are generated for the patterns. Free variables are introduced into scope with the **fresh**. Every n-ary function becomes $(n + 1)$-ary relation with the last argument unified with the result. As a final step, unifications are reordered with relation calls such that to be computed as early as it is possible.

The example of unnesting is shown in Fig. 1. The input functional program is presented in Fig. 1a. The result of introducing fresh variables for subexpressions is in Fig. 1b. The relational program before the conjuncts are reordered is shown in Fig. 1c and the result of the unnesting is presented in Fig. 1d.

[1] https://github.com/Lozov-Petr/miniKanren-2019-Relational-Interpreters-for-Search-Problems

```
let rec append a b =
  match a with
  | []      → b
  | x :: xs →
    x :: append xs b
```

(a)

```
let rec append a b =
  match a with
  | []      → b
  | x :: xs →
    let q = append xs b in
    x :: q
```

(b)

```
let rec appendᵒ a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (a ≡ x :: xs) ∧
     (appendᵒ xs b q) ∧
     (c ≡ x :: q))
```

(c)

```
let rec appendᵒ a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (a ≡ x :: xs) ∧
     (c ≡ x :: q) ∧
     (appendᵒ xs b q))
```

(d)

Fig. 1. Example of unnesting

Note, that the unnesting has limitations: it does not support higher-order functions and partial application. A more general method of translation which does not impose the same limitations was developed [Lozov et al. 2018]. Unfortunately, it uses higher-order relations which are not currently supported in conjunctive partial deduction, so we use unnesting.

The forward execution of the relation mimics the execution of the function from which it was constructed by relational conversion. This makes forward execution quite efficient, to the detriment of the execution in the backwards direction. The unnesting can be modified to improve the performance of backward execution. Let us consider the conversion of a functional conjunction "$f_1\ x_1$ && $f_2\ x_2$".

```
λ res →
  fresh (p) (
    (f₁ x₁ p) ∧
    (conde [
      (p ≡ ↑false ∧ res ≡ ↑false);
      (p ≡ ↑true  ∧ f₂ x₂ res)]))
```

Mimicking the function evaluation, the forward execution of this code first computes "$f_1\ x_1$". If it fails, then the result "res" is unified with "**false**", otherwise the second conjunct "$f_2\ x_2$" is executed and its result is unified with the result. This strategy is not efficient in the backward direction, when we know what "res" is. The following relation is much more performant when executed in the backward direction:

```
λ res →
    conde [
      (res ≡ ↑false ∧ f₁ x₁ ↑false);
      (f₁ x₁ ↑true  ∧ f₂ x₂ res)]
```

In particular, if "res ≡ ↑**true**", both conversions execute "$f_2\ x_2$ res", but when the first conversion computes "$f_1\ x_1$ p" with fresh "p", the second executes "$f_1\ x_1$ ↑**true**". Using the second conversion is enough to significantly

increase the performance in the backward direction. For example, the path search takes several minutes if the first conversion strategy is used, whereas it finishes in less than a second in the second case.

Choosing the second conversion strategy comes with a price for the forward execution. Instead of executing "$\texttt{f}_1\ \texttt{x}_1\ \texttt{p}$", where "p" is fresh, the second strategy executes both "$\texttt{f}_1\ \texttt{x}_1 \uparrow \textbf{false}$" and "$\texttt{f}_1\ \texttt{x}_1 \uparrow \textbf{true}$". In the worst case scenario, when the execution of "$\texttt{f}_1$" does not depend on the last argument, it doubles the number of executions of "$\texttt{f}_1$".

To sum up, by choosing different strategies of the relational conversion we can achieve significant performance improvement. There is no single right way of doing the conversion which improves the performance of the execution in every possible direction. Choosing a strategy per each relation and each direction manually is not feasible, but it can be achieved with a fully-automatic program transformation, such as conjunctive partial deduction.

