**Northeastern University**
Khoury College of Computer Sciences

# Proceedings of the 2020 miniKanren and Relational Programming Workshop

Online—August 27th, 2020

Edited by Jason Hemann, Northeastern University
and Dmitri Boulytchev, St. Petersburg State University

# Preface

This report aggregates the papers presented at the second miniKanren and Relational Programming Workshop, held online on August 27th, 2020 co-located with the 25th International Conference on Functional Programming.

The miniKanren and Relational Programming Workshop solicits papers and talks on the design, implementation, and application of miniKanren-like languages. A major goal of the workshop is to bring together researchers, implementers, and users from the miniKanren community, and to share expertise and techniques for relational programming. Another goal for the workshop is to push the state of the art of relational programming.

Twelve papers were submitted to the workshop, and each paper was reviewed by at least three members of the program committee. After deliberation, eleven of those papers were accepted to the workshop.

In addition to those eleven papers presented

- Eelco Visser (Delft University of Technology) gave an invited morning keynote speech, *Executing Declarative Language Definitions*, demonstrating by example declarative language definition in the Spoofax language workbench, a programming environment for the development of programming languages.
- Matthew Might (University of Alabama at Birmingham and Harvard Medical School) gave an invited afternoon keynote speech, *The Pill is in The Proof: Saving Lives with Logic*, exploring the use of miniKanren for precision medicine as embodied by the drug-repurposing tool mediKanren.

Special thanks to Eelco Visser and Matthew Might for their presentations and the program committee for reviewing and deliberating on the submitted papers.

## Program Committee

Dmitri Boulytchev, St. Petersburg State University, (PC Chair)
Adam Foltzer, Fastly
Jason Hemann, Northeastern University (General Chair)
Ekaterina Komendantskaya, Heriot-Watt University
Jan Midtgaard, University of Southern Denmark
Joseph P. Near, University of Vermont
Gregory Rosenblatt, University of Alabama at Birmingham
Ilya Sergey, Yale-NUS College and National University of Singapore
Kanae Tsushima, National Institute of Informatics

# Contents

# On Fair Relational Conjunction*

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

We present a new, more symmetric evaluation strategy for conjunctions in miniKanren. Unlike the original unfair directed conjunction, our approach controls the order of conjunct execution based on the properties of structurally recursive relations. In this paper we describe operational semantics for both "classical" and "fair" conjunctions. We also compare the performance of classical and fair conjunctions on a number of examples and discuss the results of the evaluation.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; **Semantics**;

Additional Key Words and Phrases: relational programming, miniKanren, evaluation strategies, operational semantics

## 1 INTRODUCTION

miniKanren [7, 8] is known for its capability to express solutions for complex problems [4, 6, 11] in the form of compact declarative specifications. This minimalistic language has various extensions [1, 3, 5, 14] designed to increase its expressiveness and declarativeness. However, *conjunction* in miniKanren has somewhat imperative flavor. The evaluation of conjunction is asymmetrical and the order of conjuncts affects not only the performance of the program but even the convergence. As a result, the order of conjuncts determines control flow in a relational program. We may call this directed behavior of conjunction *unfair*.

The problem of unfair behavior of conjunction has already been examined from several different points of view. One aspect of the unfair behavior of conjunction is to prioritize the evaluation of the independent branches which conjunction generates. In the original miniKanren a higher priority is given to an earlier branch. However, there is an approach [10] which allows one to balance the evaluation time of different branches. This makes the conjunction behavior fairer, but the order of the conjuncts still affects both performance and convergence. Also, the conjunction can be made fairer if we are capable of detecting the divergence in its conjuncts. The approach [12] detects the divergence at run time and performs switching of conjuncts. In this case, the data that was received during the evaluation of the diverging conjunct is erased. The approach turned out to be efficient in practice. However, the conservative rearrangement of the conjuncts does not use the information obtained when evaluating the conjunct before the rearrangement. There are also examples for this approach where the order of the conjuncts affects convergence.

The contribution of this paper is a more declarative approach to the evaluation of relational programs of miniKanren. This approach executes the conjuncts alternately, choosing a more optimal execution order. The fair conjunction that we propose is comparable in efficiency to the classic unfair one, but the order of the conjuncts weakly affects both efficiency and convergence. Our approach also demonstrates a more convergent behavior: we

*The reported study was funded by RFBR, project number 19-31-90053

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


miniKanren.org/workshop/2021/8-ART1

present some examples where classical conjunction diverges for any order of conjuncts while the fair conjunction converges.

The paper is organized as follows. In Section 2 we discuss the advantages and drawbacks of the classical directed conjunction. Section 3 contains a description of miniKanren syntax as well as operational semantics based on the unfolding operation. In Section 4 we present the semantics of a naive fair conjunction, and in Section 5 we extend these semantics by controlling the conjunction order based on structural recursion. Section 6 is devoted to the evaluation and performance comparison on different semantics in the form of interpreters. The final section concludes.

## 2 DIRECTED CONJUNCTION

In this section we consider the classic directed conjunction in the original miniKanren and demonstrate its advantages and drawbacks on examples.

In miniKanren there is a significant difference between disjunction and conjunction evaluation strategy. The arguments of disjunction are evaluated in an *interleaving* manner, switching elementary evaluation steps between disjuncts, which provides the completeness of the search. Conjunction, on the other hand, waits for the answers from the first conjunct and then calculates the second conjunct in the context of each answer.

On the one hand, this strategy is easy to implement and allows one to explicitly specify the order of evaluation when this order is essential. On the other hand, the strategy amounts to non-commutativity: the convergence of a conjunction can depend on the order of its arguments. For example, the relation

```
let rec freeze^o x = x ≡ true ∧ freeze^o x
```

either converges in one recursion step or diverges. The conjunction (`x ≡ false ∧ freeze^o x`) converges, but the same conjunction with the reverse order of conjuncts (`freeze^o x ∧ x ≡ false`) diverges. Indeed, in the first case the first conjunct produces exactly one answer which contradicts the unification in the body of `freeze^o`. In the second case the relation `freeze^o` diverges and does not produce any answer. As a result we will never begin to evaluate the second conjunct.

```
1  let rec append^o x y xy =
2    (x ≡ [] ∧ y ≡ xy) ∨
3    fresh (e xs xys) (
4      x ≡ e : xs ∧
5      xs ≡ e : xys ∧
6      append^o xs y xys)
```

```
7  let rec revers^o x y =
8    (x ≡ [] ∧ y ≡ []) ∨
9    fresh (e xs ys) (
10     x ≡ e : xs ∧
11     revers^o xs ys ∧
12     append^o ys [e] y)
```

Fig. 1. Relational list reversing

The problem in this particular example can be alleviated by using a conventional rule for miniKanren, which says that unifications must be moved first in a cluster of conjunctions. But the rule does not work for clusters with more than one relational call. Consider, for instance, the relation `revers^o` (Fig. 1), which associates an arbitrary list with the list containing the same elements in reverse order. In this relation we can see a couple of conjuncts: `revers^o` on line 11 and `append^o` on line 12. With this particular order, the call (`revers^o [1, 2, 3] q`) converges, but in reverse order it diverges after an answer is found. Moreover, the reverse order negatively affects the performance of the answer evaluation.

At the same time the call (`revers^o q [1, 2, 3]`) for a given order of conjuncts diverges, and for the reverse order it converges. As a result a different order of conjuncts is desirable depending on their runtime values.

In the following sections we describe an approach for automatically determining a "good" order during program evaluation.

## 3 THE SEMANTICS OF DIRECTED CONJUNCTION

In this section we introduce a small-step operational semantics to define the behavior of the original miniKanren with directed conjunction.

Note, although at the moment there exists a certified semantics [13] of miniKanren, however, it makes a distinction between the first and the second conjuncts, which greatly complicates the task of rearranging the conjuncts in the process of program evaluation. Therefore, for this research, we have developed a new semantics that does not distinguish between conjuncts from the start.

The semantics which we propose are based on unfolding of relational calls. At each step we select a call from the current state of the program and unfold it. This process continues until no calls remain in the state. If at some step the state becomes empty then the evaluation converges. Otherwise, the evaluation diverges. Below we define these semantics formally.

$$
\begin{array}{rcll}
\mathcal{C} & = & \{C_1^{k_1}, C_2^{k_2}, \ldots\} & \text{constructors with arities} \\
\mathcal{X} & = & \{x_1, x_2, \ldots\} & \text{syntax variables} \\
\mathcal{A} & = & \{\alpha_1, \alpha_2, \ldots\} & \text{semantic variables} \\
\mathcal{T}_{\mathcal{X}} & = & \mathcal{X} \mid C_i^{k_i}(\mathcal{T}_{\mathcal{X}}^1, \ldots, \mathcal{T}_{\mathcal{X}}^{k_i}) & \text{syntax terms} \\
\mathcal{T}_{\mathcal{A}} & = & \mathcal{A} \mid C_i^{k_i}(\mathcal{T}_{\mathcal{A}}^1, \ldots, \mathcal{T}_{\mathcal{A}}^{k_i}) & \text{semantic terms} \\
\mathcal{F} & = & \{F_1^{k_1}, F_2^{k_2}, \ldots\} & \text{names of relations with arities} \\
\mathcal{G} & = & \mathcal{T}_{\mathcal{X}} \equiv \mathcal{T}_{\mathcal{X}} & \text{unification} \\
 & \mid & \mathcal{G} \vee \mathcal{G} & \text{disjunction} \\
 & \mid & \mathcal{G} \wedge \mathcal{G} & \text{conjunction} \\
 & \mid & \textbf{fresh } (\mathcal{X})\ \mathcal{G} & \text{fresh variable introduction} \\
 & \mid & F_i^{k_i}(\mathcal{T}_{\mathcal{X}}^1, \ldots, \mathcal{T}_{\mathcal{X}}^{k_i}) & \text{relation call} \\
\mathcal{S} & = & F_i^{k_i} = \lambda \mathcal{X}_1 \ldots \mathcal{X}_n.\ \mathcal{G} & \text{relations}
\end{array}
$$

Fig. 2. The syntax of relational language

First, we define the syntax of the language (Fig. 2). We define the set of syntax terms $\mathcal{T}_{\mathcal{X}}$ and the set of semantic terms $\mathcal{T}_{\mathcal{A}}$ using constructors $\mathcal{C}$, syntactic variables $\mathcal{X}$ and semantic variables $\mathcal{A}$. Syntactic variables $\mathcal{X}$ are needed to introduce relation arguments and fresh variables. Semantic variables $\mathcal{A}$ can not occur in the source program, but they are introduced during the evaluation of the **fresh** construct.

Also we define the set $\mathcal{F}$ of relation names with arities. Next, we describe the set of goals $\mathcal{G}$. A goal is either a unification of terms, or a disjunction, conjunction, introduction of fresh variable, or a call of a relation. Finally, we define the set of relations $\mathcal{S}$. Each relation consists of a name with arity, a list of argument names, and a goal as its body.

In addition to syntax, we define an intermediate state of relational program.

$$
\begin{array}{rcll}
c_i & = & F_{j_i}^{k_{j_i}}\ (t_{\mathcal{A}}^1, \ldots, t_{\mathcal{A}}^{k_{j_i}}) & \text{calls} \\
C & = & c_1 \wedge \cdots \wedge c_n & \text{conjunction of calls} \\
\mathfrak{T} & = & \mathfrak{T} \vee \mathfrak{T} & \text{disjunction state} \\
 & \mid & \langle \sigma;\ i;\ C \rangle & \text{leaf state}
\end{array}
$$

The state has a shape of a disjunction tree. An internal node "$\vee$" corresponds to the disjunction of two descendant states. Its leaves contain intermediate substitutions $\sigma$, a counter of semantic variables $i$ and a conjunction of calls $C$. A substitution $\sigma$ is a mapping from semantic variables to semantic terms. The counter of semantic variables is necessary to evaluate the **fresh** construct, which introduces a semantic variable with a new counter. The conjunction of calls contains calls $c_i$ which must be evaluated in this branch. The number of conjuncts in a conjunction may be zero, which we denote as $\epsilon$.

We expand the set of states with an empty state

$$\mathfrak{T} + = \varnothing.$$

which corresponds to the completion of the evaluation.

Also we introduce two auxiliary functions for working with the program state. The first one, *union*, combines two extended states:

$$union\,(T_1,\,T_2) = \begin{cases} T_2, & \text{if } T_1 = \varnothing \\ T_1, & \text{if } T_1 \neq \varnothing \text{ and } T_2 = \varnothing \\ T_1 \vee T_2, & \text{otherwise} \end{cases}$$

If one of the states is empty *union* returns another state. If both states are not empty then the function returns the combined state.

The next auxiliary function, *push*, is needed to construct the state after the unfolding:

$$push\,(C,\,T) = \begin{cases} \varnothing, & \text{if } T = \varnothing \\ \langle \sigma;\, i;\, C_1 \wedge \bar{C} \wedge C_2 \rangle, & \text{if } T = \langle \sigma;\, i;\, \bar{C} \rangle \text{ and } C = C_1 \wedge \square \wedge C_2 \\ push\,(C, T_1) \vee push\,(C, T_2), & \text{if } T = T_1 \vee T_2 \end{cases}$$

The first argument of this function is a conjunction of calls which contains a *hole* "$\square$". The hole corresponds to a position of the call which we are unfolding. The second argument is the state to unfold. This function recursively traverses the state and for any leaf replaces its calls, making a substitution of this leaf calls into the hole in the first argument of *push*.

Now we define the semantics for the unfolding operation. This semantics (Fig. 3) evaluates a call of a relation and a substitution into a state which corresponds to the body of this relation. Since unfolding is a finite process we can describe it in a big-step style, defining a relation "$\Rightarrow$".

