Improving Refutational Completeness of Relational Search via Divergence Test*

Dmitri Rozplokhas
St. Petersburg Academic University
St. Petersburg, Russia
rozplokhas@gmail.com

Dmitri Boulytchev
St. Petersburg State University
St. Petersburg, Russia
dboulytchev@math.spbu.ru

ABSTRACT

We describe a search optimization technique for implementation of relational programming language miniKanren which makes more queries converge. Specifically, we address the problem of conjunction non-commutativity. Our technique is based on a certain divergence criterion that we use to trigger a dynamic reordering of conjuncts. We present a formal semantics of a miniKanren-like language and prove that our optimization does not compromise already converging programs, thus, being a proper improvement. We also present the prototype implementation of the improved search and demonstrate its application for a number of realistic specifications.

CCS CONCEPTS

• **Theory of computation** → **Constraint and logic programming**; Operational semantics; • **Software and its engineering** → **Constraint and logic languages**;

KEYWORDS

relational programming, refutational completeness, divergence test

ACM Reference Format:
Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test*. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3236950.3236958

1 INTRODUCTION

Relational programming is an approach, based on the idea of describing programs not as functions, but as relations, without distinguishing between the arguments and the result value. This technique makes it possible to "query" programs in various ways, for example, to execute them "backwards", finding all sets of arguments for a given result. Relational behavior can be reproduced using a number of logic programming languages, such as Prolog, Mercury [17], or Curry [8]. There is also a family of embedded DSLs, specifically designed for writing declarative relational programs that originates from miniKanren [7]. miniKanren is a minimalistic declarative language, initially developed for Scheme/Racket. The smallest implementation of miniKanren is reported to comprise of only 40 LOC [9, 11]; there are also more elaborate versions, including miniKanren with constraints [1, 10], user-assisted search [18], nominal unification [5], etc. Due to its simplicity, miniKanren was implemented for more than 50 other languages, such as Haskell, Go, Smalltalk, and OCaml. In a nutshell, miniKanren introduces a minimalistic set of constructs to describe relations over a set of syntactic terms, thus providing the same expressivity as a pure core of conventional logic programming[1].

miniKanren has proven to be a useful tool to provide elegant solutions for various problems, otherwise considered as non-trivial [3]. One of the most promising areas of application for miniKanren is the implementation of *relational interpreters*. Such interpreters are capable not only to interpret programs in various directions, but also to infer programs on the basis of expected input-output specification [6].

Being quite simple and easy to use by design, in implementation miniKanren introduces some subtleties. Under the hood, miniKanren uses a complete interleaving search [13]. This search is guaranteed to find all existing solutions; however, it can diverge, when no solution exists. In reality, this amounts to divergence in a number of important cases — for example, when a program is asked to return *all* existing solutions, or when the number of requested solutions exceeds the number of existing ones. It is often possible to refactor the specification of a concrete query to avoid the divergence, but this has to be done for every execution "direction" of interest that compromises the idea of fully declarative relational programming.

The specifications that do not diverge even when no solutions exist, are called *refutationally complete* [3]. Writing refutationally complete relational specifications nowadays requires knowledge of miniKanren implementation intrinsics, and is not always possible due to the undecidability of the fundamental computability problems. However, by developing a more advanced search it is possible to make more specifications refutationally complete.

In this work we address one particular problem that often leads to refutational incompleteness — the non-commutativity of conjunction. We present an optimization technique that is based on a certain non-termination test. Our optimization is *online* (performed during the search), *non-intrusive* (does not introduce new constructs and does not require any changes to be made to the existing specifications), and *conservative* (applied only when the divergence is

* This work is supported by RFBR grant No 18-01-00380.

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany

ACM ISBN 978-1-4503-6441-6/18/09…$15.00
https://doi.org/10.1145/3236950.3236958

[1] A detailed miniKanren to Prolog comparison can be found at http://minikanren.org/minikanren-and-prolog.html

detected). We prove that for the queries that return a finite number of answers, our optimization preserves convergence. We also demonstrate the application of the optimization for a number of interesting and important problems.

We express our gratitude to William Byrd and the reviwers of this paper for their constructive remarks and suggestions.

2 THE SYNTAX AND SEMANTICS OF A RELATIONAL LANGUAGE

In this section we describe the syntax and semantics of the language that is used in the rest of the paper. To some extent, this description serves as a short introduction to miniKanren. The main distinction between "the real" miniKanren and our version is that we give a proper semantics only to converging programs that deliver a finite set of answers, while in the reality of relational programming the result is represented as an infinite stream, from which any number of answers can be requested, and the request of a non-existing answer can lead to a divergence. Our semantics, thus, corresponds to scenario, when *all* answers are requested from the stream. On the other hand, we do not distinguish programs, calculating the infinite number of answers, from those diverging with no results at all. However, we consider the finite version of the semantics as an important case that is justified by the evaluation, presented in Section 5.

The syntax of our relational language is shown on Fig. 1. First, we introduce the alphabet of constructors $\mathcal{C}$, each of which is equipped with a non-negative arity. Then we in a conventional fashion inductively define the set of all terms $\mathcal{T}(X)$, parameterized by the set of variables X. We need this parameterization since later we will be dealing with two sorts of variables — *syntactic* and *semantic*, and, therefore, two sorts of terms. Next, we choose the set of syntactic variables $\mathcal{V}$ and the set of *relational symbols* $\mathcal{R}$ with arities that will be used as names for relational definitions. We also introduce a shortcut $\mathcal{T}_{\mathcal{V}}$ for the set of all terms over syntactic variables since it will be used in all other syntactic definitions.

The core syntax category in the language is a *goal*. There are five types of goals: unification of two terms, conjunction and disjunction of two goals, fresh variable introduction and a call of some relational definition. We stipulate that the calls of relational definitions respect their arities; we will also use a shortcut form fresh $(x\ y\ z\ \ldots)$ ... instead of fresh(x) (fresh (y) (fresh (z) ...) where needed.

Next, a *relational definition* $\mathcal{D}$ binds some relational symbol to a parameterized goal; the number of parameters corresponds to the arity of the symbol, and we assume that all parameter variables are pairwise distinct. Finally, the top-level syntax category is a *specification* $\mathcal{S}$ — a goal in the context of some relational definitions.

As an example of a relational program we consider a canonical specification — list concatenation relation append^o:

```
append^o  ↦  λ x y xy .
  ((x ≡ Nil) ∧ (xy ≡ y)) ∨
  (fresh (h t ty)
     (x ≡ Cons (h, t)) ∧
     (append^o t y ty) ∧
     (xy ≡ Cons (h, ty)))
```

We respect here the convention to add the "^{o}" suffix to all names of relational entities; for the simplicity, we omit the arities of constructors `A`, `Cons`, `Nil` and relational symbol append^o.

Note, we define here a language with first-order relations; in particular, we do not allow partial application. As we see later, our approach critically depends on recursive call identification that is a trivial task in the first-order case. Some existing frameworks for relational programming [14, 15] do not impose such a limitation; extending our approach for a higher-order case is a subject of future research.

We describe the semantics of our language using a conventional big-step style inference system. First, we choose an infinite set of *semantic variables* $\mathcal{W}$. As we will see shortly, the fresh(x)... construct allocates a fresh variable, not being used before, and associates it with the syntactic variable x. Thus, in the semantics we will need an infinite supply of fresh variables.