4 CONJUNCTIVE PARTIAL DEDUCTION

Specialization [Jones et al. 1993] is a natural way to tackle the problem of redundant computations when a part of the input is known. A fully-automatic specialization technique developed in the domain of logic programming is called *partial deduction* [Komorowski 1982; Lloyd and Shepherdson 1991]. It is related to the supercompilation of functional languages [Glück and Sørensen 1994; Turchin 1986]. The particular flavour of the partial deduction we are interested in is called *conjunctive partial deduction* [De Schreye et al. 1999]. As opposed to the partial deduction, conjunctive partial deduction handles conjunctions of atoms, thus being able to perform such optimizations as tupling [Hu et al. 1997] and deforestation [Wadler 1988]. Below we demonstrate by example the features of conjunctive partial deduction.

Deforestation is a program transformation which gets rid of intermediate data structures. The following example demonstrates deforestation. Consider a goal "$\texttt{append}^o\ \texttt{xs ys ts}\ \wedge\ \texttt{append}^o\ \texttt{ts zs rs}$" (note the shared "ts"), where "$\texttt{append}^o\ \texttt{x y xy}$" describes concatenation, "nil ()" constructs the empty list, and "h % t" constructs a new list from the value "h" and another list "t" (similarly to "cons" in Scheme and " :: " in OCaml).

```
let rec appendᵒ x y xy = conde [
  (x ≡ nil () ∧ xy ≡ y);
  (fresh (h t ty) (
     (x  ≡ h % t)  ∧
     (xy ≡ h % ty) ∧
     (appendᵒ t y ty)))]
```

This goal concatenates three lists: "xs", "ys", "zs", constructing an intermediate list "ts". During the execution of this goal, elements of the list "xs" are examined twice: first when "ts" is constructed, and then when the result "rs" is constructed. What is worse, "ts" is only constructed to be immediately deconstructed. Deforestation gets rid of "ts" in this example.

A better program would be such that does not construct "ts" at all. Such a program be generated from the original definition by conjunctive partial deduction and is shown below:

```
let rec doubleAppendᵒ xs ys zs rs = conde [
  (xs ≡ nil () ∧ appendᵒ ys zs rs);
  (fresh (h t ts) (
     (xs ≡ h % t)  ∧
     (rs ≡ h % ts) ∧
     (doubleAppendᵒ t ys zs ts)))]
```

Conjunctive partial deduction is also capable of *tupling*. This transformation makes sure that the same data structure is traversed once even if computing several results. The following example demonstrates such a case.

The goal "`maxLength`^{o} `xs m l`" computes both the maximum value of the list "`xs`" and its length. The elements of the list are Peano numbers with "`zero ()`" as the zero and "`succ`" as the successor function. The third argument "`b`" of the relation "`le`^{o} `x y b`" is "$\uparrow$**true**" if "`x`" is less or equal than "`y`", and "$\uparrow$**false**" otherwise. The relation "`gt`^{o} `x y b`" is similar to "`le`^{o} `x y b`", but it checks for "`x`" to be greater than "`y`".

```
let maxLength^o xs m l = max^o xs m ∧ length^o xs l

let rec length^o xs l = conde [
  (xs ≡ nil () ∧ l ≡ zero ());
  (fresh (h t m) (
    xs ≡ h % t ∧ l ≡ succ m ∧ length^o t m))]

let max^o xs m = max^o_1 xs (zero ()) m

let rec max^o_1 xs n m = conde [
  (xs ≡ nil () ∧ m ≡ n);
  (fresh (h t) (
    (xs ≡ h % t) ∧
    (conde [
      (le^o h n ↑true ∧ max^o_1 t n m);
      (gt^o h n ↑true ∧ max^o_1 t h m)])))]

let rec le^o x y b = conde [
  (x ≡ zero () ∧ b ≡ ↑true);
  (fresh (x1) (
    x ≡ succ x1 ∧ y ≡ zero () ∧ b ≡ ↑false));
  (fresh (x1 y1) (
    x ≡ succ x1 ∧ y ≡ succ y1 ∧ le^o x1 y1 b))]

let rec gt^o x y b = conde [
  (x ≡ zero () ∧ b ≡ ↑false);
  (fresh (x1) (
    x ≡ succ x1 ∧ y ≡ zero () ∧ b ≡ ↑false));
  (fresh (x1 y1) (
    x ≡ succ x1 ∧ y ≡ succ y1 ∧ gt^o x1 y1 b))]
```

Execution of the goal "`maxLength`^{o} `xs m l`" leads to "`xs`" being traversed twice. There is a way to rewrite the program so that "`xs`" is traversed once, but this requires fusing together the definitions of "`length`^{o}" and "`max`^{o}", which either restricts code reuse, or leads to code duplication. A better way is to only fuse the definitions when it is needed, and do it automatically by employing tupling.