The top-level rule in [Unfold]. Thus, this is the only rule which infers "$\Rightarrow$". It starts the unfolding process of call $F$ with the list of arguments $\bar{t}$ in the context of substitution $\sigma$ and counter of semantic variables $i$. First of all, the call $F$ is replaced by the body $b$ of the relation. Also the arguments $\bar{x}$ are substituted by terms $\bar{t}$ and the state $\langle \sigma;\, i;\, \epsilon \rangle$ is initialized. Next, we evaluate the relation body into the corresponding state using the rest of the rules.

The rule [EmptyState] handles the empty state case. The rules [UnifyFail] and [UnifySuccess] perform unification. If the most general unifier (MGU) exists, then we apply rule [UnifySuccess], which updates the substitution $\sigma$. If the MGU does not exist, then we apply rule [UnifyFail], which leads to the empty state.

Since this semantics should unfold a call exactly once, we leave all the nested calls unchanged. This behavior is specified by rule [Call]. The nested call is not evaluated, but added to the conjunction, which is contained in the state.

The rule [Fresh] corresponds to the introduction of a fresh variable. In this rule we replace a syntax variable $x$ with a semantic variable $\alpha_i$. We also increment the counter of semantic variables.

The rules [DisjGoal] and [DisjState] are required to evaluate disjunctions. The first rule evaluates both disjuncts and combines them into a new state using auxiliary function *union*. The second rule handles a disjunction

$$\frac{F = \lambda \bar{x}.b \qquad \langle \sigma;\, i;\, \epsilon \rangle \vdash b[\bar{x} \leftarrow \bar{t}] \rightsquigarrow T}{(\sigma, i) \vdash F(\bar{t}) \Rightarrow T} \quad [\textsc{Unfold}]$$

$$\varnothing \vdash g \rightsquigarrow \varnothing \quad [\textsc{EmptyState}]$$

$$\frac{\nexists\, mgu(t_1, t_2, \sigma)}{\langle \sigma;\, i;\, C \rangle \vdash (t_1 \equiv t_2) \rightsquigarrow \varnothing} \quad [\textsc{UnifyFail}]$$

$$\frac{\bar{\sigma} = mgu(t_1, t_2, \sigma)}{\langle \sigma;\, i;\, C \rangle \vdash (t_1 \equiv t_2) \rightsquigarrow \langle \bar{\sigma};\, i;\, C \rangle} \quad [\textsc{UnifySuccess}]$$

$$\langle \sigma;\, i;\, C \rangle \vdash F(\bar{t}) \rightsquigarrow \langle \sigma;\, i;\, F(\bar{t}) \wedge C \rangle \quad [\textsc{Call}]$$

$$\frac{\langle \sigma;\, i+1;\, C \rangle \vdash g[x \leftarrow \alpha_i] \rightsquigarrow T}{\langle \sigma;\, i;\, C \rangle \vdash \textbf{fresh}\ x.\, g \rightsquigarrow T} \quad [\textsc{Fresh}]$$

$$\frac{\langle \sigma;\, i;\, C \rangle \vdash g_1 \rightsquigarrow T_1 \qquad \langle \sigma;\, i;\, C \rangle \vdash g_2 \rightsquigarrow T_1}{\langle \sigma;\, i;\, C \rangle \vdash g_1 \vee g_2 \rightsquigarrow union(T_1, T_2)} \quad [\textsc{DisjGoal}]$$

$$\frac{g \neq g_1 \vee g_2 \qquad T_1 \vdash g \rightsquigarrow T_3 \qquad T_2 \vdash g \rightsquigarrow T_4}{T_1 \vee T_2 \vdash g \rightsquigarrow union(T_3, T_4)} \quad [\textsc{DisjState}]$$

$$\frac{\langle \sigma;\, i;\, C \rangle \vdash g_1 \rightsquigarrow T \qquad T \vdash g_2 \rightsquigarrow \bar{T}}{\langle \sigma;\, i;\, C \rangle \vdash g_1 \wedge g_2 \rightsquigarrow \bar{T}} \quad [\textsc{Conj}]$$

Fig. 3. Big step semantics of unfolding

which is contained in the state. As in the previous rule we perform two independent evaluations and then combine the results into a new state.

The last rule [Conj] describes the evaluation of conjunction. In this case we evaluate the first conjunct into a state $T$, and then evaluate the second conjunct in the context of $T$. Thus, the second conjunct will be evaluated in the context of all leaves of the state $T$. Note, if the first conjunct is evaluated into empty state then we can only apply the rule [EmptySet] to the second conjunct. Applying this rule will result in an empty state.

Now we have everything we need to define the semantics of a relational language with directed conjunction (Fig. 4). This small-step semantics sequentially evaluates a state and periodically produces answer substitutions. If an answer is found during program evaluation it is indicated above the transition symbol "$\rightarrow$"; otherwise, "$\circ$" is indicated.

If the current state is a leaf and does not contain any calls then we have an answer in the form of a substitution. In this case, we apply the rule [Answer].

Also, if the current state is a leaf but contains at least one call, we apply rule [ConjUnfold]. In this case we unfold the leftmost call $c$. Then we construct a new state from the remaining calls $C$ and the unfolding result $T$ using the function *push*.

Finally, if the current state is a disjunction then we evaluate the left disjunct $T_1$. We apply either rule [Disj] or rule [DisjStep] depending on the results of the evaluation for the left disjunct. The first rule corresponds to the empty state and returns the second disjunct $T_2$ as a result. The second rule corresponds to a non-empty state and returns a new state ($T_2 \vee \bar{T}_1$). Rearrangement of disjuncts is a necessary action called *interleaving* [9]. It guarantees the completeness of the search.

To make initial state from a call $c$ we need to replace all its syntactic variables with semantic ones:

$$\langle \sigma;\, i;\, \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\frac{(\sigma, i) \vdash c \Rightarrow T}{\langle \sigma;\, i;\, c \wedge C \rangle \xrightarrow{\circ} push(\square \wedge C, T)} \qquad \text{[ConjUnfold]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 4. Semantics with directed conjunction

$$\langle \{\};\, n;\, c[x_0 \leftarrow \alpha_0, \ldots, x_{n-1} \leftarrow \alpha_{n-1}] \rangle.$$

This semantics differs from certified semantics and classical implementations primarily in step size. The unfolding operation performs many actions in a row. But in the classical case interleaving is performed after each elementary action. However, the semantics we presented has retained some common features: disjuncts are switched after each step, and conjunctions are evaluated strictly from left to right.

The order of conjuncts strongly affects the results of the evaluation precisely because of the strictly fixed order of evaluation of the conjuncts. In the following sections we propose two semantics that handles conjunctions more flexibly.

## 4 A NAIVE FAIR CONJUNCTION

In this section we consider the semantics of miniKanren which fairly unfolds conjuncts. Also, we discuss their advantages and drawbacks.

Instead of unfolding the leftmost call to completion we take some finite number $N$ and bound the number of unfolding steps by $N$. If after $N$ unfoldings the leftmost call is not eliminated we start to unfold the next call. This process will continue for all calls of the leaf state. When all calls are unfolded $N$ times, we again return the leftmost once again. We call $N$ *unfolding bound*.

To implement this behavior, we need to modify the state structure.

$$\begin{array}{rcll} c_i & = & F_{j_i}^{k_{j_i}}\ (t_{\mathcal{A}}^1, \ldots, t_{\mathcal{A}}^{k_{j_i}}) & \text{calls} \\ C & = & c_1^{m_1} \wedge \cdots \wedge c_n^{m_n} & \text{conjunction of marked calls} \\ \mathfrak{T} & = & \mathfrak{T} \vee \mathfrak{T} & \text{disjunction state} \\ & | & \langle \sigma;\, i;\, C \rangle & \text{leaf state} \end{array}$$

For each call $c_i$ of the leaf we add a natural number $m_i$ which specifies the remaining number of unfoldings. Therefore, each call in the state is marked with an unfolding counter.

Note that the syntax, auxiliary function *union*, the semantics of unfolding are all remained unchanged. The function *push* also preserves its behavior, but now it takes a conjunction of marked calls with the hole and a state in a new form. In addition we need yet another function *set*, which takes an old state without counters in the leaves and a natural number. This number is attached to each call in the state:

$$set(T, m) = \begin{cases} \varnothing, & \text{if } T = \varnothing \\ \langle \sigma;\ i;\ c_1^m \wedge \ldots \wedge c_n^m \rangle, & \text{if } T = \langle \sigma;\ i;\ c_1 \wedge \ldots \wedge c_n \rangle \\ set(T_1, m) \vee set(T_2, m), & \text{if } T = T_1 \vee T_2 \end{cases}$$

Now we are ready to modify the semantics. A new semantics (Fig. 5) evaluates disjuncts in the old way, but it unfolds conjuncts fairly.

$$\langle \sigma;\ i;\ \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_n^0 \rangle \xrightarrow{\circ} (\sigma, i, c_1^N \wedge \ldots \wedge c_n^N) \qquad \text{[ConjZero]}$$

$$\frac{m > 0 \qquad (\sigma, i) \vdash c_k \Rightarrow T \qquad set(T, m-1) = \bar{T}}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge c_k^m \wedge C \rangle \xrightarrow{\circ} push(c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge \square \wedge C, \bar{T})} \qquad \text{[ConjUnfold]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 5. Simple fair semantics

First of all, we note that semantics depends on unfolding bound. If the bound is 1, then the evaluation of conjunctions becomes very similar to that for disjunctions. As in the case of disjunction, we switch to the unfolding of a next conjunct after each step. If the bound is set to infinity, then the conjunction will behave like in the directed case. Indeed, the counter of the leftmost conjunct will never become zero and this conjunct will unfold until completion.

Now we consider the semantics in more detail. The rules [Answer], [Disj] and [DisjStep] remain unchanged. However, we introduced two new rules for handling conjunctions. The rule [ConjUnfold] unfolds the leftmost conjunct $c_k$ whose counter $m$ is greater than zero. All calls in the new state $T$, which we have after the unfolding, require to attach an updated counter; we do this using the function *set*. If all calls in the leaf have a zero counter, then we apply the rule [ConjZero]. This rule updates all counters in the leaf, setting them all to unfolding bound.

```
let rec repeat^o e l =
  (l ≡ []) ∨
  fresh (ls)
    (l ≡ e : ls ∧
     repeat^o e ls)

let divergence^o l =
  repeat^o C1 l ∧
  repeat^o C2 l
```

Fig. 6. An example to demonstrate fair conjunction superiority

This semantics more fairly distributes resources between conjuncts. Because of this, relational queries converge more often. Let's go back to the `revers`$^o$ example (Fig. 1). As we said earlier, for directed conjunction the query (`revers`$^o$ [1, 2, 3] `q`) converges in the specified order of conjuncts and diverges in the reverse order. At the same time, the query (`revers`$^o$ `q` [1, 2, 3]) diverges in the specified conjunct order but converges in the reverse order. In the case of fair conjunction, however, both queries converge in both conjunct orders for any finite unfolding bound.

Moreover, some examples diverge for any order of conjuncts in case of directed conjunction but converge in case of fair conjunction (see Fig. 6). The relation searches for lists which on the one hand contain terms $\mathtt{C}_1$ only, and on the other, contain terms $\mathtt{C}_2$ only. Obviously, only an empty list has this property.

We can find this answer by evaluating query `divergence`$^o$ `l` under directed conjunction. However, the search for other answers will diverge, and any order of conjuncts preserves this effect. At the same time fair conjunction converges for any finite unfolding bound and any order of conjuncts.

However, in practice this approach has an unstable performance. On the one hand, with a certain unfolding bound the efficiency of a fair conjunction is comparable to the directed conjunction with optimal order of conjunctions. On the other hand, with the wrong unfolding bound we can get an extremely inefficient evaluation, which is hundreds of times slower than the directed conjunction. Therefore, instead of choosing the unfolding bound once and for all we would like to determine it dynamically for each conjunct.

## 5 FAIR CONJUNCTION BY STRUCTURAL RECURSION

In this section we consider a generalized semantics of miniKanren with a fair conjunction which determines the unfolding bound dynamically. We also consider its specific implementation which makes use of structural recursion of relations.

In the general case we want to parameterize the semantics with an unfolding predicate *pred*. This predicate takes a substitution and a call as arguments. It returns **true** if the call needs to be unfolded further, and **false**, if we need to move on to the next conjunct. We do not get rid of the unfolding bound completely since it is still needed to handle the case when *pred* is false for all calls in a leaf.

$$\langle \sigma;\ i;\ \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\frac{\bigvee_{j=1}^{n} pred(\sigma, c_j) = \bot}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_n^0 \rangle \xrightarrow{\circ} \langle \sigma;\ i;\ c_1^N \wedge \ldots \wedge c_n^N \rangle} \qquad \text{[ConjZero]}$$

$$\frac{m_k > 0 \qquad \bigvee_{j=1}^{n} pred(\sigma, c_j) = \bot \qquad (\sigma, i) \vdash c_k \Rightarrow T \qquad set(T, m_k - 1) = \bar{T}}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge c_k^{m_k} \wedge \ldots \wedge c_n^{m_n} \rangle \xrightarrow{\circ} push(c_1^0 \wedge \ldots \wedge c_i^0 \wedge \square \wedge \ldots \wedge c_n^{m_n}, \bar{T})} \qquad \text{[ConjUnfold]}$$

$$\frac{\bigvee_{j=1}^{k-1} pred(\sigma, c_j) = \bot \quad pred(\sigma, c_k) = \top \quad (\sigma, i) \vdash c_k \Rightarrow T \quad set(T, \max(0,\ m_k - 1)) = \bar{T}}{\langle \sigma;\ i;\ c_1^{m_1} \wedge \ldots \wedge c_{k-1}^{m_{k-1}} \wedge c_k^{m_k} \wedge C_2 \rangle \xrightarrow{\circ} push(c_1^{m_1} \wedge \ldots \wedge c_{k-1}^{m_{k-1}} \wedge \square \wedge C_2, \bar{T})} \qquad \text{[ConjUnfoldPred]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 7. Semantics of fair conjunction by structural recursion

The semantics parameterized by the unfolding predicate is shown in Fig. 7. Since we modify only the behavior of conjunction the rules [Answer], [Disj] and [DisjStep] remain unchanged. Three updated rules are responsible for conjunction behavior. If the predicate *pred* is $\top$ for at least one call, then we apply the rule [ConjUnfoldPred], which unfolds the leftmost such call and decrements its counter. If the predicate *pred* is $\bot$ for all calls, but there is at least one call with a nonzero counter, then we apply the rule [ConjUnfold], which unfolds the leftmost such call and decrements its counter. If the predicate is $\bot$ for all calls and all the counters are equal to zero, then we apply the rule [ConjZero], which sets all the counters to unfolding bound.