Next, we introduce the *interpretation* of syntactic variables ι as a (partial) mapping

$$\iota : \mathcal{V} \to \mathcal{T}(\mathcal{W})$$

The role of the interpretation is twofold: first, it binds syntactic variables, used in the fresh construct, to their semantic counterparts, and second, it binds relational parameters to their values — terms over semantic variables. For a syntactic term t and an interpretation ι we denote $t\iota$ the result of substitution of all syntactic variables in t by their interpretations according to ι; we assume $t\iota$ to be defined only when ι is defined for all variables in t. Thus, $t\iota$, if defined, is always an element of $\mathcal{T}(\mathcal{W})$.

Then, we borrow some conventional machinery from unification theory [2]. Namely, we define a substitution σ to be a partial mapping between semantic variables and semantic terms:

$$\sigma : \mathcal{W} \to \mathcal{T}(\mathcal{W})$$

For any substitution σ we assume that the set of all free variables of all terms in the range of σ has an empty intersection with the domain of σ, and we denote by $\sigma \circ \theta$ the composition of substitutions, defined in a usual way. For arbitrary $t \in \mathcal{T}(\mathcal{W})$ and a substitution σ we denote the result of application of σ to t as $t\sigma$.

The basic inference relation for our semantics has the form

$$\Gamma, \iota \vdash (\sigma, \delta) \stackrel{g}{\Longrightarrow} S$$

where Γ is an environment that binds relational symbols to their definitions, ι — an interpretation, σ — a substitution, δ — a set of allocated semantic variables, g — a goal, and S — a set of pairs (σ', δ'), where σ' and δ' — a substitution and a set of allocated semantic variables respectively. Informally speaking, we interpret a goal g in the context of relational definitions Γ, current interpretation ι, current substitution σ and current set of allocated semantic variables σ. As a result, we obtain a (possibly empty) set of answers. Each answer consists of a new substitution, accumulated through the execution of g, and a new set of allocated semantic variables (note, in the original miniKanren a goal can produce the same answer multiple number of times, but this property is not important in our case).

$$
\begin{array}{rcll}
\mathcal{C} & = & \{C^k, \dots\} & \textit{(constructors)} \\
\mathcal{T}(X) & = & x \in X \mid C^k\ (t_1, \dots, t_k),\ t_i \in \mathcal{T}(X) & \textit{(terms)} \\
\mathcal{V} & = & \{x, y, z, \dots\} & \textit{(syntactic variables)} \\
\mathcal{T}_{\mathcal{V}} & = & \mathcal{T}(\mathcal{V}) & \textit{(syntactic terms)} \\
\mathcal{R} & = & \{r^k, \dots\} & \textit{(relational symbols)} \\
\mathcal{G} & = & t_1 \equiv t_2,\ t_i \in \mathcal{T}_{\mathcal{V}} & \textit{(unification)} \\
 & & g_1 \wedge g_2 & \textit{(conjunction)} \\
 & & g_1 \vee g_2 & \textit{(disjunction)} \\
 & & \underline{\texttt{fresh}}\ (x)\ g & \textit{(fresh variable introduction)} \\
 & & r^k\ t_1 \dots t_k,\ t_i \in \mathcal{T}_{\mathcal{V}} & \textit{(relational reference)} \\
\mathcal{D} & = & r^k\ \mapsto\ \lambda x_1 \dots x_k\,.\,g,\ x_i \in \mathcal{V} & \textit{(relational definition)} \\
\mathcal{S} & = & d_1, \dots, d_k; g & \textit{(specification)}
\end{array}
$$

Figure 1: The syntax of the source language

$$\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{t_1 \equiv t_2} \varnothing,\ mgu\ (t_1\iota\sigma,\ t_2\iota\sigma) = \bot \qquad [\textsc{UnifyFail}]$$

$$\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{t_1 \equiv t_2} \{(\sigma \circ \Delta,\ \delta)\},\ mgu\ (t_1\iota\sigma,\ t_2\iota\sigma) = \Delta \neq \bot \qquad [\textsc{UnifySuccess}]$$

$$\frac{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{g_1} S_1, \quad \Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{g_2} S_2}{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{g_1 \vee g_2} S_1 \cup S_2} \qquad [\textsc{Disj}]$$

$$\frac{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{g_1} \bigcup_i \{(\sigma_i, \delta_i)\}, \quad \forall i\ :\ \Gamma, \iota \vdash (\sigma_i, \delta_i) \xRightarrow{g_2} S_i}{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{g_1 \wedge g_2} \bigcup_i S_i} \qquad [\textsc{Conj}]$$

$$\frac{\Gamma, \iota[x \leftarrow \alpha] \vdash (\sigma, \delta \cup \{\alpha\}) \xRightarrow{g} S}{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{\underline{\texttt{fresh}}(x)\ g} S},\ \alpha \in \mathcal{W} \setminus \delta \qquad [\textsc{Fresh}]$$

$$\frac{\Gamma, [x_i \leftarrow v_i] \vdash (\epsilon, \delta) \xRightarrow{g} \bigcup_j \{(\sigma_j, \delta_j)\}}{\Gamma, \iota \vdash (\sigma, \delta) \xRightarrow{r^k\ t_1 \dots t_k} \bigcup_j \{(\sigma \circ \sigma_j, \delta_j)\}},\ \Gamma(r^k) = \lambda x_1 \dots x_k.g,\ v_i = t_i\iota\sigma \qquad [\textsc{Invoke}]$$

Figure 2: Big-step operational semantics

The inference rules themselves are shown on Fig. 2. The first two rules handle two possible outcomes of the unification. Note, we use here the most general unifier (*mgu*) of two semantic terms; we assume "occurs check" to be incorporated in the unification algorithm. Since the unification goal is built of syntactic terms, we have to interpret them first (by applying ι), and take into account current substitution σ.

The rule for the disjunction first interprets the constituents of the disjunction in the same state and then combines the outcomes.

The rule for the conjunction threads the execution of its subgoals in a left-to-right successive manner: first the left conjunct is evaluated, providing a set $\{(\sigma_i, \delta_i)\}$. Then the second conjunct is evaluated for each element of the set, and the results are eventually combined. Note, the evaluation of both conjuncts is performed under the *same* interpretation ι since both of them occur in the *same* bounding context. The substitution and the set of allocated semantic variables, on the other hand, are inherited from left to right since the evaluation of the right conjunct has to be performed in the context of the results, provided by the left one.

The rule for the $\underline{\texttt{fresh}}$ construct allocates arbitrary semantic variable, not taken before, and evaluates the single subgoal in the updated interpretation that associates the syntactic variable, bound in this $\underline{\texttt{fresh}}$, with the chosen semantic one.

Finally, the rule for relational definition invocation describes its evaluation in a few steps. First, the body of the definition is found, using the environment Γ. Then, the terms t_i, specified as the arguments of the invocation, are converted into their semantic forms v_i using current interpretation ι and current substitution σ. Next, the body of the definition is evaluated in the context of *empty* substitution ϵ and an interpretation, containing nothing else, than the bindings for the formal parameters of the definition. This way of handling interpretation models the behavior of a call stack in conventional languages with no nested functions. Finally, the result substitutions are composed with the original one[2].

[2]We could use the original substitution instead of the empty one without the need to use composition; however we found the approach we took more proof-friendly since each relational definition is evaluated in initially empty substitution.

Given this big-step evaluation relation for goals, we can describe the evaluation for the top-level specification $s = d_1, \ldots, d_k; g$. First, we construct the associated environment Γ_s that properly binds all relational symbols in s to their bodies. Then, we evaluate the top-level goal

$$\Gamma_s,\ \bot \vdash (\epsilon,\ \varnothing) \overset{g}{\Longrightarrow} S_s$$

obtaining the set of results S_s; here we use the empty (everywhere undefined) interpretation $\bot$ and the empty substitution ϵ as a starting point. Finally, we choose all substitutions from S_s.