The desirable implementation of the "`maxLength`^{o} `xs m l`" relation is the following (the definitions of "`gt`^{o}" and "`le`^{o}" are left out for brevity). It can be achieved with conjunctive partial deduction as well:

```
let maxLength^o xs m l = maxLength^o_1 xs m (zero ()) l

let rec maxLength^o_1 xs m n l = conde [
  (xs ≡ nil () ∧ m ≡ n ∧ l ≡ zero ());
  (fresh (h t l_1)
     (xs ≡ h % t) ∧
     (l ≡ succ l_1) ∧
     (conde [
       (le^o h n ∧ maxLength^o_1 t m n l);
       (gt^o h n ∧ maxLength^o_1 t m h l)]))]
```

4.1 CPD for Prolog-like languages

Initially, conjunctive partial deduction was developed for Prolog-like languages. Conjunctive partial deduction partially evaluates goals, which are conjunctions of atoms, using two levels of control: local and global [Glück et al. 1996]. The global control determines which atoms are to be partially deduced. The local control — what the definitions for the atoms selected at the global control shall be. Both local and global control construct tree structures which represent the input program.

Local control constructs finite SLD-trees for conjunctions of atoms. The construction is guided with an *unfold* operator: it selects a literal from the leaf of the partially constructed SLD-tree and adds its resolvents as children at each step. Since, in general, SLD-trees are infinite, a decision to stop unfolding should be made at some point. There are several techniques for doing this, the most promising of them combine determinacy and either some well-founded or well-quasi order, such as homeomorphic embedding, or other measures.

Global control determines the set of the conjunctions for which partial SLD-trees are built. The important goal of the global control is to ensure termination. The termination is achieved with the *abstraction*. If there is a goal which is embedded into the current goal, it points to the possibility of nontermination. The embedding tells that there is a certain similarity between the two goals, and if a current goal keeps being processed, then their similar subpart will appear again and again, causing nontermination. Whenever the embedding goals are detected, the current goal is abstracted to remove the common subgoal from consideration.

When the partial deduction is done, the only thing left is to construct the *residual program*. The clauses are generated from a partial SLD-tree, one tree per conjunction at the global level. A conjunction is uniquely *renamed* to give a name for the predicate being defined. All free variables of the root of the tree become arguments of the predicate. For each non-failing path in the SLD-tree a clause is generated: a substitution collected along the path is substituted into the head of the clause, and the body is generated from what is in the leaf.

4.2 CPD for miniKanren

In this section we describe how we adapted conjunctive partial deduction for miniKanren. We describe the particular unfolding and generalization strategies as well as discuss how the conjunctive partial deduction had to be modified as a response to the differences between Prolog and miniKanren.

4.2.1 Local Control. Goals in miniKanren are different from those in Prolog-like languages: besides conjunction, disjunction and relation calls, there are explicit unification and introduction of fresh variables. We normalize the input goal so that it was a disjunction of conjunctions of relation calls. To do so, we first pop all the fresh variables to the top level ("`fresh (x) (p (x) ∧ fresh (y) (q(x) ∨ r(y, x)))`" becomes "`fresh (x y) (p(x) ∨ q(x) ∧ r(y, x))`"). Then we transform the goal to be a disjuction of conjunctions of relation calls or unifications. All unifications in each conjunction are evaluated to some substitution (or the

conjunct is discarded, if some unification fails). The normalization allows us to only consider conjunctions of relation calls while doing conjunctive partial deduction.