As for the predicate, we need a criterion that can tell a call that is profitable to unfold now apart from a call which is worth deferring. We propose a criterion that works correctly for structurally recursive relations. Such relations have at least one argument which structurally decreases with each step of the recursion. This property allows us to control the depth of unfolding. We propose to use the following predicate:

$$pred(\sigma, F^k(t_1, \ldots, t_k)) = \begin{cases} & \text{if } F^k \text{ is structural recursion relation,} \\ \top, & i \text{ is number of structural recursion argument,} \\ & t_i \text{ is not fresh variable in } \sigma \\ \bot, & \text{otherwise.} \end{cases}$$

As long as at least one argument along which structural recursion is performed is not a free variable, we continue to unfold this call. If all such arguments are free, then the call will diverge in the current substitution, so we proceed to evaluate the next call. Since structurally recursive arguments decrease, in a finite number of steps the evaluation will either complete, or all arguments of structural recursion will become free variables.

```
let rec appendᵒ x y xy =
  (x ≡ [] ∧ y ≡ xy) ∨
  fresh (e xs xys) (
    x ≡ e : xs ∧
    xs ≡ e : xys ∧
    appendᵒ xs y xys)
```

Fig. 8. Relational concatenation

For example, the relation $\mathsf{append}^o$ (Fig. 8) is structurally recursive on its first and third arguments. Indeed, the nested call $\mathsf{append}^o$ takes `xs` as its first argument, which is a subterm of `x`. Also, $\mathsf{append}^o$ takes `xys` as the third argument, which is a subterm of `xy`. If at least one of them is a fixed-length list, then the relation will converge. Otherwise, $\mathtt{x} \equiv t_1 : \ldots t_n : \alpha_1$ and $\mathtt{xy} \equiv \bar{t}_1 : \ldots \bar{t}_m : \alpha_2$. Therefore, in $max(n, m)$ steps, both arguments become free variables.

We are currently working on proving the independence of this semantics from the order of the conjuncts in the case when all relations are structurally recursive.

## 6 EVALUATION

In this section we present the results of the evaluation of three semantics on a set of examples; the semantics were implemented in Haskell in the form of interpreters. For evaluation we've chosen two simple programs (list reversing and list sorting) and three more complicated (the "Hanoi Towers"[1] solver, the "Bridge and torch

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi

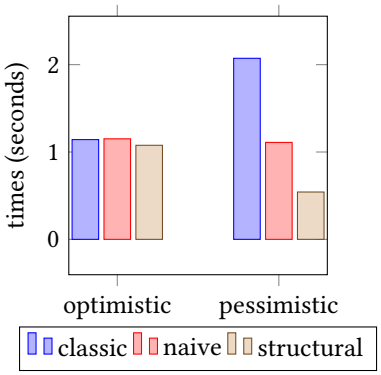


Fig. 9. revers$^o$ evaluation for a list with a length of 90

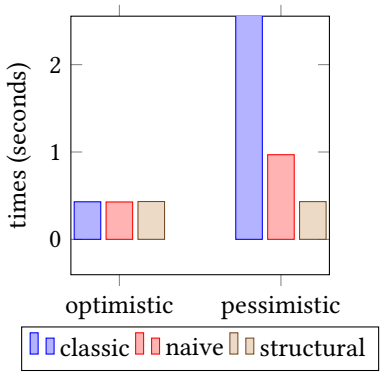


Fig. 10. sort$^o$ evaluation for a list with a length of 5

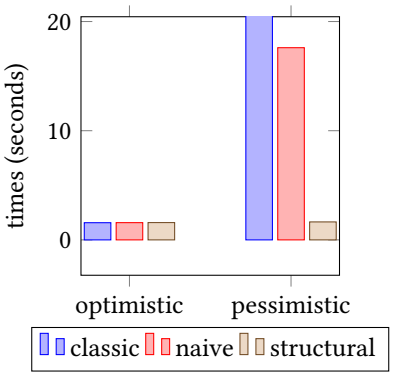


Fig. 11. "The Tower of Hanoi" solver evaluation

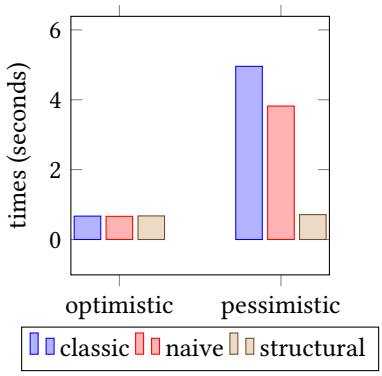


Fig. 12. "Bridge and torch problem" solver evaluation

problem"[2] solver and "Water pouring puzzle"[3] solver). Each program was written in two versions: "optimistic" (with the order of important conjuncts set to provide the best performance) and "pessimistic" (with the order of important conjuncts set to provide the worst performance). Also we evaluated list reversing and list sorting in both directions. In the case of the list reversing, queries (`revers`$^o$ `[1;2;3] q`) and (`revers`$^o$ `q [1;2;3]`) will give the same answer `q = [3;2;1]` but the "optimistic" order of conjuncts is different for them. In the case of list sorting, queries (`sort`$^o$ `[1;2;3] q`) and (`sort`$^o$ `q [1;2;3]`) will give different answers. The first one gives sorted list `q = [1;2;3]`, the second one gives all permutations of list `[1;2;3]`.

All benchmarks were run ten times, and the average time was taken. For the naive fair conjunction we cherry-picked the best value of unfolding bound manually. For the fair conjunction based on structural recursion the bound was set to 100.

Fig. 9-12 show the results of evaluation in the form of bar charts. In the optimistic case, the results are similar for all semantics. In the pessimistic case the evaluation time of the directed conjunction rapidly increases, the evaluation time of the naive fair conjunction also increases, but not so much. The fair conjunction based on structural recursion demonstrates a similar efficiency as in the optimistic case.

[2]https://en.wikipedia.org/wiki/Bridge_and_torch_problem
[3]https://en.wikipedia.org/wiki/Water_pouring_puzzle

The results are presented in more detail in Fig. 13. “Hanoi Towers” solver has name `hanoi`$^o$, “Bridge and torch problem” solver has name `bridge`$^o$ and “Water pouring puzzle” solver has name `water`$^o$. We can conclude that forward and backward `sort`$^o$ runtime in the pessimistic case increases very rapidly with increasing the list length for directed and naive fair conjunctions. In the case of the fair conjunction based on structural recursion the running time in pessimistic case increases on a par with that in the optimistic one. Also the solver `water`$^o$ very slow in the pessimistic case for directed and naive fair. However, fair conjunction based on structural recursion pessimistic case is no different from an optimistic case.

| relation | size | directed conjunction | | naive fair conjunction | | structural recursion | |
|---|---|---|---|---|---|---|---|
| | | optimistic | pessimistic | optimistic | pessimistic | optimistic | pessimistic |
| forward revers$^o$ | 30 | 0.465 | 0.532 | 0.468 | 0.461 | 0.438 | 0.425 |
| | 60 | 0.579 | 0.828 | 0.577 | 0.658 | 0.545 | 0.450 |
| | 90 | 1.142 | 2.073 | 1.151 | 1.110 | 1.077 | 0.542 |
| backward revers$^o$ | 30 | 0.541 | 0.573 | 0.550 | 0.540 | 0.516 | 0.518 |
| | 60 | 0.637 | 0.867 | 0.642 | 0.805 | 0.636 | 0.637 |
| | 90 | 1.268 | 1.778 | 1.274 | 1.359 | 1.289 | 1.273 |
| forward sort$^o$ | 3 | 0.418 | 0.432 | 0.420 | 0.420 | 0.424 | 0.425 |
| | 4 | 0.424 | 3.924 | 0.424 | 0.455 | 0.429 | 0.429 |
| | 5 | 0.430 | >300 | 0.428 | 0.969 | 0.433 | 0.432 |
| | 6 | 0.434 | >300 | 0.430 | 11.577 | 0.434 | 0.437 |
| | 30 | 1.664 | >300 | 1.636 | >300 | 1.723 | 1.751 |
| backward sort$^o$ | 3 | 0.511 | 0.511 | 0.509 | 0.513 | 0.516 | 0.518 |
| | 4 | 0.525 | 0.823 | 0.534 | 0.539 | 0.534 | 0.530 |
| | 5 | 0.667 | 69.725 | 0.692 | 1.443 | 0.689 | 0.697 |
| | 6 | 2.880 | >300 | 2.891 | 56.107 | 2.921 | 2.936 |
| hanoi$^o$ | - | 1.574 | >300 | 1.579 | 17.604 | 1.585 | 1.646 |
| bridge$^o$ | - | 0.669 | 4.956 | 0.663 | 3.820 | 0.675 | 0.712 |
| water$^o$ | - | 3.132 | >300 | 3.168 | >300 | 3.220 | 3.414 |

Fig. 13. The results of evaluation: running times of benchmarks in seconds

To summarize, the fair conjunction based on structural recursion does not introduce any essential overhead in comparison with directed conjunction in an optimistic case. At the same time it weakly depends on the order of the conjuncts, and thus demonstrates much better performance in the pessimistic case.

## 7 CONCLUSION

In this paper we proposed a new approach for the execution of relational programs with structural recursion. This approach reduces the performance impact of conjunct order.

In the future we plan to generalize the proposed approach to a larger class of programs. We also plan to formalize our approach in Coq [2] proof assistant system and prove that the convergence of the execution does not depend on the order of the conjuncts.

## REFERENCES

[1] Claire E. Alvis, Jeremiah J. Willcock, and William E. Byrd. 2011. cKanren: miniKanren with Constraints. *Workshop on Scheme and Functional Programming* (2011).

[2] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[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. $\alpha$kanren: A Fresh Name in Nominal Logic Programming. *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming* (2007), 79–90.
[6] 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).
[7] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.
[8] Jason Hemann, Daniel Friedman, William Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. 96–107. https://doi.org/10.1145/2989225.2989230
[9] 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
[10] Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. Towards a miniKanren with fair search strategies. *The miniKanren and Relational Programming Workshop.*
[11] Joseph Near, William Byrd, and Daniel Friedman. 2008. $\alpha$leanTAP: A Declarative Theorem Prover for First-Order Classical Logic, Vol. 5366. 238–252. https://doi.org/10.1007/978-3-540-89982-2_26
[12] Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018.* 18:1–18:13. https://doi.org/10.1145/3236950.3236958
[13] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitri Boulytchev. 2019. Certified Semantics for miniKanren. *The miniKanren and Relational Programming Workshop.*
[14] Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren. *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming* (2013).

# An Empirical Study of Partial Deduction for miniKanren*

EKATERINA VERBITSKAIA, DANIIL BEREZUN, and DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We explore partial deduction, an advanced specialization technique aimed at improving the performance of a relation in the given direction, in the context of miniKanren. On several examples, we demonstrate issues which arise during partial deduction of relational programs. We describe a novel approach to specialization of miniKanren based on partial deduction and supercompilation. Although the proposed approach does not give the best results in all cases, we view it as a stepping stone towards the efficient optimization of miniKanren.

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

Additional Key Words and Phrases: relational programming, partial deduction, specialization

## 1 INTRODUCTION

The core feature of the family of relational programming languages miniKanren[1] is their ability to run a program in different directions. Having specified a relation for adding two numbers, one can also compute the subtraction of two numbers or find all pairs of numbers which can be summed up to get the given one. Program synthesis can be done by running *backwards* a relational interpreter for some language. In general, it is possible to create a solver for a recognizer by translating it into miniKanren and running in the appropriate direction [13].

The search employed in miniKanren is complete which means that every answer will be found, although it may take a long time. The promise of miniKanren falls short when speaking of performance. The running time of a program in miniKanren is highly unpredictable and varies greatly for different directions. What is even worse, it depends on the order of the relation calls within a program. One order can be good for one direction, but slow down the computation drastically in the other direction.

Specialization or partial evaluation [7] is a technique aimed at improving the performance of a program given some information about it beforehand. It may either be a known value of some argument, its structure (i.e. the length of an input list) or, in case of a relational program, — the direction in which it is intended to be run. An earlier paper [13] showed that *conjunctive partial deduction* [3] can sometimes improve the performance of miniKanren programs. Unfortunately, it may also not affect the running time of a program or even make it slower.