Our semantics is almost deterministic — the only source of ambiguity is the rule for the fresh construct, where we choose a new semantic variable arbitrarily. If we fix the order, in which semantic variables are allocated, the semantics becomes completely deterministic. It is also easy to see that if each relational symbol is unambiguously defined in the specification and called with a proper number of parameters, and all goals in all relational definitions and the top-level goal are closed (i.e. each variable occurrence is bound either in some fresh construct or in a parameter list of enclosing definition), then during the evaluation all syntactic variables are properly interpreted — in other words, the execution cannot break down halfway through and either diverges or finishes with some results.

We illustrate the evaluation, determined by this semantics, by the following query:

```
fresh (q) (append^o (Cons (A, Nil)) Nil q)
```

where append^o is a list concatenation relation, presented earlier in this section. Since we require the top-level goal to be closed, from now on we conventionalize the use of the top-level fresh construct as a binder for the variables, whose values we are interested in (in this particular example q).

The evaluation is illustrated on Fig. 3. Note, the derivation is shown in a top-down manner, opposite to the direction, prescribed by the inference rules. We use numbers in bold font to denote semantic variables. For the sake of brevity and in order to make the illustration observable we do not show the binding environment for relational definitions and as a rule denote by ellipses all inherited components of derivation tree (the components in the left side of "$\Rightarrow$" are inherited top-down, in the right side — bottom-up).

We start from the top-level goal and first apply the rule Fresh (see Fig. 3a). Since we did not use any semantic variables yet, we allocate the first one (**0**), update the interpretation and the set of used semantic variables and continue. The next construct is the call for append^o, so we unfold its definition, replace the interpretation of syntactic variables by the bindings for the formal parameters, and evaluate the body w.r.t. the empty substitution (which has no difference from the current one yet). The body of append^o definition is a disjunction, so we take its left constituent, which is a conjunction, so we, in turn, take its left constituent, which is a unification $x \equiv$ Nil. This unification clearly fails, as current interpretation binds x to Cons (A, Nil). This completes the whole branch for the first disjunct of append^o with empty result.

The evaluation of the second disjunct is shown on Fig. 3b. Its top-level construct is fresh (h t ty), so we allocate three successive semantic variables **1**, **2** and **3** and save the bindings in the interpretation. The next construct is a conjunction of three goals (assuming "$\wedge$" is right-associative in the concrete syntax), and we proceed with the first one, which is a unification $x \equiv$ Cons(h, t). Since x, h and t are free in the current substitution and x is bound to Cons(A, Nil) by current interpretation, the unification succeeds with the substitution [**1**↦A, **2**↦Nil]. The evaluation of remaining conjuncts is shown on Fig. 3c.

The first one is a recursive call to append^o. We evaluate the actual parameters — t, y and ty — in the current interpretation and substitution, obtaining Nil, Nil and **3** respectively, replace the interpretation to bind formal parameters to these values, and recurse to the body with empty current substitution. Again, we have the disjunction, and the first disjunct is a conjunction $(x \equiv \text{Nil}) \wedge (xy \equiv y)$. Now $x \equiv$ Nil succeeds, as x is already bound to Nil by the interpretation, and $xy \equiv y$ succeeds as well, providing a new substitution [**3**↦Nil]. We omit the detailed evaluation of the second top-level disjunct of append^o since it contains a unification $x \equiv$ Cons (_, _) which, clearly, does not contribute anything.

Finally, we return from the recursive call to append^o and take the composition of substitutions — the one before the call, and another after — which gives us [**1**↦A, **2**↦Nil, **3**↦Nil] (see Fig. 3d). We only need now to interpret the last conjunct of the second disjunct of append^o — $xy \equiv$ Cons (h, ty) — which gives us the final substitution [**1**↦A, **2**↦Nil, **3**↦Nil, **0**↦Cons (A, Nil)]. Now, we have to remember that the topmost bound variable of the top-level goal is q, and corresponding semantic variable is **0**. Thus, the answer is q = Cons (A, Nil), which is rather expected.

3 REFUTATIONAL INCOMPLETENESS AND CONJUNCTION NON-COMMUTATIVITY

The language, defined in the previous section, is expected to allow defining computable relations in a very concise and declarative form. In particular, it is expected from a relational specification to preserve its behavior regardless the order of conjunction/disjunction constituents. Regretfully, in general this is not true, and one of the most important manifestations of this deficiency is *refutational incompleteness*.

In the context of relational programming, refutational completeness [3] is understood as a capability of a program to discover the absence of solutions and stop. At the first glance, the divergence in the case of solution absence does not seem to be a severe problem. However, as we see shortly, refutational incompleteness leads to many observable negative effects in numerous practically important cases.

We demonstrate the effect of refutational incompleteness with a very simple example. Let us take the definition of append^o from the previous section and try to evaluate the following query:

```
fresh (p q) (append^o p q Nil)
```

We would expect this query to converge to the single answer p = Nil, q = Nil; however, in the reality the query diverges. We sketch here the explanation, omitting some non-essential technical details, such as semantic variables allocation, etc.:

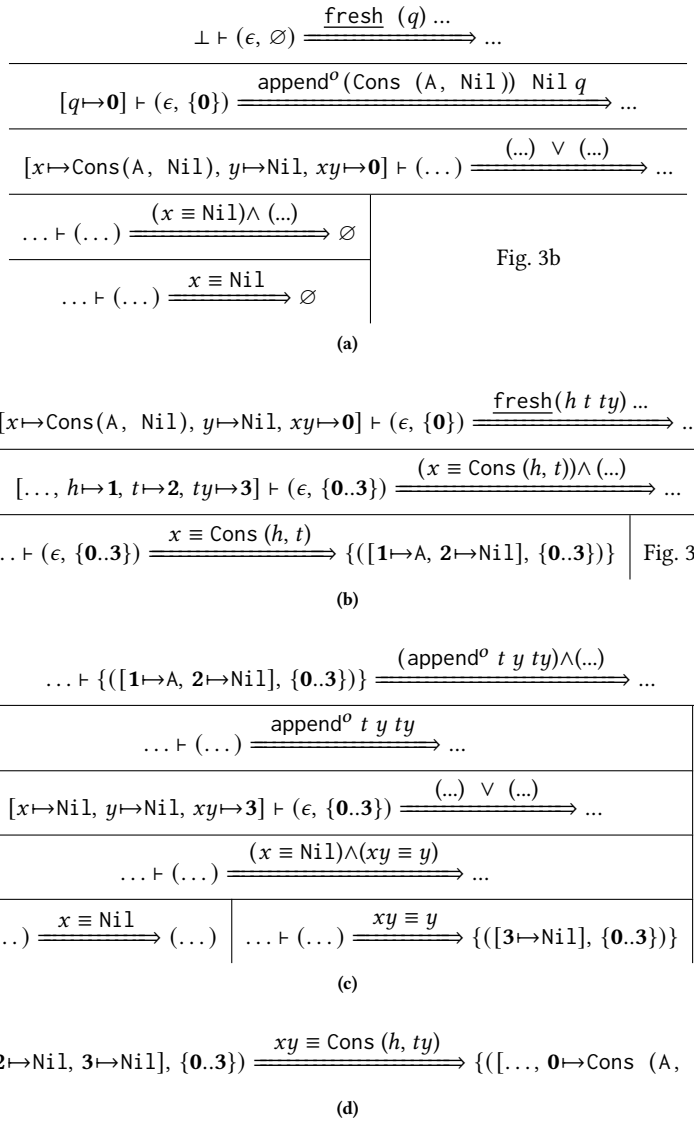

Figure 3: An example of relational evaluation

- First we evaluate the first disjunct of append^o's body and unify p with `Nil` (successfully) and `Nil` with q (successfully), which gives us the first (expected) answer.
- Then we proceed to the second disjunct, which is a conjunction of three simpler goals:
  - in the first one we unify p with `Cons (h, t)` (successfully);
  - in the second we encounter a recursive call append^o t q `Nil`; since its arguments are merely the renamings of the enclosing one, we repeat from the top and never stop.