The local control constructs the following tree structure which represents the goal:

```
type local_tree =
    Fail
  | Success of subst
  | Leaf    of goal list * subst
  | Disj    of local_tree list
  | Conj    of local_tree * goal list
```

Leaf nodes can be either "`Fail`", "`Success`" or "`Leaf`". The "`Fail`" node is created whenever the evaluation of the current goal fails. When the current goal evaluates to some substitution, we create the "`Success`" node with this substitution. The last leaf node is called "`Leaf`", it corresponds to some partially evaluated goal. This type of node contains a substitution which has been computed up to this point, and a residual goal. The goal in this type of node is then examined at the global level.

"`Disj`" node corresponds to a disjunction in a goal: its children are the local control trees constructed for all disjuncts. The last type of nodes is a "`Conj`" node. It is a transient node, which keeps track of a conjunction being unfolded.

In general, unfolding replaces some of the relation calls with their bodies and partially evaluates them. The particular unfolding strategy we adhere to is the following. At each step only one relation call is replaced with its body: the leftmost selectable relation call. The selectable relation call is the one which does not embed any of its predecessors — goals which were unfolded in order to get the current goal. Embedding here is the modification of the homeomorphic embedding defined for the conjunctions of goals in conjunctive partial deduction literature [De Schreye et al. 1999]. Since using pure embedding to control unfolding leads to hideously big programs, we also allow only one non-deterministic unfold.

4.2.2 Global Control. The conjunctions in the "`Leaf`" nodes are processed at the global level. This step is responsible for the termination of the transformation. Generally speaking, the danger for nontermination arises whenever we encounter a subgoal which we have encountered before: processing the same thing will lead to itself over and over again. To break the vicious circle, one needs to stop unfolding the encountered subgoal, this is what *abstraction* serves for.

The simplest case here is when we come upon the goal which is equal up to variable renaming to any other goal at the global level. When this happens, we stop exploring the goal. This is called *variant check* in the literature, and it is done both at the global and the local control levels.

The more complicated case is when a subpart of the goal repeats. This case we test with the modification of the homeomorphic embedding relation (strict homeomorphic embedding), initially developed for conjunctions. A conjunction $\overline{A}$ is considered embedded into a conjunction $\overline{B}$ when there is an ordered subconjunction within $\overline{A}$, each conjunct of which is embedded into the corresponding conjunct of $\overline{B}$:

$$\overline{A} = A_0 \wedge A_1 \wedge \cdots \wedge A_n \trianglelefteq B_0 \wedge B_1 \wedge \cdots \wedge B_m = \overline{B},\ \textit{if}\ \exists\{i_0 \ldots i_m \mid \forall j.i_j < i_{j+1}\} : \forall j \in \{0 \ldots m\}.A_{i_j} \trianglelefteq B_j$$

A single conjunct is embedded into another ($A_i \trianglelefteq B_j$) when the following relation holds and A_i is *not* a strict instance of the second one B_j:

$$X \trianglelefteq Y, \text{where } X \text{ and } Y \text{ are variables}$$

$$f(x_0, x_1, \ldots, x_n) \trianglelefteq f(y_0, y_1, \ldots, y_n) \Leftrightarrow \forall i \in \{0 \ldots n\}.x_i \trianglelefteq y_i$$

$$f \trianglelefteq g(y_0, y_1, \ldots y_m) \Leftrightarrow \exists i \in \{0 \ldots m\}.f \trianglelefteq y_i$$

This check determines two major causes of the growth within the conjunctions. The conjunction can grow in some argument of a relation call or the number of conjuncts itself can grow. To mitigate the first source of the growth, the bigger conjunction can be replaced with a *most specific generalization* of the two conjunctions. Otherwise we need to *split* the embedded subconjunction from the rest and start processing them separately. This process called *abstraction* removes the subconjunctions which cause potential nontermination, and what is left should indeed be processed further.