Control issues in partial deduction of logic programming language Prolog have been studied before [11]. The ideas described there are aimed at left-to-right evaluation strategy of Prolog. Since the search in miniKanren is complete, it is safe to reorder some relation calls within the goal ahead-of-time for better performance. While

*The reported study was funded by RFBR, project number 18-01-00380

[1] miniKanren language web site: http://minikanren.org. Access date: 17.07.2020.

Authors' address: Ekaterina Verbitskaia, kajigor@gmail.com; Daniil Berezun, daniil.berezun@jetbrains.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University, Russia , JetBrains Research, Russia.


miniKanren.org/workshop/2021/8-ART9

sometimes conjunctive partial deduction gives great performance boost, sometimes it does not behave as well as it could have.

In this paper, we show on examples some issues which conjunctive partial deduction faces. We also describe *conservative partial deduction* — a novel specialization approach for the relational programming language miniKanren. We compare it to the existing specialization algorithms on several programs and discuss why some miniKanren programs run slower after specialization.

## 2 RELATED WORK

Specialization is an attractive technique aimed to improve the performance of a program if some of its arguments are known statically. Specialization is studied for functional, imperative and logic programing and comes in different forms: partial evaluation [7] and partial deduction [12], supercompilation [15], distillation [5], and many more.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is a *process tree* where nodes correspond to *configurations* which represent computation state. For example, in the case of pure functional programming languages, the computational state might be a term. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variables' values, equalities and disequalities, and precomputing of all deterministic semantic evaluation steps. The latter process, also known as *deforestation*, means combining of consecutive process tree nodes with no branching. When the tree is constructed, the resulting, or *residual*, program can be extracted from the process tree by the process called *residualization*. Of course, process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure supercompilation termination. If a whistle signals during the construction of some branch, then something should be done to ensure termination. The most common approaches are either to stop driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or to fold the process tree to a *process graph*. The main instrument to perform such a folding is *generalization*. Generalization, abstracting away some computed data about the current term, makes folding possible. One source of infinite branches is consecutive recursive calls to the same function with an accumulating parameter: by unfolding such a call further one can only increase the term size which leads to nontermination. The accumulating parameter can be removed by replacing the call with its generalization. There are several ways to ensure process correctness and termination, most-specific generalization (anti-unification) and *homeomorphic embedding* [6, 9] as a whistle being the most common.

While supercompilation generally improves the behaviour of input programs and distillation can even provide superlinear speedup, there are no ways to predict the effect of specialization on a given program in the general. What is worse, the efficiency of residual program from the target language evaluator point of view is rarely considered in the literature. The main optimization source is computing in advance all possible intermediate and statically-known semantics steps at program transformation-time. Other criteria, like the size of the generated program or possible optimizations and execution cost of different language constructions by the target language evaluator, are usually out of consideration [7]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [1, 8] and cause code explosion [14]. Moreover, it may be hard to predict the real speedup of any given program on concrete examples even disregarding the problems above because of the complexity of the transformation algorithm. The worst-case for partial evaluation is when all static variables are used in a dynamic context, and there is some advice on how to implement a partial evaluator as well as a target program so that specialization indeed improves its performance [2, 7]. There is a lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [3] makes an effort to provide reasonable control for the left-to-right evaluation strategy of Prolog. CPD constructs a tree which models goal evaluation and is similar to a SLDNF tree, then a residual program is generated from this tree. Partial deduction itself resembles driving in supercompilation [4]. The specialization is done in two levels of control: the local control determines the shape of the residual programs, while the global control ensures that every relation which can be called in the residual program is indeed defined. The leaves of local control trees become nodes of the global control tree. CPD analyses these nodes at the global level and runs local control for all those which are new.

At the local level, CPD examines a conjunction of atoms by considering each atom one-by-one from left to right. An atom is *unfolded* if it is deemed safe, i.e. a whistle based on homeomorphic embedding does not signal for the atom. When an atom is unfolded, a clause whose head can be unified with the atom is found, and a new node is added into the tree where the atom in the conjunction is replaced with the body of that clause. If there is more than one suitable head, then several branches are added into the tree which corresponds to the disjunction in the residualized program. An adaptation of CPD for the miniKanren programming language is described in [13].

The most well-behaved strategy of local control in CPD for Prolog is *deterministic unfolding* [10]. An atom is unfolded only if precisely one suitable clause head exists for it with the single exception: it is allowed to unfold an atom non-deterministically once for one local control tree. This means that if a non-deterministic atom is the leftmost within a conjunction, it is most likely to be unfolded and to introduce many new relation calls within the conjunction. We believe this is the core problem with CPD which limits its power when applied to miniKanren. The strategy of unfolding atoms from left to right is reasonable in the context of Prolog because it mimics the way programs in Prolog execute. But it often leads to larger global control trees and, as a result, bigger, less efficient programs. The evaluation result of a miniKanren program does not depend on the order of atoms (relation calls) within a conjunction, thus we believe a better result can be achieved by selecting a relation call which can restrict the number of branches in the tree. We describe our approach which implements this idea in the next section.

## 3 CONSERVATIVE PARTIAL DEDUCTION

In this section, we describe a novel approach to relational programs specialization. This approach draws inspiration from both conjunctive partial deduction and supercompilation. The aim was to create a specialization algorithm which is simpler than conjunctive partial deduction and uses properties of miniKanren to improve the performance of the input programs.

The algorithm pseudocode is shown in Fig. 1. For the sake of brevity and clarity, we provide functions `drive_disj` and `drive_conj` which describe how to process disjunctions and conjunctions respectively. Driving itself is a trivial combination of the functions provided (line 2).

A driving process creates a process tree, from which a residual program is later created. The process tree is meant to mimic the execution of the input program. The nodes of the process tree include a *configuration* which describes the state of program evaluation at some point. In our case a configuration is a conjunction of relation calls. The substitution computed at each step is also stored in the tree node, although it is not included in the configuration.

Hereafter, we consider all goals and relation bodies to be in *canonical normal form* — a disjunction of conjunctions of either calls or unifications. Moreover, we assume all fresh variables to be introduced into the scope and all unifications to be computed at each step. Those disjuncts in which unifications fail are removed. Each other disjunct takes the form of a possibly empty conjunction of relation calls accompanied with a substitution computed from unifications. Any miniKanren term can be trivially transformed into the described form. In Fig. 1 the function `normalize` is assumed to perform term normalization. The code is omitted for brevity.

```
1  ncpd goal = residualize ∘ drive ∘ normalize (goal)
2  drive     = drive_disj ∪ drive_conj
3
4  drive_disj :: Disjunction → Process_Tree
5  drive_disj D@(c_1, ..., c_n) = ⋁_{i=1}^{n} t_i ← drive_conj (c_i)
6
7  drive_conj :: (Conjunction, Substitution) → Process_Tree
8  drive_conj ((r_1, ..., r_n), subst) =
9    C@(r_1, ..., r_n) ← propagate_substitution subst on r_1, ..., r_n
10   case whistle (C) of
11   | instance (C', subst')      ⇒ create_fold_node (C', subst')
12   | embedded_but_not_instance ⇒ create_stop_node (C , subst )
13   | otherwise ⇒
14   | | case heuristically_select_a_call (r_1, ..., r_n) of
15   | | | Just r ⇒
16   | | | | t ← drive ∘ normalize ∘ unfold (r)
17   | | | | if trivial ∘ leafs (t)
18   | | | | then
19   | | | | | C' ← propagate_substitution (C \ r, extract_substitution (t))
20   | | | | | drive C'[r ↦ extract_calls (t)]
21   | | | | else
22   | | | | | t ⋀ drive (C \ r, subst)
23   | | | Nothing ⇒ ⋀_{i=1}^{n} t_i ← drive ∘ normalize ∘ unfold (r_i)
```

Fig. 1. Conservative Partial Deduction Pseudo Code

There are several core ideas behind this algorithm. The first is to select an arbitrary relation to unfold, not necessarily the leftmost which is safe. The second idea is to use a heuristic which decides if unfolding a relation call can lead to discovery of contradictions between conjuncts which in turn leads to restriction of the answer set at specialization-time (line 14; `heuristically_select_a_call` stands for heuristics combination, see section 3.2 for details). If those contradictions are found, then they are exposed by considering the conjunction as a whole and replacing the selected relation call with the result of its unfolding thus *joining* the conjunction back together instead of using *split* as in CPD (lines 15–22). Joining instead of splitting is why we call our transformer *conservative* partial deduction. Finally, if the heuristic fails to select a potentially good call, then the conjunction is split into individual calls which are driven in isolation and are never joined (line 23).

When the heuristic selects a call to unfold (line 15), a process tree is constructed for the selected call *in isolation* (line 16). The leaves of the computed tree are examined. If all leaves are either computed substitutions or are instances of some relations accompanied with non-empty substitutions, then the leaves are collected and each of them replaces the considered call in the root conjunction (lines 19–20). If the selected call does not suit the criteria, the results of its unfolding are not propagated onto other relation calls within the conjunction, instead, the next suitable call is selected (line 22). According to the denotational semantics of miniKanren it is safe to compute individual conjuncts in any order, thus it is ok to drive any call and then propagate its results onto the other calls.

This process creates branchings whenever a disjunction is examined (lines 4–5). At each step, we make sure that we do not start driving a conjunction which we have already examined. To do this, we check if the current conjunction is a renaming of any other configuration in the tree (line 11). If it is, then we fold the tree by creating a special node which then is residualized into a call to the corresponding relation.

In this approach, we decided not to generalize in the same fashion as CPD or supercompilation. Our conjunctions are always split into individual calls and are joined back together only if it meaningful. If the need for generalization

arises, i.e. homeomorphic embedding of conjunctions [3] is detected, then we immediately stop driving this conjunction (line 12). When residualizing such a conjunction, we just generate a conjunction of calls to the input program before specialization.

### 3.1 Unfolding

Unfolding in our case is done by substitution of some relation call by its body with simultaneous normalization and computation of unifications. The unfolding itself is straightforward however it is not always clear what to unfold and when to *stop* unfolding. Unfolding in the specialization of functional programming languages, as well as inlining in imperative languages, is usually considered to be safe from the residual program efficiency point of view. It may only lead to code explosion or code duplication which is mostly left to a target program compiler optimization or even is out of consideration at all if a specializer is considered as a standalone tool [7].

Unfortunately, this is not the case for the specialization of a relational programming language. Unlike in functional and imperative languages, in logic and relational programming languages unfolding may easily affect the target program's efficiency [11]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the unfolding's results onto neighbouring conjuncts.

There is a fine edge between too much unfolding and not enough unfolding. The former is maybe even worse than the latter. We believe that the following heuristic provides a reasonable approach to unfolding control.

### 3.2 Less-Branching Heuristic

This heuristic is aimed at selecting a relation call within a conjunction which is both safe to unfold and may lead to discovering contradictions within the conjunction. An unsafe unfolding leads to an uncontrollable increase in the number of relation calls in a conjunction. It is best to first unfold those relation calls which can be fully computed up to substitutions.

We deem every static (non-recursive) conjunct to be safe because they never lead to growth in the number of conjunctions. Those calls which unfold deterministically, meaning there is only one disjunct in the unfolded relation, are also considered to be safe.

Those relation calls which are neither static nor deterministic are examined with what we call the *less-branching* heuristic. It identifies the case when the unfolded relation contains fewer disjuncts than it could possibly have. This means that we found some contradiction, some computations were gotten rid of, and thus the answer set was restricted, which is desirable when unfolding. To compute this heuristic we precompute the maximum possible number of disjuncts in each relation and compare this number with the number of disjuncts when unfolding a concrete relation call. The maximum number of disjuncts is computed by unfolding the body of the relation in which all relation calls were replaced by a unification which always succeeds.

```
1 | heuristically_select_a_call :: Conjunction → Maybe Call
2 | heuristically_select_a_call C = find heuristic C
3 |
4 | heuristic :: Call → Bool
5 | heuristic r = isStatic r || isDeterministic r || isLessBranching r
```

Fig. 2. Heuristic selection pseudocode

The pseudocode describing our heuristic is shown in Fig. 2. Selecting a good relation call can fail (line 1). The implementation works such that we first select those relation calls which are static, and only if there are none, we proceed to consider deterministic unfoldings and then we search for those which are less branching. We believe this heuristic provides a good balance in unfolding.

## 4 EVALUATION

We implemented the new conservative partial deduction[2] and compared it with the CPD adaptation for miniKanren of [13]. We have also employed the branching heuristic instead of the deterministic unfolding in the CPD to check whether it can improve the quality of the specialization.

We used the following programs to test the specializers.

- Two implementations of an evaluator of logic formulas.
- A program to compute a unifier of two terms.
- A program to search for paths of a specific length in a graph.

The last two relations are described in [13] thus we will not describe them here.

### 4.1 Evaluator of Logic Formulas

The relation `eval`$^o$ describes an evaluation of a subset of first-order logic formulas in a given substitution. It has 3 arguments. The first argument is a list of boolean values which serves as a substitution. The *i*-th value of the substitution is the value of the *i*-th variable. The second argument is a formula with the following abstract syntax. A formula is either a *variable* represented with a Peano number, a *negation* of a formula, a *conjunction* of two formulas or a *disjunction* of two formulas. The third argument is the value of the formula in the given substitution.

All examples of miniKanren relations in this paper are written in OCanren[3] syntax. We specialize the `eval`$^o$ relation to synthesize formulas which evaluate to $\uparrow$**true**[4]. To do so, we run the specializer for the goal with the last argument fixed to $\uparrow$**true**, while the first two arguments remain free variables. Depending on the way the `eval`$^o$ is implemented, different specializers generate significantly different residual programs.