The problem is that the semantics of conjunction, in fact, is not commutative: when the first conjunct diverges and the second fails, the whole conjunction diverges. We stress that this is not a deviation of our semantics, but a well-known phenomenon, manifesting itself in all known miniKanren implementations. In our example, switching two last conjuncts in the definition of append^o solves the problem — now the whole search stops after the unsuccessful attempt to unify `Nil` and `Cons (h, ty)` with no recursive call. This, improved version of append^o, is known to be refutationally complete. In fact, there is a conventional "rule of thumb" for miniKanren programming to place the recursive call as far right as possible in a list of conjuncts.

This convention, however, does not always help; to tell the truth, it often makes the things worse. Consider as an example yet another relation on lists:

```
revers^o ↦ λ x x_r .
  ((x ≡ Nil) ∧ (x_r ≡ Nil)) ∨
```

```
(fresh (h t t_r)
   (x ≡ Cons (h, t)) ∧
   (append^o t_r (Cons (h, Nil)) x_r) ∧
   (revers^o t t_r)
)
```

This relation corresponds to a relational list reversing; as we see, the recursive call is placed to the end. However, the following query

```
fresh (q) (revers^o (Cons (A, Nil)) q)
```

diverges, while

```
fresh (q) (revers^o q (Cons (A, Nil)))
```

converges to the expected results. If we switch the two last conjuncts in the definition of $\texttt{revers}^o$, the situation reverses: the first query converges, while the second diverges. This example demonstrates that the desired position of a recursive call (and, in general, the order of conjuncts) depends on the direction, in which the relation of interest is evaluated.

There are, however, some cases, when the same relation is evaluated in both directions, regardless the query. We can take as an example relational permutations, which can be implemented by running relational list sorting in both directions:

```
sort^o ↦ λ x x_s . ...
perm^o ↦ λ x x_p .
  fresh (x_s)
    (sort^o x x_s) ∧ (sort^o x_p x_s)
```

The idea of this implementation is very simple. Let us want to calculate all permutations of some list l. We first sort l, obtaining the sorted version l'; then we ask for all lists which, being sorted, become equal to l'. Obviously, all such lists are merely permutations of the original list l. The important observation is that the existence of a single list sorting relation is sufficient to implement this idea.

The concrete definition of the relational list sorting $\texttt{sort}^o$ is not important, so we omit it due to the space considerations (an interested reader can refer to [14]). The important part is that it is obviously recursive and not refutationally complete, and it is being evaluated in *both* directions within the body of $\texttt{perm}^o$. So, $\texttt{perm}^o$ is expected to perform poorly regardless the order of recursive calls in $\texttt{sort}^o$ implementation; it, indeed, does. First, if we request all solutions, both `fresh (q) (perm^o l q)` and `fresh (q) (perm^o q l)` diverge for arbitrary non-empty list `l` regardless the implementation of $\texttt{sort}^o$; second, even if we request only a first few existing solutions, it does not provide any results in a reasonable time even for very small list lengths (4, 5, etc.).

Interesting, that if we interested in all solutions, we have to accurately precompute their number in order not to request more, than exists. For some problems, it may be not so simple, as it looks at a first glance (for example, the number of all permutations is not a factorial, but a number of permutations with repetitions). Finally, getting the number of solutions can itself be an objective for writing a relational specification (we provide some examples in Section 5), and without refutational completeness requesting all solutions to calculate their number is out of reach.

4 SEARCH IMPROVEMENT

As we've seen in the previous section, the non-commutativity of conjunction in the presence of recursion is one of the reasons for refutational incompleteness. Switching arguments of a certain conjunction can sometimes improve the results; there is, however, no certain static order, beneficial in all cases. Thus, we can make the following observations:

- the conjunction to change has to be properly identified;
- the order of conjunct evaluation has to be a subject of a *dynamic* choice.

Our improvement of the search is based on the idea of switching the order of conjuncts only when the divergence of the first one is detected. More specifically:

- during the search, we keep track of all conjunctions being performed;
- when we detect the divergence, we roll back to the nearest conjunction, for which we did not try all orders of constituents yet, switch its constituents, and rerun the search from that conjunction.

The important detail is the divergence test. Of course, due to the fundamental results in computability theory, there is no hope to find a *precise* computable test that constitutes the necessary and sufficient condition of divergence. However, in our case a sufficient condition is sufficient. Indeed, a sufficient condition identifies a case, when the search, being continued, will lead to an incompleteness (since a divergence in our semantics always means incompleteness). Thus, it is no harm to try some other way.

Another important question is the discipline of conjuncts reordering. Indeed, simply switching any two operands of, for example, $(g_1 \wedge g_2) \wedge g_3$, would not allow us to try $(g_1 \wedge g_3) \wedge g_2$. Thus, we have to flatten each "cluster" of nested conjunctions into a list of conjuncts $\bigwedge g_i$, where none of the goals g_i is a conjunction. Then, it may seem at the first glance that the number of orderings to try is exponential on the number of conjuncts; we are going to show that, fortunately, this is not the case, and a quadratic number of orders is sufficient.

In the rest of the section we address all these issues in details: first, we formally present the divergence criterion and prove the necessity property; then, we describe an efficient reordering discipline. Finally, we present a modified version of the semantics with incorporated divergence test and reordering. This semantics can be considered as a modified version of the search, and we prove that this modification is a proper improvement in terms of convergence.

4.1 The Divergence Test

Our divergence test is based on the following notion:

Definition 1. We say that a vector of terms $\overline{a_i}$ is more general, than a vector of terms $\overline{b_i}$ (notation $\overline{a_i} \geq \overline{b_i}$), if there is a substitution τ, such that $\forall i\ b_i = a_i\tau$.

The idea of the divergence test is rather simple: it identifies a recursive call with more general arguments than (some) enclosing one. To state it formally and prove it using the semantics from section 2, we need several definitions and lemmas.

Definition 2. A semantic variable v is *observable* w.r.t. the interpretation ι and substitution σ, if there exists a syntactic variable x, such that $v \in FV(\iota(x)\sigma)$.

Definition 3. A triplet of interpretation, substitution and a set of allocated semantic variables (ι, σ, δ) is called *coherent*, if $dom(\sigma) \subseteq \delta$, and any semantic variable, observable w.r.t. ι and σ, belongs to δ.

Definition 4. A semantic statement

$$\Gamma, \iota \vdash (\sigma,\ \delta) \xRightarrow{g} S$$

is *well-formed*, if (ι, σ, δ) is a coherent triplet.

Note, the root semantic statement $\Gamma, \bot \vdash (\epsilon,\ \varnothing) \xRightarrow{g} S$ is always well-formed.

Lemma 1. For a well-formed semantic statement, every statement in its derivation tree is also well-formed.

The proof is by induction on the derivation tree. Note, we need to generalize the statement of the lemma, adding the condition that $(\iota, \sigma_r, \delta_r)$ is a coherent triplet for any $(\sigma_r,\ \delta_r) \in S$.