4.2.3 Residualization. After the transformation is finished, a *residual* program is constructed from the global control tree. A relation definition is generated for each conjunction at the global level (this is done with the renaming step of the original conjunctive partial deduction). First, a unique name is given for each conjunction. Then free variables of the conjunction are collected to become the arguments of the relation: the constructors and constants are omitted (for example "`f x (succ y)` $\wedge$ `g (zero ()) z`" becomes "`fG x y z`". The body of the definition is generated from the local control tree which corresponds to the conjunction under consideration. The body is formed as a disjunction of conjunctions for the non-failure nodes of the local control tree. A computed substitution is transformed into a conjunction of unifications. Suitable definitions are chosen for a goal in a leaf, and the conjunction of their applications is generated. As a final step we perform redundant argument filtering as described in [Leuschel and Sørensen 1996], and introduce fresh variables where necessary.

5 EVALUATION

In this section we present an evaluation of the proposed approach. We have implemented several relational interpreters for different search problems which can be found in the repository mentioned before. Some of the simpler interpreters demonstrate good performance for different directions on their own and for them CPD transformation is not needed. Thus, we will focus on two search problems which are more complex: searching for a path in a graph and searching for a unifier [Baader and Snyder 2001] of two terms. For each problem we compare four programs.

(1) The solver generated by the unnesting alone.
(2) The solver generated by the unnesting strategy aimed at backward execution.
(3) The solver generated by the unnesting and then specialized by conjunctive partial deduction for the backward direction.
(4) The interpretation of the functional verifier with the relational interpreter implemented in Scheme [Byrd et al. 2017].

First, let us compare the performance of the solvers for the path searching problem. The implementation of the functional verifier for this problem is described in Section 2. We ran the search on a graph with 20 nodes and 30 edges, consequentially searching for paths of the length 5, 7, 9, 11, 13, and 15. We averaged the execution times over 10 runs of the same query. We the limited the execution time by 300 seconds, and if the execution time of some query exceeded the timeout, we put ">300s" in the result table and did not request the execution of queries for longer paths. The results are presented in Table 1.

We can conclude that the execution time increases with the length of the path to search, which is expected, since with the length of the path the number of the subpaths to be explored is increasing as well. The solver generated by the unnesting alone and the interpretation with the relational intepreter demonstrate poor performance. The first one is due to its inherently inefficient execution in backward direction, while the second suffers from the interpretation overhead. Both the unnesting aimed at the backward execution and the solver automatically transformed with conjunctive partial deduction show good performance. Conjunctive partial deduction performs more thorough specialization, thus producing more efficient program.

Now let us consider the problem of finding a unifier of two terms which have free variables. A term is either a variable $(X, Y, \dots)$ or some constructor applied to terms $(nil, cons(H, T), \dots)$. A substitution

Path length	5	7	9	11	13	15
Only conversion	0.01s	1.39s	82.13s	>300s	—	—
Backward oriented conversion	0.01s	0.37s	2.68s	2.91s	4.88s	10.63s
Conversion and CPD	0.01s	0.06s	0.34s	2.66s	3.65s	6.22s
Scheme interpreter	0.80s	8.22s	88.14s	191.44s	>300s	—

Table 1. Searching for paths in the graph

Terms	f(X, a) f(a, X)	f(a % b % nil, c % d % nil, L) f(X % XS, YS, X % ZS)	f(X, X, g(Z, t)) f(g(p, L), Y, Y)
Only conversion	0.01s	>300s	>300s
Backward oriented conversion	0.01s	0.11s	2.26s
Conversion and CPD	0.01s	0.07s	0.90s
Scheme interpreter	0.04s	5.15s	>300s

Table 2. Searching for a unifier of two terms

maps a variable to a term. A unifier of two terms t and s is a substitution σ which equalizes them: $t\sigma = s\sigma$ by simultaneously substituting the variables for their images. For example, a unifier of the terms $cons(42, X)$ and $cons(Y, cons(Y, nil))$ is a substitution $\{X \mapsto cons(42, nil), Y \mapsto 42\}$.

We implemented a functional verifier which checks if a substitution equalizes two input terms. We represent a variable name as a unique Peano number. A substitution is represented as a list of terms, in which the index of the term is equal to the variable name to which the term is bound, so the substitution $\{X \mapsto cons(42, nil), Y \mapsto 42\}$ is represented as a list "`[cons (42, nil); 42]`". The verifier returns true if the input terms can be unified with the candidate substitution and false otherwise.