*4.1.1 The Order of Relation Calls.* One possible implementation of the evaluator in the syntax of OCanren is presented in Listing 1. Here the relation `elem`$^o$ `subst v res` unifies `res` with the value of the variable `v` in the list `subst`. The relations `and`$^o$, `or`$^o$, and **not**$^o$ encode corresponding boolean operations.

```
let rec eval^o subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v ∧ elem^o subst v res);
    (fm ≡ conj x y ∧ eval^o st x v ∧ eval^o st y w ∧ and^o v w res);
    (fm ≡ disj x y ∧ eval^o st x v ∧ eval^o st y w ∧ or^o v w res);
    (fm ≡ neg x ∧ eval^o st x v ∧ not^o v res))]
```

Listing 1. Evaluator of formulas with boolean operation last

Note, that the calls to boolean relations `and`$^o$, `or`$^o$, and **not**$^o$ are placed last within each conjunction. This poses a challenge to the CPD-based specializers. Conjunctive partial deduction unfolds relation calls from left to right, so when specializing this relation for running backwards (i.e. considering the goal `eval`$^o$ `subst fm` $\uparrow$**true**), it fails to propagate the direction data onto recursive calls of `eval`$^o$. Knowing that `res` is $\uparrow$**true**, we can conclude that in the call `and`$^o$ `v w res` variables `v` and `w` have to be $\uparrow$**true** as well. There are three possible options for these variables in the call `or`$^o$ `v w res` and one for the call **not**$^o$. These variables are used in recursive calls of

[2]The repository of the miniKanren specialization project: https://github.com/kajigor/uKanren_transformations. Access date: 17.07.2020.
[3]OCanren: statically typed miniKanren embedding in OCaml. The repository of the project: https://github.com/JetBrains-Research/OCanren. Access date: 17.07.2020.
[4]An arrow lifts ordinary values to the logic domain.

$\texttt{eval}^o$ and thus restrict the result of driving them. CPD fails to recognize this, and thus unfolds recursive calls of $\texttt{eval}^o$ applied to fresh variables. It leads to over-unfolding, big residual programs and poor performance.

The conservative partial deduction first unfolds those calls which are selected with the heuristic. Since exploring boolean operations makes more sense, they are unfolded before recursive calls of $\texttt{eval}^o$. The way conservative partial deduction treats this program is the same as it treats the other implementation in which boolean operations are moved to the left, as shown in Listing 2. This program is easier for CPD to transform which demonstrates how unequal the behaviour of CPD for similar programs is.

```
let rec eval^o subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v ∧ elem^o subst v res);
    (fm ≡ conj x y ∧ and^o v w res ∧ eval^o st x v ∧ eval^o st y w);
    (fm ≡ disj x y ∧ or^o v w res ∧ eval^o st x v ∧ eval^o st y w);
    (fm ≡ neg x ∧ not^o v res ∧ eval^o st x v))]
```

Listing 2. Evaluator of formulas with boolean operation second

*4.1.2 Unfolding of Complex Relations.* Depending on the way a relation is implemented, it may take a different number of driving steps to reach the point when any useful information is derived through its unfolding. Partial deduction tries to unfold every relation call unless it is unsafe, but not all relation calls serve to restrict the search space and thus not every relation call should be unfolded. In the implementation of $\texttt{eval}^o$ boolean operations can effectively restrict variables within the conjunctions and should be unfolded until they do. But depending on the way boolean operations are implemented, different number of driving steps should be performed for that. The simplest way to implement these relations is with a table as demonstrated with the implementation of $\textbf{not}^o$ in Listing 3. It is enough to unfold such relation calls once to derive useful information about variables.

```
let not^o x y = conde [
   (x ≡ ↑true ∧ y ≡ ↑false;
    x ≡ ↑false ∧ y ≡ ↑true)]
```

Listing 3. Implementation of boolean **not** as a table

The other way to implement boolean operations is via one basic boolean relation such as $\texttt{nand}^o$ which has, in turn, a table-based implementation (see Listing 4). It will take several sequential unfoldings to derive that variables `v` and `w` should be $\uparrow\textbf{true}$ when considering a call $\texttt{and}^o$ `v w` $\uparrow\textbf{true}$ implemented via a basic relation. Conservative partial deduction drives the selected call until it derives useful substitutions for the variables involved. CPD with deterministic unfolding may fail to derive useful substitutions.

## 4.2 Evaluation Results

In our study, we considered two implementations of $\texttt{eval}^o$, one we call `plain` and the other — `last`, and compared how specializers behave on them. The `plain` relation uses table-based boolean operations and places them further to the left in each conjunction. The relation `last` employs boolean operations implemented via $\texttt{nand}^o$ and places them at the end of each conjunction. These two programs are complete opposites from the standpoint of CPD.

We measured the time necessary to generate 1000 formulas over two variables which evaluate to $\uparrow\textbf{true}$. We compared the results of specialization of the goal $\texttt{eval}^o$ `subst fm` $\uparrow\textbf{true}$ by our implementation of CPD, the

```
let not^o x y = nand^o x x y

let or^o x y z = nand^o x x xx ∧nand^o y y yy ∧nand^o xx yy z

let and^o x y z = nand^o x y xy ∧nand^o xy xy z

let nand^o a b c = conde [
  ( a ≡ ↑false ∧ b ≡ ↑false ∧ c ≡ ↑true );
  ( a ≡ ↑false ∧ b ≡ ↑true ∧ c ≡ ↑true );
  ( a ≡ ↑true ∧ b ≡ ↑false ∧ c ≡ ↑true );
  ( a ≡ ↑true ∧ b ≡ ↑true ∧ c ≡ ↑false )]
```

Listing 4. Implementation of boolean operations via nand

| | last | plain | unify | isPath |
|---|---|---|---|---|
| Original | 1.06s | 1.84s | — | — |
| CPD | — | 1.13s | 14.12s | 3.62s |
| ConsPD | 0.93s | 0.99s | 0.96s | 2.51s |
| Branching | 3.11s | 7.53s | 3.53s | 0.54s |

Table 1. Evaluation results

new conservative partial deduction, and the CPD modified with the less-branching heuristic. Our evaluation confirmed that CPD behaves very differently on these two implementations of the same relation. CPD improves the execution time of the `plain` relation, however CPD performs too much unfolding of the `last` relation which is why the specialized relation `last` fails to terminate in under 10 seconds. The execution time of two programs generated with the novel conservative partial deduction is very similar and it is a little bit better than the best by CPD. CPD with the less-branching heuristic constructs residual programs of different quality, worsening the execution time for both implementations. The results are shown in table 1.

Besides the evaluator of logic formulas we also run the transformers on the relation `unify`, which searches for a unifier of two terms, and the relation `isPath` specialized to search for paths in a graph. These two relations are described in paper [13] so we will not go into too many details here.

The `unify` relation was executed to find a unifier of the terms $f(X, X, g(Z, t))$ and $f(g(p, L), Y, Y)$. The original miniKanren program fails to terminate on this goal in 30 seconds. On this example, the most performant is the program generated by conservative partial deduction (0.96 seconds).

The last test executed the `isPath` relation to search for 5 paths in a graph with 20 vertices and 30 edges. The original miniKanren program fails to terminate on this goal in 30 seconds. On this program, CPD with branching heuristic showed much better transformation result than both CPD and conservative partial deduction, although all specialized versions show improvement as compared with the original relation.

All evaluation results are presented in the table 1. Each column corresponds to the relation being run as described above. The row marked “Original” contains the execution time of the original miniKanren relation before specialization, “CPD” and “ConsPD” correspond to conjunctive and conservative partial deduction respectively while “Branching” is for the CPD modified with the branching heuristic.

## 5 CONCLUSION

In this paper, we discussed some issues which arise in partial deduction of a relational programming language, miniKanren. We presented a novel approach to partial deduction which uses less-branching heuristic to select the most suitable relation call to unfold at each step of driving. We compared this approach to the earlier implementation of conjunctive partial deduction and the implementation of CPD equipped with the new less-branching heuristic.

The conservative partial deduction improved the execution time of all relations while the implementations of CPD degraded the performance of some of them. However, CPD equipped with the less-branching heuristic improved the execution time of one relation the most compared with the other specializers. We conclude that there is still no one good technique which definitely speeds up every relational program. More research is needed to develop models to predict performance of relations, these models can further be used in specialization.

## REFERENCES


[1] Maximilian C. Bolingbroke and Simon L. Peyton Jones. 2010. Supercompilation by evaluation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, Jeremy Gibbons (Ed.). ACM, 135–146. https://doi.org/10.1145/1863523.1863540

[2] Mikhail A. Bulyonkov. 1984. Polyvariant Mixed Computation for Analyzer Programs. *Acta Inf.* 21 (1984), 473–484. https://doi.org/10.1007/BF00271642

[3] 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.

[4] 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.

[5] Geoff W Hamilton. 2007. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 61–70.

[6] G. Higman. 1952. Ordering by divisibility in abstract algebras. In *Proceedings of the London Mathematical Society*, Vol. 2. 326–336.

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

[8] Peter A. Jonsson and Johan Nordlander. 2011. Taming code explosion in supercompilation. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, Siau-Cheng Khoo and Jeremy G. Siek (Eds.). ACM, 33–42. https://doi.org/10.1145/1929501.1929507

[9] J. B. Kruskal. 1960. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.* 95, 210–225.

[10] Michael Leuschel. 1997. Advanced techniques for logic program specialisation. (1997).

[11] Michael Leuschel and Maurice Bruynooghe. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming* 2, 4-5 (2002), 461–515.

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

[13] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational Interpreters for Search Problems. In *miniKanren and Relational Programming Workshop*. 43.

[14] Neil Mitchell and Colin Runciman. 2007. A Supercompiler for Core Haskell. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5083)*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, 147–164. https://doi.org/10.1007/978-3-540-85373-2_9

[15] Morten Heine Soerensen, Robert Glück, and Neil D. Jones. 1996. A positive supercompiler. *Journal of functional programming* 6, 6 (1996), 811–838.

# MicroKanren in J: an Embedding of the Relational Paradigm in an Array Language with Rank-Polymorphic Unification

RAOUL SCHORER, Geneva University Hospitals, Switzerland

This work presents a reimplementation within the array programming paradigm of MicroKanren, a minimal relational language designed to be embedded in host languages as a library. The particularities of the implementation relative to the array paradigm are detailed. The implementation is discussed in relation to the reference Racket version and a rank-polymorphic unification algorithm is presented. Those elements are discussed in light of future prospects.

CCS Concepts: • **Computing methodologies** → *Logic programming and answer set programming*; • **Software and its engineering** → **Language types**.

Additional Key Words and Phrases: array programming, logic programming, microKanren, miniKanren, relational programming, streams, unification

## INTRODUCTION

Logic programming allows a high level of abstraction in both code and problem solving concepts. Language representatives of this paradigm of which Prolog is the most well-known traditionally aim at a declarative approach and are remarkably versatile [21]. MiniKanren and its even more minimalist sibling language MicroKanren offer an alternative and distinct tradeoffs with similar capabilities in the context of relational programming [6, 9]. MicroKanren semantics are designed to be easily transcribed to a variety of host languages, to the point that implementing a shallow embedding of Micro- or MiniKanren as a library has become a rite of passage for newcomers to the Kanren language family. The purely functional semantics of MicroKanren translates to a great variety of programming paradigms with little change in general syntax and operators, yielding a somewhat portable notation for relational programs [5]. It is also interesting that shallow embedding allows Micro/MiniKanren to inherit host language characteristics. A notable illustration of this principle can be found in OCanren, an embedding of Minikanren that cooperates with the underlying OCaml type system [20]. It is therefore worthwile to explore other well-chosen host languages and their interaction with the relational paradigm. This work presents an implementation of MicroKanren in the J array language and discusses the resulting features in relation to a reference implementation in Racket as described by Hemann *et al.* [12]. We begin by a short presentation of the array language J and the features that motivated its use. Next, we describe the main points of the implementation of MicroKanren on top of J. We then examine example programs and benchmarks. In the final section, we discuss related works as well as limitations and future prospects before concluding. The main contributions of this paper are:

(1) to show that array rank operators allow flexible unification over multidimensional arrays.
(2) to provide a prototype Kanren implementation based exclusively on arrays with the long term goals of
   - exploring adaptations of the logic programming paradigm to vector processing.

Author's address: Raoul Schorer, Geneva University Hospitals, Switzerland, raoul.schorer@hcuge.ch.


miniKanren.org/workshop/2021/8-ART3

- developing a logic language hosted on vector platforms such as graphical processor units (GPU) or that can make good use of single instruction multiple data (SIMD) instruction sets.

## CHOOSING J

The array language J is a sibling of APL restricting its character set to ASCII. Interested readers unfamiliar with J may find introductions and tutorials on the website (jsoftware.com), which will enable them to understand the code below. Additionally, the J dictionary (jsoftware.com/wiki/nuvoc) contains accessible descriptions and examples applications of all language primitives. Array languages are characterized by a mostly-pure functional programming paradigm with severe syntactic constraints, and J is no exception. The power of array languages resides in their vast collection of primitives usually implemented in a low-level language allowing high-level array manipulation at speed. While J has most of the features that make a good Kanren host language (garbage collection, object orientation, metaprogramming through quotation), some syntactic features such as the strict limit of two arguments per function make the implementation less straightforward [10]. Below, we describe some features that were most helpful in the implementation of MicroKanren in J.