The next lemma ensures that any substitution in the RHS of a semantic statement is a correct refinement of that in the LHS:

Lemma 2. For a well-formed semantic statement

$$\Gamma, \iota \vdash (\sigma,\ \delta) \xRightarrow{g} S$$

and any result $(\sigma_r,\ \delta_r) \in S$, there exists a substitution Δ, such that:

(1) $\sigma_r = \sigma \circ \Delta$;
(2) any semantic variable $v \in dom(\Delta) \cup ran(\Delta)$ either is observable w.r.t. ι and σ, or does not belong to δ (where $ran(\Delta) = \bigcup_{v \in dom(\Delta)} FV(\Delta(v))$).

The proof is by induction on the derivation tree; we as well need to generalize the statement of the lemma, adding the condition that the set of all allocated semantic variables δ can only grow during the evaluation.

The final lemma formalizes the intuitive considerations that the evaluation for a certain state (σ', δ') cannot diverge, if the evaluation for a more general state (σ, δ) doesn't diverge:

Lemma 3. Let

$$\Gamma, \iota \vdash (\sigma,\ \delta) \xRightarrow{g} S$$

be a well-formed semantic statement, $(\iota', \sigma', \delta')$ be a coherent triplet, and let τ be a substitution, such that $\iota'(x)\sigma' = \iota(x)\sigma\tau$ for any syntactic variable x. Then

$$\Gamma, \iota' \vdash (\sigma',\ \delta') \xRightarrow{g} S'$$

is well-formed and its derivation height is not greater than that for the first statement.

The proof is by induction on the derivation tree for the first statement. We need to generalize the statement of the lemma, adding the requirement that for any substitution s'_r in the RHS of the second statement, there has to be a substitution s_r in the RHS of the first statement, such that there exists a substitution τ_r, such that $\iota'(x)\sigma'_r = \iota(x)\sigma_r\tau_r$ for any syntactic variable x. In the cases of Fresh and Invoke rules, some semantic variables can become non-observable, and we need to define a substitution τ_r separately for these "forgotten" variables and those, which remain observable, using Lemma 2.

Now we are ready to claim and prove the divergence criterion.

Theorem 1 (Divergence criterion). For any well-formed semantic statement

$$\Gamma, \iota \vdash (\sigma,\ \delta) \xRightarrow{r^k\ t_1 \dots t_k} S$$

if its proper derivation subtree has a semantic statement

$$\Gamma, \iota' \vdash (\sigma',\ \delta') \xRightarrow{r^k\ t'_1 \dots t'_k} S'$$

then $\overline{t'_i \iota' \sigma'} \not\geq \overline{t_i \iota \sigma}$.

Proof. Assume that $\overline{t'_i \iota' \sigma'} \geq \overline{t_i \iota \sigma}$.
By Lemma 1, the semantic statement

$$\Gamma, \iota' \vdash (\sigma',\ \delta') \xRightarrow{r^k\ t'_1 \dots t'_k} S'$$

is well-formed.
By Lemma 3, the derivation tree for

$$\Gamma, \iota' \vdash (\sigma',\ \delta') \xRightarrow{r^k\ t'_1 \dots t'_k} S'$$

has greater or equal height than that for

$$\Gamma, \iota \vdash (\sigma,\ \delta) \xRightarrow{r^k\ t_1 \dots t_k} S$$

which contradicts the theorem condition.

□

The theorem justifies that, indeed, our test constitutes a sufficient condition for a divergence: if the execution reaches a relation call with more general arguments, than those of some enclosing one, then it has no derivation in our semantics, and, thus, it is not terminating.

4.2 Conjuncts Reordering

In this section we consider the discipline of conjuncts reordering. Recall, we flatten all nested conjunctions in clusters $\wedge g_i$, where none of g_i is a conjunction. To evaluate a cluster, we have to evaluate its conjuncts one after another, threading the results, starting from the initial substitution. Each time we evaluate a conjunct, we can have three possible outcomes:

- The evaluation converges with some result. In this case, we can proceed with the next conjunct.
- The evaluation diverges undetected. In this case, nothing can be done.
- A divergence is detected by the test. This is the case when the reordering takes place.

$$\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{t_1 \equiv t_2} \varnothing,\ mgu\,(t_1\iota\sigma,\ t_2\iota\sigma) = \bot \qquad [\textsc{UnifyFail}^+]$$

$$\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{t_1 \equiv t_2} (\sigma \circ \Delta,\ \delta),\ mgu\,(t_1\iota\sigma,\ t_2\iota\sigma) = \Delta \neq \bot \qquad [\textsc{UnifySuccess}^+]$$

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_1} S_1;\quad \Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_2} S_2}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_1 \vee g_2} S_1 \cup S_2} \qquad [\textsc{Disj}^+]$$

$$\frac{\Gamma,\ \iota[x \leftarrow \alpha],\ h \vdash (\sigma,\ \delta \cup \{\alpha\}) \xRightarrow[e]{g} S^\dagger}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{\underline{\mathtt{fresh}}(x)\,g} S^\dagger},\ \alpha \in \mathcal{W} \setminus \delta \qquad [\textsc{Fresh}^+]$$

Figure 4: Improved search: inherited rules

$$\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{r^k t_1 \ldots t_k} \dagger,\ v_i = t_i\iota\sigma,\ (v_1, \ldots, v_k) \geq h\,r^k \qquad [\textsc{InvokeDiv}^+]$$

$$\frac{\Gamma,\ \epsilon[x_i \leftarrow v_i],\ h[r^k \leftarrow (v_1, \ldots, v_k)] \vdash (\epsilon,\ \delta) \xRightarrow[e]{g} \bigcup_j \{(\sigma_j,\ \delta_j)\}}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{r^k t_1 \ldots t_k} \bigcup_j \{(\sigma \circ \sigma_j, \delta_j)\}},\ v_i = t_i\iota\sigma,\ \Gamma\,r^k = \lambda x_1 \ldots x_k.g,\ (v_1, \ldots, v_k) \not\geq h\,r^k \qquad [\textsc{Invoke}^+]$$

Figure 5: Improved search: invocation and divergence detection

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_1} \dagger}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_1 \vee g_2} \dagger} \qquad [\textsc{DivDisjLeft}^+]$$

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_2} \dagger}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{g_1 \vee g_2} \dagger} \qquad [\textsc{DivDisjRight}^+]$$

$$\frac{\Gamma,\ \epsilon[x_i \leftarrow v_i],\ h[r^k \leftarrow (v_1, \ldots, v_k)] \vdash (\epsilon,\ \delta) \xRightarrow[e]{g} \dagger}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \xRightarrow[e]{r^k t_1 \ldots t_k} \dagger},\ v_i = t_i\iota\sigma,\ \Gamma\,r^k = \lambda x_1 \ldots x_k.g,\ (v_1, \ldots, v_k) \not\geq h\,r^k \qquad [\textsc{DivInvoke}^+]$$

Figure 6: Improved search: divergence propagation

In a general case, for each cluster there can be some converging prefix ω we've managed to evaluate so far (initially empty), and the rest of the conjuncts g_i. Since ω converges, we have some set of substitutions S_ω that corresponds to the result of ω evaluation.

Suppose none of g_i converges on S_ω (i.e. for each g_i there is at least one substitution in S_ω, on which g_i diverges). We claim that reordering conjuncts inside ω would not help. Indeed, with any other order of conjuncts, ω either diverges or converges with the same result (up to the renaming of semantic variables). Thus, making any permutations inside ω is superfluous.