As in the previous problem, we compare four solvers generated for the verifier described. With each solver, we search for a unifier of two terms and compare the execution times. The time comparison is presented in Table 2. The first two rows of each column contain two terms being unified. We use uppercase letters from the end of the alphabet ($X, Y, \ldots$) to denote variables, lowercase letters from the beginning of the alphabet ($a, b, \ldots$) to denote constants (constructors with zero arguments), identifiers which start from the lowercase letter ($f, g, \ldots$) to denote constructors.

Note, we compute a unifier for two terms, but not necessarily the most general unifier. We can implement the most general unification in miniKanren, but achieving the comparable performance using relational verifiers requires additional check that the unifier is indeed the most general. We are currently working on the implementation of such relational verifier.

Here four solvers compare to each other similarly to the previous problem: unnesting demonstrates the worst execution time, relational interpretation in Scheme is a little better, while unnesting aimed at backward execution and conjunctive partial deduction significantly improve the performance.

There exist pairs of terms, for which either of the solvers fails to compute a unifier under 300 seconds. The example of such terms is "`f(A,B,C,A,B,C,D)`" and "`f(B,C,D,x(R,S),x(a,T),x(Q,b),x(a,b))`". This is caused by how general and declarative the verifier is: there is nothing in it to restrict the search space. We can modify the verifier with the additional check to ensure that there are no bound variables in the candidate unifier. This modification restricts the search space when there are many variables in the input terms. But it also changes the semantics of the initial verifier and, as a consequence, the solvers: only idempotent unifiers can be found.

To sum up, we demonstrated by two examples that it is possible to create problem solvers from verifiers by using relational conversion and conjunctive partial deduction. Currently conjunctive partial deduction improves the performance the most, as compared to interpreting verifiers with Scheme relational interpreter or doing relational conversion which is solely aimed at backward or forward execution.

6 CONCLUSION AND FUTURE WORK

We have presented a way to construct a solver for a search problem from its solution verifier by first doing a relational conversion and then specializing the relation according to the desired execution direction by means of conjunctive partial deduction.

There are a few directions for future work.

Even if we generate a relation optimized for the particular direction, executing it with miniKanren still carries some overhead of interpretation. We believe that the best performance can be achieved by generating a functional program from the relation optimized for the particular direction. This way we can avoid interpretation overhead but still get the benefits of the approach. The second direction is to explore other specialization techniques besides conjunctive partial deduction which are better suited for miniKanren programs.

REFERENCES

Franz Baader and Wayne Snyder. 2001. Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, Chapter Unification Theory.

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Indiana University, Bloomington.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages. https://doi.org/10.1145/3110252

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). *Workshop on Scheme and Functional Programming* (2012).

William F. Clocksin and Christopher S. Mellish. 2003. *Programming in Prolog* (5 ed.). Springer, Berlin. https://doi.org/10.1007/978-3-642-55481-0

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.

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

Yoshihiko Futamura. 1971. Partial evaluation of ccomputation process-an approach to a compiler-compiler. *Systems, Compurters, Controls* 25 (1971), 45–50.

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

Robert Glück, Jesper Jørgensen, Bern Martens, and Morten Heine Sørensen. 1996. Controlling conjunctive partial deduction. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 152–166.

Robert Glück and Morten Heine Sørensen. 1994. Partial deduction and driving are equivalent. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 165–181.

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. 1997. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices* 32, 8 (1997), 164–175.

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

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*. ACM, 255–267.

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. *ACM SIGPLAN Workshop on ML* (2016).

Michael Leuschel, Stephen J Craig, Maurice Bruynooghe, and Wim Vanhoof. 2004. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*. Springer, 340–375.

Michael Leuschel and Morten Heine Sørensen. 1996. Redundant argument filtering of logic programs. In *International Workshop on Logic Programming Synthesis and Transformation*. Springer, 83–103.

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

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.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64.

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

Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*. Springer, 344–358.