*Rank Polymorphism.* Rank is a crucial concept in J and underlies the contributions made by this paper. The rank of an array is defined as its number of dimensions. In an attempt at improved practicality, primitives were made rank-polymorphic in implementations of some array languages such as J, thereby allowing the user to choose the dimension along which a procedure is to be applied. Conceptually, rank is closely related to looping constructs and emulates iteration over a particular dimension in the traditional nested loop program flow typical of ALGOL-inspired languages. J boxing procedures allow a clear illustration of this concept. Starting from a flat array of letters:

```
a. {~ 65 + i. 26
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The rank of this array is 1, as shown by counting its dimensions.

```
# $ a. {~ 65 + i. 26
1
```

Now, we have two available ranks for application: 0 (scalar) and 1. < and " are respectively the boxing and rank operators. Notice the difference in boxing pattern when using different ranks:

```
<"0 a. {~ 65 + i. 26
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
<"1 a. {~ 65 + i. 26
+--------------------------+
|ABCDEFGHIJKLMNOPQRSTUVWXYZ|
+--------------------------+
```

Similarly, this concept extends to any number of dimensions:

```
2 3 $ i. 6
0 1 2
3 4 5
<"1 [ 2 3 $ i. 6
+-----+-----+
|0 1 2|3 4 5|
+-----+-----+
<"2 [ 2 3 $ i. 6
+-----+
|0 1 2|
|3 4 5|
+-----+
```

This feature is extensively used throughout our implementation and forms the basis of a rank-polymorphic unification procedure that will be described below.

*Metaprogramming.* We use J's first-class quote and eval operators to delay and force evaluation. An internal value can be automatically converted to its string representation by the runtime using the `(5!:5)` function, while its counterpart `".` evaluates J expressions in string form. Unlike Racket which has to interact with the lexical context to recover variable bindings when using `eval`, J metaprogramming is much more primitive and done purely through evaluation of strings without any notion of the surrounding environment.

*Fixed Point and Recursion.* The `^:_` primitive allows recursion up to a fixed point, which is critical in our unification algorithm implementation heavily inspired by Robinson's classic work [25]. J does not provide tail-call optimization and recursion may result in stack overflow. While transforming the reference implementation to use iteration instead of recursion is an option, the J `F.` operator family provides very flexible folding capabilities used to circumvent restrictions on recursion.

*Tree Manipulation.* J differs from its more popular sibling APL by providing tree processing primitives. Tree manipulation in array languages is considered a major weakness, but this has recently improved significantly and has become a point of major research interest within the array language community [16]. J's preferred tree representation is nested arrays of boxed elements. The `L:` and `S:` primitives perform left pre-order tree traversal and respectively apply a function in place or apply and collect leaves up to a user-defined tree height. The intrinsic depth-first search capabilities of those primitives are of particular interest in the context of logic programming. The primitive `{::` forms the basis of our unification algorithm proper (see below). `{::` takes a J tree and returns a tree of identical shape where leaves have been replaced by paths to those leaves in left preorder traversal, as exposed below:

```
('a' ;< 'c';'d');<('b' ;< 'e';'f')
```

```
+---------+---------+
|+-+-----+|+-+-----+|
||a|+-+-+|||b|+-+-+||
|| ||c|d|||| ||e|f|||
|| |+-+-+||| |+-+-+||
|+-+-----+|+-+-----+|
+---------+---------+
```

```
{:: ('a' ;< 'c';'d');<('b' ;< 'e';'f')
```

```
+-------------------------+-------------------------+
|+-----+-----------------+|+-----+-----------------+|
||+-+-+|+-------+-------+|||+-+-+|+-------+-------+||
|||0|0|||+-+-+-+|+-+-+-+|||||1|0|||+-+-+-+|+-+-+-+|||
||+-+-+|||0|1|0|||0|1|1|||||+-+-+|||1|1|0|||1|1|1||||
||     ||+-+-+-+|+-+-+-+||||     ||+-+-+-+|+-+-+-+|||
||     |+-------+-------+|||     |+-------+-------+||
|+-----+-----------------+|+-----+-----------------+|
+-------------------------+-------------------------+
```

## MICROKANREN IMPLEMENTATION

This section describes the J implementation of the core MicroKanren operators with some MiniKanren extensions. We modified these operators' names to accordance with J's restrictions on identifiers (table 1).

Table 1. J procedure names mapping to the reference implementation.

| Scheme | J |
|---|---|
| == | equ |
| call/fresh | fsh |
| disj | dis |
| conj | con |
| append | app |
| append-map | apm |
| var? | var |
| find | get |
| occurs? | occ |
| ext-s | ext |
| unify | uni |
| run | run |

The reference implementation `define-relation` operator is replaced by quotation at the top of every relation definition as described in Hemann *et al.* [12]. Just like for the Racket version, variables are defined as scalar natural number values. In this section we discuss some of the implementation's more interesting portions. The term "list" in the context of our implementation denotes a flat array. The full code is available in appendix A at

the end of this paper as well as online. [1] The J version used was j901/j64avx2/linux release e, with library version 9.01.23.

## Variable Scoping and Substitution

MicroKanren uses a substitution table for representing the *most general unifier* from mathematical logic. Ideally, this should be implemented using a hash table or a datastructure with similar insertion and retrieval characteristics although association lists may be used instead. Associative arrays are traditional and often used in APL descendents. However, we will use a flat boxed array as described in Brown & Guerreiro for our substitutions [3]. First, this removes some levels of indirections in our code when walking the substitution and follows the minimalistic spirit of MicroKanren. Second and most importantly, variable scoping is defined by the length of the substitution array as the semi-open interval [0 ; length), as below.

```
fsh =: ,<@#
fsh 3 ; 1 ; 2 ; 'a'
+-+-+-+-+
|3|1|2|a|4|
+-+-+-+-+
```

The above code block shows the application of `fsh` to an array of length 4. When using this array as a substitution list, scoping rules respectively allow variables 0 through 3 in the original state and 0 through 4 after extension by `fsh`. Position 0 contains the value 3 and points to this location in the list, which itself contains the character *a*. Variable 0 is therfore bound to the value *a*. Variables 1 and 2 are not bound and hence point to themselves. While this system works well in practice, it does not respect lexical scoping since the introduction of a variable causes it to stay in scope wherever its associated substitution table is found. For example, the variables in scope will be passed along when the substitution is returned from a procedure.

## Unification

The Racket version of unification relies on tree recursion, which is an obvious choice in this context. Since J does not offer advanced recursion facilities, we instead reuse the tree primitives depth-first search characteristics to yield an iterative algorithm in an attempt to compensate for J's weaknesses.

*Walking The Substitution.* `get` searches for the value of a variable in a substitution. Similarly to the reference implementation, `get` is the identity function in case the argument is not found in the substitution. The definition uses the `u^:v y` control flow operator where the predicate `v` passes its argument `y` to the function `u` on success. In case of failure, the expression is the identity function and `y` is returned. This gives us "backtracking for free" in a single operator. `{::` retrieves the selected element from the substitution. This pointer-directed walk across the substitution array is repeated until a fixed point is reached using `^:_`, thereby allowing a terse expression of our desired behaviour.

```
get =: {::~^:(var@])"(_ 0)^:_
```

*The Occurs Check.* The occurs check is non-optional in Kanren-style unification. This is again done using tree recursion in Racket. Here, we rely on J's `S:` operator to avoid writing the tree traversal ourselves. The `occ` predicate is implemented by first checking whether the top-level elements to be compared are identical, in which case we immediately fail since by definition identical terms do not include each other. We then traverse the tree and flatten it to a list of leaves using `< S: 0`. Membership is then tested using `e.`.

```
occ =: (e. < S: 0)`0:@.-:
```

[1] [https://github.com/Rscho314/mk]

*Extending The Substitution.* The orginal Racket code performs an occurs check and extends the state on failure of the predicate. `ext` is more involved because it operates on the collected set of candidate substitutions it receives from `uni`, using the largest possible fragment of the candidate substitution set on each iteration. Due to our non-lexical scoping system, we have to verify whether the substitution already contains incompatible values and abort if such is the case. This is done by grounding the variables in the tree `y` according to the existing substitution list `x` (`ext`, line 4). `_1` is the error code value (`ext`, line 8). Additionally, duplicate subterms have to be filtered out (`ext`, line 5). Iterations of the while loop insert ground values into the substitution and attempt to ground remaining variables until only free variables (or nothing) remains. This is done by following a chain of pointers from the substitution as mentioned above (`ext`, lines 10-15). Finally, the remaining free variables are inserted into the substitution (`ext`, line 17). The extension of the substitution list involves copying and no sharing of the substitution table occurs, unlike Racket where the runtime can use structure-sharing to preserve space-efficiency. In principle nothing forbids similar sharing in array languages, but most representatives of the paradigm use mutable arrays and rely on interpreters of relatively simple structure to perform optimizations through array-specific instructions such as single instruction multiple data (SIMD) intrinsics.

```
 1  ext =: 4 : 0
 2   while. 1
 3    do.
 4     y =. x&get L:0 y
 5     y =. ~. (#~ (i.@# < i.&1"1@(-:"1/ |."1)~))~ y
 6     if. '' -: y do. x return. end.
 7     isvar =. > var &.> y
 8     if. +./ (occ/"1 y) , (+:/"1 isvar) do. _1 return. end.
 9     candidates =. (*:/"1 isvar) # y {"1~ |."1 isvar
10     if. '' -.@-: , candidates
11      do.
12       x =. ({:"1 candidates) (>@{."1 candidates) } x
13       y =. y -. candidates
14     else. break.
15     end.
16   end.
17   if. '' -.@-: , y do. ({:"1 y) (>@{."1 y) } x  else. x end.
18  )
```

*Unification.* Rank polymorphism in unification is implemented by boxing arguments `u` and `v` according to the left (`{.x`) and right ranks (`{:x`) (`uni`, lines 4-5). This step is actually independent of the rest of the algorithm which could be replaced by a totally different one without impacting rank polymorphism capabilities. Regarding unification proper, the key to unification using mostly flat array operations is the `{::` (map) J primitive. The explanations and examples regarding `> "` and `{::` found above ("Choosing J" section) are relevant here. `{::` allows us to partition the path tree between variables, corresponding substitutions and residual paths. The path tree is built from `u` (left) and `v` (right) subtrees and flattened to a list of paths using `< S: 1` (`uni`, line 7-8). We also flatten the original tree to a list of leaves with `S: 0` which allows us to create a boolean mask and extract variables and substitutions paths in `varpaths_candidates` and `substpaths_candidates` (`uni`, line 9-10). Substitution paths are obtained by boolean negation of the first path element of `varpaths_candidates`, which points to the corresponding leaf in the opposing subtree (left or right). This avoids traversing the full tree twice (`uni`, line 10) but some of the resulting paths may not actually exist. Those spurious paths stem from variables having no substitutions, *i.e.* free variables. Non-existing paths are identified by checking whether they exist in the set of all

path prefixes and free-variable-spurious-substitution pairs are removed if not found (`uni`, lines 11-13). Filtering free variables yields the `varpaths` and `substpaths` lists. As a penultimate step, residual paths not belonging to substitutions are tested for symmetry or emptiness (`uni`, lines 15-19). Finally, we build a table of variables with their corresponding substitutions. This is then passed to `ext` which performs the actual extension of the substitution list (`uni`, line 21). `uni` errors out if out-of-scope variables are present or if rank-defined reboxing results in an unblanced number of cells.

```
 1  uni =: 2 : 0
 2   (_ _) u uni v y
 3  :
 4   u =. , <"({.x) u
 5   v =. , <"({:x) v
 6   if. (#u) ~: #v do. _1 return. end.
 7   tree =. u;<v
 8   paths =. (<S:1)@{:: tree
 9   varpaths_candidates =. paths #~ var S:0 tree
10   substpaths_candidates =. ((-. &.>@{.) 0} ]) &.> varpaths_candidates
11   prefix_paths =. ;@:(,@(<\) &.>)
12   subst_exists =. e.&(prefix_paths paths) substpaths_candidates
13  'varpaths substpaths' =. subst_exists&# &.> varpaths_candidates ;< substpaths_candidates
14   residual_paths =. paths -. varpaths , substpaths
15   if. -. *./ ((-.~ prefix_paths)~ residual_paths) e. prefix_paths substpaths
16    do. if. (0 = 2&|) # {::&tree &.> residual_paths
17         do. if. ~:/ ($~ (2,-:@#))~ {::&tree &.> residual_paths do. _1 return. end.
18        else. _1 return.
19        end.
20   end.
21   y ext {::&tree L:1 varpaths ,. substpaths
22  )
```

At this point, we have all the pieces necessary to implement the equality constraint. As in the reference implementation, this can be done in a straightforward manner by simply unifying ground terms. The result is boxed to correspond to the Kanren custom of returning a list of substitutions even if there only one such substitution. The example below shows the main advantage of our implementation. Here, J's ability to operate seamlessly over multidimensional arrays naturally extends to MicroKanren and allows us to unify two three-dimensional arrays. One array contains the alphabet drawn from the ASCII set while the other contains a sequence of the variables 0 through 25.

```
] abc =. (13 2 1 $ ;/ a. {~ 97+i.26) equ (13 2 1 $ ;/ i.26) fsh^:26 ''
+-----------------------------------------------------+
|+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
||a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z||
|+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
+-----------------------------------------------------+
```

The shape polymorphism inherent to our unification capabilities arises naturally from the use of J's operators, and therefore prevents unification of arrays of incompatible shapes. In this context though, incompatible does not mean that unification is restricted to terms of identical shape. Through the direct manipulation of rank, we can define which term fragments are to be unified. The example below illustrates this process: the left and right

ranks are r espectively 0 and 1. This allows us to unify each of the two fresh variables 26 and 27 of the left array with an entire row of the 2 X 6 letter array on the right. As is expected from MicroKanren relational semantics, we can now use bound variables from the substitution table to produce an array of any desired shape. bpro is syntactic sugar allowing simplified answer projection.