Next, suppose we have two different goals g_1 and g_2, which both converge on S_ω (i.e. both converge on each substitution in S_ω). Do we need to try both cases (g_1 and g_2) to extend the converging prefix? It is rather easy to see that if, say, g_2 converges on S_ω, then it will as well converge on the result of evaluation of g_1 on S_ω. Indeed, for arbitrary $(\sigma, \delta) \in S_\omega$ we have

$$\ldots \vdash (\sigma, \delta) \xRightarrow[e]{g_1} S'_\omega$$

where each σ' (such that $(\sigma', \delta') \in S'_\omega$) is a "more specific", than σ, by Lemma 2. By Lemma 3, since g_2 converges on $(\sigma, \delta) \in S_\omega$, it converges on each $(\sigma', \delta') \in S'_\omega$ as well.

In other words, to extend a converging prefix we can choose arbitrary conjunct, which converges immediately after this prefix, and this choice will never have to be undone.

Now we can specify the reordering discipline. Since we never re-evaluate a converging prefix, we do not represent it. Thus, each cluster we consider from now on is a suffix of some initial cluster after evaluation of some converging prefix (and, perhaps, after some reorderings performed so far).

Let us have a cluster $\bigwedge_{i=1}^{k} g_i$. We evaluate it on some substitution σ in the context of some integer value p (initially $p = 1$), which describes, which conjunct we have to try next. We operate as follows:

(1) We try to evaluate g_p on σ. If the evaluation succeeds with a result S', we remove g_p from the cluster and evaluate the rest for each substitution in S' and $p = 1$.
(2) If a divergence is detected, and $p \leq k$, then increment p, and repeat from step 1 (which will try the next goal).
(3) Otherwise, we give up and rollback to the enclosing cluster (if any).

Thus, we apply a greedy approach: each time we have a converging prefix of conjuncts (possibly empty), and some tail. We try to put each conjunct from the tail immediately after the prefix. If we find a converging conjunct, we attach it to the prefix and continue; if no, then the list of conjuncts diverges. Thus, we can find a converging order (if any) in a quadratic time. Note, for different substitutions in the result of a converging prefix evaluation the order of remaining conjuncts can be different.

4.3 Improved Search Semantics

Here we combine all observations, presented in the preceding subsections — the divergence test, conjunct clustering and reordering, — and express the improved search in terms of a big-step operational semantics that is an extension of the initial one, presented in Section 2.

We denote "$\underset{e}{\Rightarrow}$" the semantic relation for the improved search, and we add another component to the environment — a history h, — which maps a relational symbol to a list of fully interpreted terms as its arguments. As we are (sometimes) capable of detecting the divergence, besides a regular set of answers S as a result of evaluation we can have a divergence signal, which we denote †; $S^\dagger$ ranges over both the set of answers S and the divergence signal †.

For the convenience of presentation we split the set of semantic rules into a few groups. The first one is the inherited rules (see Fig. 4) — those, which did not change (except for the extension in the environment and evaluation result). Note, the rule Disj$^+$ does not handle the divergence detection in either of disjuncts.

The next group describes the invocation and divergence detection (see Fig. 5). On relation invocation, we first consult with the history. If the history indicates that the invocation is performed in the context of the same relation evaluation with more specific arguments, then we raise the divergence signal; otherwise we perform normally. Note, the rule Invoke$^+$ does not handle the divergence in the *body* of invoked relation.

The next group describes the divergence signal propagation (see Fig. 6). Here the divergence signal, raised in one of the disjuncts or in the body of relational definition, is propagated to the upper levels of the derivation tree.

The final group handles the conjunct reordering (see Fig. 7). As we need a reordering parameter p (see Section 4.2), we introduce another relation "$\underset{r}{\Rightarrow}$" with environment, enriched by p.

The rule ClusterStart$^+$ describes the case, when we make an attempt to evaluate a cluster. It can happen, when we either first encounter an original cluster or try to evaluate a suffix of some initial cluster past some converging prefix. As the reordering starts now, we recurse to the reordering relation with the parameter $p = 1$ (which means, that the first conjunct will be tried to evaluate next).

Two next rules describe the case, when the p-th conjunct, being tried to evaluate, succeeds with some result. In the rule ClusterStep$^+$ we handle the case, when all other conjuncts can be evaluated in the context of that result: we combine the outcomes, which completes the evaluation of the whole cluster. In the rule ClusterDiv$^+$ we consider the opposite case: now there is some conjunct g_j, which raises a divergence signal, being evaluated in the context of the results, delivered by the evaluation of g_p. As we argued in Section 4.2, nothing can be done, and we propagate the divergence signal.

The rule ClusterNext$^+$ describes the case, when the p-th conjunct raises the divergence signal, and there are some other conjuncts to try. We increment p and proceed.

Finally, in the rule ClusterStop$^+$ we handle the situation, when all available conjuncts in a cluster were tried to evaluate first and raised the divergence signal. We propagate the signal in this case.

The following theorem is rather easy to prove:

Theorem 2. *For arbitrary* Γ *and* g *if*

$$\Gamma,\ \bot \vdash (\epsilon,\ \varnothing) \overset{g}{\Longrightarrow} S$$

then

$$\Gamma,\ \bot,\ \bot \vdash (\epsilon,\ \varnothing) \overset{g}{\underset{e}{\Longrightarrow}} S$$

Indeed, due to Theorem 1, from the condition we can conclude that the divergence signal is never raised during the evaluation, according to "$\underset{e}{\Rightarrow}$"; but in this case the evaluation steps coincide with those, according to "$\Rightarrow$". Thus, the improved search preserves the convergence.

5 IMPLEMENTATION AND EVALUATION

We implemented the improved version of the search, described in the previous section, as a prototype over existing miniKanren implementation for OCaml, called OCanren [14]. We added a history support and the divergence test and changed the syntax to make relational definitions visible for the interpreter. Programs in OCanren can be easily converted to respect the new syntax. In our implementation the divergence propagation is implemented via exception mechanism that is known to be efficient in OCaml. With our prototype implementation some answers for a certain query, being evaluated under the improved search, can be repeated multiple times in comparison with those delivered by the original search. While these two situations are indistinguishable w.r.t. our set-theoretic variant of the semantics, we consider filtering out the repeating answers as a problem for future work.

We evaluated our implementation on a number of benchmarks — virtually all available non-refutationally complete queries reported in the literature, whose incompleteness is caused by conjunction non-commutativity. With no exceptions, the improved search was able to fix refutational incompleteness. Thus, we do not know any realistic cases, which are not improved by our approach. On the other hand, it is rather easy to construct an artificial counterexample. For this, we can define a relation

$$\texttt{dummy} \quad \mapsto \quad \lambda\, x \quad .\ \texttt{dummy}\ (\texttt{S}\ \ x)$$

$$\frac{\Gamma,\ \iota,\ h,\ 1 \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r S^\dagger}{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_e S^\dagger} \quad [\textsc{ClusterStart}^+]$$

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \overset{g_p}{\Longrightarrow}_e \bigcup_j \{(\sigma_j,\ \delta_j)\};\quad \forall j\ :\ \Gamma,\ \iota,\ h \vdash (\sigma_j,\ \delta_j) \overset{\bigwedge_{i\neq p} g_i}{\Longrightarrow}_e S_j}{\Gamma,\ \iota,\ h,\ p \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r \bigcup S_j},\ 1 \le p \le n \quad [\textsc{ClusterStep}^+]$$

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \overset{g_p}{\Longrightarrow}_e \bigcup_j \{(\sigma_j,\ \delta_j)\};\quad \exists j\ :\ \Gamma,\ \iota,\ h \vdash (\sigma_j,\ \delta_j) \overset{\bigwedge_{i\neq p} g_i}{\Longrightarrow}_e \dagger}{\Gamma,\ \iota,\ h,\ p \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r \dagger},\ 1 \le p \le n \quad [\textsc{ClusterDiv}^+]$$

$$\frac{\Gamma,\ \iota,\ h \vdash (\sigma,\ \delta) \overset{g_p}{\Longrightarrow}_e \dagger;\quad \Gamma,\ \iota,\ h,\ p+1 \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r S^\dagger}{\Gamma,\ \iota,\ h,\ p \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r S^\dagger},\ 1 \le p \le n \quad [\textsc{ClusterNext}^+]$$

$$\Gamma,\ \iota,\ h,\ p \vdash (\sigma,\ \delta) \overset{\bigwedge_{i=1}^{n} g_i}{\Longrightarrow}_r \dagger,\ p > n \quad [\textsc{ClusterStop}^+]$$

Figure 7: Improved search: conjuncts reordering

and consider a query `(dummy O) ∧ fail`, where `fail` — some never succeeding goal (like `A ≡ B`). Since the argument of the recursive call to `dummy` is always performed with a more specific argument, the divergence test will never succeed, and the whole query will diverge despite the absence of answers.

In the following subsections we, first, describe the benchmarks in details and, then, present the results of a quantitative evaluation.

5.1 Relations on lists

As we have seen in Section 3, for some simple relations, like `append`^{o}, a recursive call has to be placed last in the sequence of conjuncts in order for the relation to be refutationally complete. Specifically for `append`^{o}, with the improved search the divergence is discovered and refutational incompleteness is fixed regardless the position of the recursive call.

As a more interesting example, consider the `revers`^{o} relation (see Section 3 for the definition). As we've seen, in order for different queries to work it requires different orders of conjuncts to be used in the implementation. Again, the improved search lifts this requirement. Even more interesting, in the query

```
fresh (q) (revers^o (Cons (A, Nil)) q)
```

the divergence is discovered for the recursive call of `append`^{o}, but, as none of conjunct orders within the definition of `append`^{o} help, the improved search goes even further back and switches the conjuncts within the definition of `revers`^{o}. This example demonstrates the importance of operating on a dynamic invocation order.

Another example we've already looked at is relational sorting/permutations. With the improved search, both queries `fresh (q) (perm`^{o} `l q)` and `fresh (q) (perm`^{o} `q l)` terminate for any list `l` and now work in a reasonable time. Moreover, now relational permutations can be used as a (we admit, somewhat exotic) way to calculate the number of permutations with repetitions.

5.2 Binary arithmetics

The implementation of a refutationally complete relational arithmetics is an important problem since it is utilized in a number of elaborated relational specifications. For the performance reasons, it is preferrable to use binary numbers, not comfy Peano encoding, which makes the problem more complicated. As it is shown in [3], the naive implementation of binary arithmetics turns out to be inappropriate due to multiple problems.

Fixing these problems takes a lot of efforts: it requires some additional conditions, excess on the first glance, to be introduced in the specification to ensure the non-overlapping of the cases and the correctness of number representation. And still, even with all these improvements, arithmetic relations diverge for some routine

queries with one order of consituents, and for others with another. To overcome this problem, arithmetics in the standard miniKanren implementation [7] uses digital logic and some advanced techniques of bounding the sizes of the terms [12]. As a result, the implementation, proven to be refutationaly complete, is quite complicated and takes time to understand.

At the same time, some of these problems are exactly the consequences of the non-commutativity of conjunction. Thus, the improved search makes it possible to stick with the naive version (for addition, multiplication, comparisons, division with a reminder) without complicated optimizations.

As the most impressive example, for the division with a reminder, instead of a very complicated recursive definition from [7] (20 lines of code, not including auxiliary functions), one can just write down the following definition

```
divo ↦ λ x y q r .
  (fresh (yq)
     (multo y q yq) ∧
     (pluso yq r x) ∧
     (lto r y)
  )
```

and for all queries

```
fresh (q r) (divo 23 5 q r)
fresh (y q r) (divo 19 y q r)
fresh (x r) (divo x 17 4 r)
```

the search terminates and shows the performance, comparable with the advanced version (here $\overline{n}$ denotes a binary representation of the number n). However, in this case we do not have a proof of refutational completeness.

5.3 Binary trees

For a natural representation of binary trees using two constructors `Leaf` and `Node` $(l,\ r)$, it is easy to implement the relation to count the number of leaves in a tree (using arithmetic relations `plus`^{o} for addition and `pos`^{o} for positivity):

```
leaveso ↦ λ t s .
  ((t ≡ Leaf) ∧ (s ≡ 1)) ∨
  (fresh (l r sl sr)
     (t ≡ Node (l, r)) ∧
     (poso sl) ∧
     (poso sr) ∧
     (leaveso l sl) ∧
     (leaveso r sr) ∧
     (pluso sl sr s)
  )
```

By running this relation backwards

```
fresh (q) (leaveso q n)
```

it becomes possible to generate all binary trees with given number of leaves n. The improved search provides the termination of this query; the number of discovered answers corresponds to the number of such trees, so this relational program may be seen as (an exotic) way of calculating the Catalan numbers.

relation	size	optimistic	optimistic$^+$	pessimistic	pessimistic$^+$
append^o	100	0.0640	0.0706	0.3339	0.0795
	200	0.3604	0.5409	3.4632	0.4953
	300	1.6803	1.9091	14.6674	2.0216
revers^o	30	0.0251	0.0439	0.1753	0.0453
	60	0.1634	0.3069	4.3270	0.3230
	90	0.5830	1.0870	20.8215	1.0620
sort^o	2	0.0002	0.0003	0.0007	0.0028
	3	0.0013	0.0004	0.0481	0.0046
	4	0.0009	0.0022	—	0.0108
	30	0.7473	1.2663	—	36.2886
perm^o	2	0.0006	0.0009	0.0022	0.0040
	3	0.0009	0.0017	—	0.0161
	6	0.5445	0.8317	—	4.5092
mult^o	4	0.0049	0.0057	0.0475	0.0077
	5	0.0145	0.0151	19.7969	0.0189
	8	0.1887	0.2397	—	0.9222
div^o	3	0.0073	0.0072	0.0256	0.0568
	4	0.0173	0.0212	—	0.3543
	5	0.0942	0.1018	—	4.5093
leaves^o	4	0.0125	0.0101	0.0126	0.0151
	5	0.0439	0.0508	60.1352	0.0553
	8	3.6135	3.6208	—	3.7984

Figure 8: The results of a quantitative evaluation: running times of benchmarks in seconds

5.4 Interpreters

Program synthesis with relational interpreters is one of the most useful applications of miniKanren. The construction of a relational interpreter for a small Scheme-like language is considered in details in [6]. However, this simple interpreter also reveals some problems, caused by the non-commutativity of conjunction. For example, consider the following query:

```
fresh (e1 e2 r1 r2) (evalo (list e1 3 e2) Nil (r1 4 r2))
```

Here the first argument of *evalo* is a program (a list of something ($e1$), 3, and something ($e2$)), which is evaluated in an empty environment (Nil) into a list of something ($r1$), 4, and something ($r2$). Clearly, there are no $e1$, $e2$, $r1$, $r2$ to fulfill this contract, yet the evaluation leads to a divergence under the conventional search; no simple reordering can fix it. Under the improved search, however, the contradiction is found and the query terminates with no answers. Relational interpreters, used in practice for more complex problems [4], include a lot of optimizations and take significant effort to implement. We hope that some performance problems with them are caused by the non-commutativity and can be fixed automatically with the improved search.

5.5 A Quantitative Evaluation

While the improved search indeed fixes all considered realistic cases in a qualitative sense, it still introduces some runtime overhead when no divergence is detected. In order to assess the overall performance gain, we performed a quantitative evaluation.