```
(26 27) bpro (3 :'(0 1) ((#y),(>:#y)) equ (2 6 $ > 15 4 0 13 20 19 1 20 19 19 4 17) fsh^:2 y') > abc
+------+------+
|peanut|butter|
+------+------+
```

### Disjunction, Conjunction and Further Constraints

app, apm, dis and con are almost literal transcriptions of their Racket counterparts. The only modification resides in delayed evaluation handling. J quotation implemented as the (5!:5) function results in a string that may later be executed using the ". primitive. Expressions that are already in string form but require additional quotation prior to execution are obtained using the quote function. Furthermore, our promises are boxed so that they can be appended to substitutions, thereby constituting what is called an "immature stream" in Kanrens. Promises are built both in app and apm, and also at the top level of every relation definition as a replacement to Racket's define-relation macro [12]. In the code below, 2&=@(3!:0)@> x is the equivalent of Racket's promise? predicate. line 5 in app builds the boxed string promise.

```
1  app =: 4 : 0
2   if. x -: ''
3    do. y
4   elseif. 2&=@(3!:0)@> x
5    do. < '(' , ((5!:5)<'y') , ') app ((3 :' , (quote 0 {:: x) , ')''''')'
6   elseif. do. ({. , (y app~ }.)) x
7   end.
8  )
```

In our J implementation we bypassed the call/initial-state - take - pull call chain from the reference implementation and directly implemented run. J's F: primitive allows us to iterate over the immature stream and collect results as a mature stream all at once. On each iteration, the promise resulting from the previous execution is forced using ". and the result is collected by taking the head ({.) of the new value. The definition of run can be found in appendix A.

## EXAMPLE GOALS AND BENCHMARKING

Here, we illustrate the execution steps of our implementation on a simple goal. The fives_and_sixes goal yields a stream of alternating values up to a user-defined length and uses run facilities identical to the reference implementation. fives_and_sixes is implemented as the disjunction of subgoals fives and sixes and shows that goal combinator functionalities similar to the reference implementation are available. Our ability to generate a stream of alternating values illustrates the interleaving search typical of Kanrens and the fact that our program correctly handles infinte recursive goals. As shown below, the execution of a recursive goal yields a pair of a value and a newly built promise. On further execution, the value is added to the result while the promise is forced to yield the following value-promise pair in the sequence.

```
fives_and_sixes fsh ''
10 run fives_and_sixes fsh ''
+---+-----------------------------------------------------+
|+-+|((<,<6.),<'sixes (,<00)') app ((3 :'fives (,<00)')'')|
||5||                                                     |
|+-+|                                                     |
+---+-----------------------------------------------------+

+---+---+---+---+---+---+---+---+---+---+
|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|
||5|||6|||5|||6|||5|||6|||5|||6|||5|||6||
|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|
+---+---+---+---+---+---+---+---+---+---+
```

We will now explore some goals through microbenchmarking which will reveal some of the stronger and weaker points of the J implementation. All benchmarking programs can be consulted in appendix B. Performance was not the main focus in this work. First, let us test the unification of two flat arrays of size *N*, one of which comprises only variables and the other only random float values (goal `floats`, appendix B).

Table 2. flat array of random floats unification benchmark (mean of 10 runs). Racket CS 7.7 on an Intel Core i7-6800K. Time columns are in seconds. Other columns are cumulative (cum.), step-by-step (step) and raw time ratios (Racket *vs.* J).

| | J | | | Racket | | | Racket vs. J |
|---|---|---|---|---|---|---|---|
| N | time | cum. | step | time | cum. | step | |
| 4 | 0.000165 | _ | _ | 0 | _ | _ | _ |
| 16 | 0.000242 | 1.5 | 1.5 | 0 | _ | _ | _ |
| 64 | 0.001764 | 10.7 | 7.3 | 0 | _ | _ | _ |
| 256 | 0.022942 | 139 | 13 | 0.0001 | _ | _ | 230 |
| 1024 | 0.327932 | 1987.5 | 14.3 | 0.003 | _ | 30 | 109 |
| 4096 | 4.84869 | 29386 | 14.8 | 0.0521 | _ | 17.4 | 93 |
| 16384 | 77.4688 | 469507.9 | 16 | 1.1027 | _ | 21.2 | 70 |

The reference implementation is much faster but raw running time comparison is not the most interesting element of our benchmarks given the completely different runtime architectures. Results from table 2 suggest that our trivial unification algorithm scales at least as well and perhaps better than the original in this case although J's interpreter is rather simplistic compared to Racket's compiler. This benchmark represents a particular situation that is most favourable to the J program because all variables are immediately unified with a value. In such a situation, the `ext` while loop immediately breaks without iterating as there are no candidate terms to filter, no remaining free variables and the whole array is therefore processed all at once without the need to walk the substitution list. Let us now illustrate the opposite situation. The following benchmark again tests the unification of two flat arrays but this time with a single starting value that propagates to the whole result array through a chain of variables (goal `as`, appendix B).

From the results in table 3, we see that Racket now has a dramatic advantage in speed. The same advantage would likely be seen in step ratios but J is slow enough on this benchmark that further testing of algorithmic scaling goes beyond our available time resources. Those results were expected since the test causes *N* iterations of the interpreted while loop in `ext`, and walking the substitution list is required on each iteration. As an intermediate between the programs from tables 2 and 3 additionally mobilizing the `run` facilities in both implementations.

Table 3. unification benchmark of two flat arrays comprising only variables at the exception of a single value that propagates to yield a list of size *N* including only repeated instances of the single starting value (mean of 10 runs). Racket CS 7.7 on an Intel Core i7-6800K. Time columns are in seconds. Other columns are cumulative (cum.), step-by-step (step) and raw time ratios (Racket *vs.* J).

| | J | | | Racket | | | Racket vs. J |
|---|---|---|---|---|---|---|---|
| N | time | cum. | step | time | cum. | step | |
| 4 | 0.000823 | _ | _ | 0 | _ | _ | _ |
| 16 | 0.003725 | 4.5 | 4.5 | 0 | _ | _ | _ |
| 64 | 0.014833 | 18 | 4 | 0 | _ | _ | _ |
| 256 | 0.22464 | 273 | 15.1 | 0 | _ | _ | _ |
| 1024 | 7.53526 | 9155.9 | 33.5 | 0.0009 | _ | _ | 8373 |
| 4096 | 411.842 | 500415.6 | 54.7 | 0.0185 | _ | 20.6 | 22262 |

## RELATED WORK

Logic programming over arrays is not a new idea but has historically enjoyed relatively little interest compared to LISP or Prolog solutions both prioritzing homoiconicity and lists as fundamental datastructure, making them ideal candidates for symbolic applications. In the array language domain some commercial systems include a limited unification system in their library, although array languages have traditionally focused on statistical and numerical computing [1]. Work focused on unification algorithms in the array context was explored by Brown *et al.* in 1987 [3]. By then, a large *corpus* of literature regarding logic capabilities of array languages was already available and those languages were already viewed by some as candidates for the "5th generation" computing project [22]. Following the work of Brown *et al.*, predicate calculus was implemented in APL2 by Engelmann *et al.*, bringing array languages closer to the Prolog world [8]. The newer trend of artificial intelligence based upon statistical rather than symbolic computing was then followed by APL-family representatives, unsurprisingly revealing themselves very capable environments in those array-heavy computation models up to this day [28]. The mixing of array- and logic-based solutions remains a research topic of substantial interest in a variety of fields, from end-user ergonomics to compilation techniques [7, 29]. Conceptually, hybrid reasoning may result in broader capabilities for artificial intelligence systems. In hierarchical control systems such as those found in behaviour-based robotics, the high-level symbolic component can rely on a reactive lower-level statistical (also termed *subsymbolic*) layer. The higher-levels benefit from relaxed time constraints allowing them to perform planning. Hybrid reasoning can result in a model of artificial intelligence capable of reasoning through induction, decuction, abduction and analogy and emulates natural systems [2]. Some recent works suggest that the unification of symbolic and statistical artificial intelligence may perhaps find potential for progress in the array-language world [19]. Indeed, arrays are a *must-have* for statistical computing and easy manipulation of arrays therefore must be available to the user wanting to combine statistical and symbolic reasoning. There are currently a variety of possibilities for programmers wanting to combine logic and array programming ranging from impure (`setarg/3`) or unwieldy (`arg/3`) Prolog idioms to non Turing-completeness *a la* Datalog. Those solutions represent two faces of the same Prolog coin, the former suffering from the usual pain points regarding logical purity and termination guarantees, while the latter imposes important semantic restrictions on the user. Furthermore, the implementation of performant Prolog or similar systems is far from trivial.

## LIMITATIONS AND FUTURE PROSPECTS

The limitations of our system are in part those of MicroKanren in that only the equality constraint is implemented and most interesting logic problems require other more expressive constraints. Unfortunately, syntactic constraints

make the elaboration of programs in our system rather fastidious and will require the addition of syntactic sugar to ease the writing of additional constraints. A second limitation is that as in the original MicroKanren we abandon most of the arithmetic capabilities of the host language by using natural numbers as variables. This is an opportunity to implement a relational arithmetic suite in the spirit of Kiselyov *et al.* [18]. Finally, the rank-polymorphism of our unification process is a natural extension benefiting from the host language's capabilities and is actually completely distinct from the unification algorithm itself. In its current form, our unification procedure is expected to be slow with regard to the state of the art and should perhaps be replaced by a program with better time and space complexity as described for example in Paterson & Wegman [24]. Interestingly, performant unification algorithms aim at the constitution of equivalence classes and do not use a "find" procedure. Such limitation of indirections in our program would perhaps allow a more vector-friendly implementation. However, some research suggests that the theoretical advantages of alternatives to Robinson's algorithm are not always realized in practice [14]. In this context, a first step will be the developement of an appropriate benchmarking framework geared towards vectorization opportunities. While our current implementation does not offer the features of mature logic environments, it delivers on the promise of a lightweight and logically pure basis for further exploration. Indeed, porting MicroKanren to an alternative datastructure basis might result in a new program writing style deriving from different tradeoffs imposed from the underlying arrays. This has not been explored in this paper, as the focus was on showing that the core language could be adapted. By implementing Kanren on top of an array language, we get close to a relational database management system and its traditional command language: SQL. MiniKanren-inspired systems for database querying have been deployed in production environments with success [4, 27]. Although extensions have recently made SQL Turing-complete, its impure semantics do not allow it to be considered a strict implementation of either predicate calculus or relational algebra. In contrast, MicroKanren has very clean semantics and holds potential for user-defined extensions, parallelism and constraint logic programming. The parallel drawn to database systems does not stop here: J includes Jd, an integrated column-store database. This is an opportunity for the implementation of a tabled version of MicroKanren based on Byrd's thesis that hopefully would improve both efficiency and termination guarantees of the system [6]. A second opportunity offered by Jd is to use it as a constraint store to implement constraint MicroKanren as described by Hemann *et al.* [11]. This would bring us closer to systems such as XSB Prolog while remaining in the purely relational context [26]. Another potential improvement would be constructive negation, which was recently described for MiniKanren and especially in combination with tabling would take us closer to supporting stable model semantics in Kanrens [23]. Finally, running a MicroKanren-inspired system on the GPU for massive parallelism is an encouraging research perspective that was already considered in Eric Holk's Harlan project which initially implemented a MiniKanren type checker and region inferencer [15]. In the array language world, the APL Co-dfns compiler recently brought us closer to running general programs on the GPU. However, Co-dfns does not currently provide delayed evaluation capabilities nor does it support nested arrays and alternatives will be required if the microKanren path is to be pursued on this platform [17]. Running on top of high-level GPU languages such as Futhark might also be considered, especially since prior art exists for this use case in the TAIL APL compiler [13].

## CONCLUSION

This work led us to the fact that J has many features making it ideal for the developement of a logic library, and that unification can be made rank-polymorphic by using characteristics of the array language paradigm as a stepping stone. While there are many enticing future perspectives in the line of research combining array and logic programming, many of those are open research problems. Despite that, this work provides a starting point for more involved forays into array-based pure relational programming.

, Vol. 1, No. 1, Article 3. Publication date: August 2021.

## ACKNOWLEDGMENTS

The author would like to express his gratitude to Dr. J. Hemann for his kindness and proofreading of the manuscript. Thanks also go to Dr. A. Hsu for his encouragments and to all members of the programming@jsoftware.com mailing list and r/apljk subreddit and especially Henry Rich, Julien Fondren and Raul Miller for their help with J language concepts. Finally, the author thanks the reviewers of the ICFP 2020 MiniKanren workshop for their useful advice.

## REFERENCES

[1] [n.d.]. Dyalog APL - Unification of Expressions (Visited May 11, 2020). https://dfns.dyalog.com/n_unify.htm.
[2] J.S. Albus. 1993. A Reference Model Architecture for Intelligent Systems Design. In *An Introduction to Intelligent and Autonomous Control*. Kluwer Academic Publishers, 27–56.
[3] James A. Brown and Ramiro Guerreiro. 1987. APL2 Implementations of Unification. In *Proceedings of the International Conference on APL: APL in Transition (APL '87)*. Association for Computing Machinery, Dallas, Texas, USA, 216–225. https://doi.org/10.1145/28315.28342
[4] Craig Brozefsky. 2013. SQL and Core.Logic Killed My ORM ; ClojureWest 2013 ; (https://www.infoq.com/presentations/Core-logic-SQL-ORM/).
[5] William Byrd. 2017. Relational Interpreters, Program Synthesis, and Barliman: Strange, Beautiful, and Possibly Useful ; CodeMesh 2017 London ; (https://codemesh.io/codemesh2017/william-e-byrd).
[6] William E. Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University, USA. Advisor(s) Friedman, Daniel P.
[7] Philip T. Cox and Patrick Nicholson. 2008. Unification of Arrays in Spreadsheets with Logic Programming. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Paul Hudak and David S. Warren (Eds.). Springer, Berlin, Heidelberg, 100–115. https://doi.org/10.1007/978-3-540-77442-6_8
[8] U. Engelmann, Th. Gerneth, and H. P. Meinzer. 1989. Implementation of Predicate Logic in APL2. *ACM SIGAPL APL Quote Quad* 19, 4 (July 1989), 124–128. https://doi.org/10.1145/75145.75162
[9] Jason Hemann and Daniel P Friedman. 2013. $\mu$Kanren: A Minimal Functional Core for Relational Programming. In *Scheme and Functional Programming Workshop*, Vol. 2013.
[10] Jason Hemann and Daniel P Friedman. 2015. How to Be a Good Host: miniKanren as a Case Study ; Curry On 2015 Prague ; (https://www.curry-on.org/2015/sessions/how-to-be-a-good-host-minikanren-as-a-case-study.html).
[11] Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. *Electronic Proceedings in Theoretical Computer Science* 234 (Jan. 2017), 135–149. https://doi.org/10.4204/EPTCS.234.10 arXiv:1701.00633
[12] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A Small Embedding of Logic Programming with a Simple Complete Search. In *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016*. ACM Press, Amsterdam, Netherlands, 96–107. https://doi.org/10.1145/2989225.2989230
[13] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing (FHPC 2016)*. Association for Computing Machinery, Nara, Japan, 38–43. https://doi.org/10.1145/2975991.2975997
[14] Kryštof Hoder and Andrei Voronkov. 2009. Comparing Unification Algorithms in First-Order Theorem Proving. In *KI 2009: Advances in Artificial Intelligence (Lecture Notes in Computer Science)*, Bärbel Mertsching, Marcus Hund, and Zaheer Aziz (Eds.). Springer, Berlin, Heidelberg, 435–443. https://doi.org/10.1007/978-3-642-04617-9_55
[15] Eric Holk, William E Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. 2011. Declarative Parallel Programming for GPUs. 297–304.
[16] Aaron W. Hsu. 2016. The Key to a Data Parallel Compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. Association for Computing Machinery, Santa Barbara, CA, USA, 32–40. https://doi.org/10.1145/2935323.2935331
[17] Aaron W. Hsu. 2019. A Data Parallel Compiler Hosted on the GPU. (Nov. 2019).
[18] Oleg Kiselyov, William E Byrd, Daniel P Friedman, and Chung-chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). In *International Symposium on Functional and Logic Programming*. Springer, 64–80.
[19] Ryosuke Kojima and Taisuke Sato. 2019. A Tensorized Logic Programming Language for Large-Scale Data. *arXiv:1901.08548 [cs]* (Jan. 2019). arXiv:1901.08548 [cs]
[20] Dmitrii Kosarev and Dmitry Boulytchev. 2018. Typed Embedding of a Relational Language in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (Dec. 2018), 1–22. https://doi.org/10.4204/EPTCS.285.1 arXiv:1805.11006
[21] Robert Kowalski and Steve Smoliar. 1982. Logic for Problem Solving. *ACM SIGSOFT Software Engineering Notes* 7, 2 (April 1982), 61–62. https://doi.org/10.1145/1005937.1005947

[22] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. 1984. Nial: A Candidate Language for Fifth Generation Computer Systems. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge (ACM '84)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/800171.809618

[23] Evgenii Moiseenko. 2019. Constructive Negation for MiniKanren. https://www.semanticscholar.org/paper/Constructive-Negation-for-MiniKanren-Moiseenko/1b7fd5635770759232f3d88682c4d5bb7b5a7b54.

[24] M. S. Paterson and M. N. Wegman. 1978. Linear Unification. *J. Comput. System Sci.* 16, 2 (April 1978), 158–167. https://doi.org/10.1016/0022-0000(78)90043-0

[25] J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41. https://doi.org/10.1145/321250.321253

[26] Konstantinos Sagonas, Terrance Swift, and David S. Warren. 1994. XSB as an Efficient Deductive Database Engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM Press, 442–453.

[27] Ryan Senior. 2012. Practical Core.Logic ; ClojureWest 2012 ; (https://www.infoq.com/presentations/core-logic/).

[28] Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 69–79. https://doi.org/10.1145/3315454.3329960

[29] Justin Slepak, Panagiotis Manolios, and Olin Shivers. 2018. Rank Polymorphism Viewed as a Constraint Problem. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 34–41. https://doi.org/10.1145/3219753.3219758

## APPENDIX A: MICROKANREN SOURCE