As we've pointed out earlier, the performance of the same specification essentially depends on the "direction" of the query being evaluated. Moreover, the improved search can be faster than the original for one direction and slower for another, which make the

quiantitative evaluation problematic. In order to cope with this difficulty we considered two important versions for each of the benchmarks — "optimistic"and "pessimistic". Both versions were constructed by a careful analysis of each specification-query pair. As a result of the analysis, the "important" conjunctions were discovered. The order of conjuncts in these conjunctions was adjusted to provide the fastest convergence for the optimistic version and the slowest for the pessimistic one. Thus, to some extent these versions provide the efficiency boundaries for a benchmark: when the "direction" of a query plays along with the order of conjuncts in the specification, the whole specification-query performs as in optimistic version; otherwise the pessimistic scenario takes place. Our conjecture was that the improved search would speed up the pessimistic cases and slow down the optimistic, thus we've run the improved search for both versions and compared the running time against that for the original search. The running time in seconds is shown on Fig. 8 (the results for the improved search are marked by "+" superscript); by "–" we marked the cases, when the search did not complete in two minutes[3]. The workstation configuration was Intel Core i7 CPU M 620, 2.67GHz x 4, 8GB RAM running Ubuntu 16.04. We used a native-code OCaml compiler with optimistic settings for the garbage collector to prevent it from interfering and blurring the results.

For all benchmarks, under the improved search the convergence depended neither on the direction of queries, nor on the order of conjuncts. With the exception of very small input sizes, the improved search provided a speedup of up to few orders of magnitude in pessimistic cases. At the same time, the slowdown of optimistic cases under the improved search did not exceed a factor of 2. The behavior in the pessimistic cases also allows us to discriminate between two interesting situations:

- either in the pessimistic case the standard search was much slower than the improved one, but still measurable, and the improved search worked with approximately the same performance, as the standard search for the optimistic case;
- or in the pessimistic case the standard search quickly became non-measurable (indicated by "—"), and the improved search for the pessimistic case worked much slower, than the standard one for the optimistic case, but still measurable.

We may conclude that for the wide variety of realistic cases our improvement indeed delivers a fully-automatic and lightweight way to provide refutational completeness. In some cases, however, in order to achieve the best performance, a relational specification has to be optimized manually.

6 RELATED WORKS

The non-commutativity of conjunction evaluation in miniKanren is a well-known problem. In [3] some language extensions are discussed that, presumably, can be used to provide the commutativity. They include both simple enumeration of conjunct orders and more advanced techniques, based on a combination of tabling, parallel goal evaluation, and continuations. However, by now no successful implementations were reported. The tabling technique, described in the same work, can indeed be used to provide the convergence of some queries, but it deals with the problems, orthogonal to the non-commutativity, and, thus, does not heal the queries, which we do (but heals some other cases, like divergence of path-finding queries for graphs with cycles, which we do not).

For a number of problems, some *ad-hoc* refutationally complete solutions were already presented before. For example, in [7] a number of relations for binary arithmetics, implemented using the idea of bounding the sizes of terms, are presented. In a follow-up paper [12] this technique is explained in details, and the proof of refutational completeness is given. Unfortunately, the specifications, written using this technique, are verbose and hard to understand, and the implementation requires insight. Our improvement, on the other hand, makes it possible to stick with the simplest definitions, and although we do not provide a proof of refutational completeness for each case, for the majority of realistic queries they converge and demonstrate the same performance, as those, implemented with advanced methods.

In a broader context of logical programming, the problem of search convergence/termination was addressed multiple times. However, it is rather hard to establish a direct correspondence between our proposal and the reported results, since they were developed for essentially different language. For example, in [19] a tabling-based improvement of a resolution-based search — OLDT resolution — is described, and a complete search strategy is developed. However, in miniKanren the original search is already complete, and we address rather a different issue. We can speculate, that OLDT resolution roughly corresponds to the miniKanren with tabling and, thus, possesses the same properties, and relates to our proposal in a similar manner.

An approach to a static transformation of logic programs is described in [16]. Given a definitional program and a query, the transformation results in a new program and a new query that is guaranteed to converge in a finite number of steps under arbitrary search strategy. The approach is proven to be sound and complete for queries that deliver a finite number of solutions, and utilizes the fact, that all these solutions can be discovered via a complete traversal of an SLD-tree up to a finite depth. The authors introduce a sufficient condition for the finiteness of the number of solutions by utilizing the notion of level mapping and present a method to calculate the limit for the SLD-tree depth. The important distinction with our proposal is that we perform a dynamic analysis; as we argued in Section 3, there are cases, when no concrete static form of a specification can guarantee convergence for any query of interest. In addition, our approach detects the divergence rather than the convergence of program evaluation.

7 CONCLUSION

We presented an improvement of a search strategy for relational programming that is aimed at improving refutational completeness. We've proven that in the case of a finite number of answers our modification is a proper improvement over the original strategy in terms of convergence. Our evaluation shows, that w.r.t. the improved search many practically important refutationally incomplete queries became refutationally complete; in addition, in a number

[3]The evaluation repository is available at https://github.com/rozplokhas/OCanren-improved-search

of cases the performance was greatly improved since our modification, as a side effect, causes the search to choose more "optimistic" branches.

We can identify the following directions for future work.

First, we believe, that our result on refutational improvement for a finite number of answers can be extended to the general case as well (note, in our current development we did not make any use of the *completeness* property of miniKanren search). For this, we would also need another, more general, semantics.

Another direction is extending the language with disequality constraints. Our evaluation has shown, that disequality constraints do not compromise our improvement in all user benchmarks, but we do not have a proof, that they are indeed harmless.

Next, we are working on a certified proof of the main theorem in Coq.

Finally, our practical evaluation is performed only for a prototype. We consider the embedding of our improvement in a full-fledged implementation to be an important task.

REFERENCES

[1] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. In *Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming*.

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

[3] William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.

[4] 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

[5] William E. Byrd and Daniel P. Friedman. 2007. αkanren: A Fresh Name in Nominal Logic Programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*. 79–90.

[6] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

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

[8] Michael Hanus, Herbert Kuchen, and Juan José Moreno-Navarro. 1995. Curry: A Truly Functional Logic Language. (1995).

[9] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

[10] Jason Hemann and Daniel P. Friedman. 2015. A Framework for Extending microKanren with Constraints. In *Proceedings of the 2015 Annual Workshop on Scheme and Functional Programming*.

[11] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A Small Embedding of Logic Programming with a Simple Complete Search. *SIGPLAN Not.* 52, 2 (Nov. 2016), 96–107. https://doi.org/10.1145/3093334.2989230

[12] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-Chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). In *Proceedings of the 9th International Conference on Functional and Logic Programming (FLOPS'08)*. Springer-Verlag, Berlin, Heidelberg, 64–80. http://dl.acm.org/citation.cfm?id=1788446.1788456

[13] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). *SIGPLAN Not.* 40, 9 (Sept. 2005), 192–203. https://doi.org/10.1145/1090189.1086390

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

[15] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed Relational Conversion. In *Proceedings of the International Symposium on Trends in Functional Programming*.

[16] Dino Pedreschi and Salvatore Ruggieri. 1999. Bounded Nondeterminism of Logic Programs. In *International Conference on Logic Programming (D. De Schreye, Ed.)*.

[17] Z Somogyi, F J. Henderson, and T C. Conway. 1998. Mercury, an Efficient Purely Declarative Logic Programming Language. (10 1998).

[18] Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

[19] Hisao Tamaki and Taisuke Sato. 1986. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming*. 84–98.