```
var =: -.@#@$*.e.&4 1@(3!:0)
get =: {::~^:(var@])"(_ 0)^:_
occ =: (e. < S: 0)`0:@.-:
ext =: 4 : 0
while. 1
 do.
  y =. x&get L:0 y
  y =. ~. (#~ (i.@# < i.&1"1@(-:"1/ |."1)~))~ y
  if. '' -: y do. x return. end.
  isvar =. > var &.> y
  if. +./ (occ/"1 y) , (+:/"1 isvar) do. _1 return. end.
  candidates =. (*:/"1 isvar) # y {"1~ |."1 isvar
  if. '' -.@-: , candidates
   do.
    x =. ({:"1 candidates) (>@{."1 candidates) } x
    y =. y -. candidates
  else. break.
  end.
end.
if. '' -.@-: , y do. ({:"1 y) (>@{."1 y) } x  else. x end.
)
uni =: 2 : 0
 (_ _) u uni v y
:
 u =. , <"({.x) u
 v =. , <"({:x) v
 if. (#u) ~: #v do. _1 return. end.
```

```
  tree =. u;<v
  paths =. (<S:1)@{:: tree
  varpaths_candidates =. paths #~ var S:0 tree
  substpaths_candidates =. ((-. &.>@{.) 0} ]) &.> varpaths_candidates
  prefix_paths =. ;@:(,@(<\) &.>)
  subst_exists =. e.&(prefix_paths paths) substpaths_candidates
 'varpaths substpaths'=.subst_exists&#&.>varpaths_candidates;<substpaths_candidates
  residual_paths =. paths -. varpaths , substpaths
  NB.check if residual values are inside a substitution
  if. -. *./ ((-.~ prefix_paths)~ residual_paths) e. prefix_paths substpaths
  NB.if not inside a substitution, check if non-identical and if so, fail.
   do. if. (0 = 2&|) # {::&tree &.> residual_paths
        do. if. ~:/ ($~ (2,-:@#))~ {::&tree &.> residual_paths do. _1 return. end.
       else. _1 return.
       end.
  end.
  y ext {::&tree L:1 varpaths ,. substpaths
 )
 equ =: 2 : 0
  (_ _) u equ v y
 :
  < x (y get u) uni (y get v) y
 )
 fsh =: ,<@#
 app =: 4 : 0
  if. x -: ''
   do. y
  elseif. 2&=@(3!:0)@> x
   do. < '(' , ((5!:5)<'y') , ') app ((3 :' , (quote 0 {:: x) , ')''''')'
  elseif. do. ({. , (y app~ }.)) x
  end.
 )
 apm =: 2 : 0
  if. u -: ''
   do. ''
  elseif. 2&=@(3!:0)@> u
   do. < '((3 :' , (quote@> u) , ')''''') apm (' , ((5!:5)<'v') , ')'
  elseif. do. (v (>u)) app (}.u) apm v
  end.
 )
 dis =: 2 : '(u y) app (v y)'
 con =: 2 : '(u y) apm v'
 ground =: 3 : '(y&get L:0)^:_ y'
 runhelper =: (".@(1&{::) , (<: &.>@{:)) [ _2&Z:@-.@*@(2&{::)
 run =: 2 : 0
```

```
  if. u > 1
   do. 'initval initprom' =. v y
       initval ; ground &.> runhelper F: {. _ ; initprom ; < <: u
  elseif. u = 1 do. {. v y
  elseif. u = 0 do. ''
  elseif. do. _1
  end.
 :
  if. u > 1
   do. 'initval initprom' =. v y
       (x&{ &.> initval) ; (x&{@ground &.>) runhelper F: {. _ ; initprom ; < <: u
  elseif. u = 1 do. {. v y
  elseif. u = 0 do. ''
  elseif. do. _1
  end.
 )
 bpro =: 4 : '(>y)&get L:0 x&{ > y'
 upro =: 4 : '((>y)&get L:0)@> x&{ > y'
```

## APPENDIX B: BENCHMARKING CODE

### J

```
floats =: (3 : '(0 0) y uni (i. # y) fsh^:(#y) '''' ')
floats_perf =: 3 : 0
 n =. */\ 4 #~ y
 data =. (?@#&0) &.> ;/ n
 t =. (x:!.0)(6!:2)@> 'floats '&,@":@(?@#&0) &.> ;/ n
 ct =. _ , */\ (x:2) %~/\ ; t
 st =. _ , (x:2) %~/\ ; t
 legend =. 'N';'time [s]';'cumulative time ratio';'step time ratio'
 legend , <"0 n ,. t ,. ct ,. st
)
floats_perf 7
as =: 3 : '(0 0) ((<''a'') , ;/ i. <: y) uni (;/ i. y) (;/ i. y)'
as_perf =: 3 : 0
 n =. */\ 4 #~ y
 t =. (x:!.0) (6!:2)@> ('as ' , ":) &.> ;/ n
 ct =. _ , */\ (x:2) %~/\ ; t
 st =. _ , (x:2) %~/\ ; t
 legend =. 'N';'time [s]';'cumulative time ratio';'step time ratio'
 legend , <"0 n ,. t ,. ct ,. st
)
as_perf 6
```

### Racket

```
(require microkanren
         atomichron
         racket/flonum)
(define (powers-of-n n iterations)
  (map (lambda (exponent) (expt n (add1 exponent))) (build-list iterations values)))
(define (gensym-list n)
  (build-list n (lambda (x) (gensym))))
(define (random-float-list n)
  (define rg (make-pseudo-random-generator))
  (for/list ([i (build-list n values)]) (flrandom rg)))
(define (time-in-seconds benchmark-times)
  (map (lambda (n) (exact->inexact (/ n (expt 10 9)))) benchmark-times))
(define (random-floats-benchmark l r)
  (make-microbenchmark
   #:name 'random-floats-benchmark
   #:iterations 1
   #:microexpression-iterations 10
   #:microexpression-builder
   (lambda (iteration)
     (make-microexpression #:thunk (lambda () (unify l r '()))))))
(define float-times
    (map
     (lambda (x)
       (let ([left (random-float-list x)]
             [right (build-list x values)])
         (microbenchmark-result-average-real-nanoseconds
          (microbenchmark-run! (random-floats-benchmark left right)))))
     (powers-of-n 4 7)))
(begin (display "time in seconds (random floats): ") (time-in-seconds float-times))
(define (as-benchmark l r)
  (make-microbenchmark
   #:name 'as-benchmark
   #:iterations 1
   #:microexpression-iterations 10
   #:microexpression-builder
   (lambda (iteration)
     (make-microexpression #:thunk (lambda () (unify l r '()))))))
(define as-times
  (map
   (lambda (x)
     (let ([left (cons #\a (build-list (sub1 x) values))]
           [right (build-list x values)])
       (microbenchmark-result-average-real-nanoseconds
 (microbenchmark-run! (as-benchmark left right)))))
```

```
   (powers-of-n 4 7)))
(begin (display "time in seconds (list of as): ") (time-in-seconds as-times))
```

# Higher-order Logic Programming with λKanren

WEIXI MA, KUANG-CHEN LU, and DANIEL P. FRIEDMAN, Indiana University, USA

We present λKanren, a new member of the Kanren family [2] that is inspired by λProlog [5]. With a shallow embedding implementation, the term language of λKanren is represented by the functions and macros of its host language. As a higher-order logic programming language, λKanren is extended with a subset of higher-order hereditary Harrop formulas [7].

## 1 INTRODUCTION

λKanren introduces four new operators to μKanren [3]: `tie`, `app`, `assume-rel`, and `all`. The opertors `tie` and `app` create binding structures. In addition, the ≡ operator recognizes $\alpha\beta$-conversions between binding structures. The `assume-rel` and `all` operators enable more expressive reasoning with Hereditary Harrop formulas [6]. To demonstrate λKanren's increment to μKanren, we first review the two forms of logic, *fohc* and *hohh*, behind these two languages.

μKanren implements First-order Horn clause (fohc) [1]. The grammar of *Horn clause* is shown in Fig 1. We say μKanren is *first-order*, as its unification algorithm identifies only structural equivalence. As an example that illustrates the correspondence between μKanren definitions and fohc formulas, consider the relation append$^o$.

```
(defrel (appendᵒ xs ys zs)
  (condᵉ
    [(≡ nil xs) (≡ ys zs)]
    [(fresh (a d r)
       (≡ `(,a . ,d) xs)
       (appendᵒ d ys r)
       (≡ `(,a . ,r) zs))]))
```

*D formulas of fohc.* In μKanren, a `defrel` introduces a *D formula*. For example, the append$^o$ definition corresponds to this *D formula*,

$$\begin{aligned}\forall xs\,\forall ys\,\forall zs \quad & (\equiv\ xs\ nil) \wedge (\equiv\ ys\ zs)\\ & \vee\ \exists a\,\exists d\,\exists r\,(\equiv\ xs\ `(a.d)) \wedge (append^o\ d\ ys\ r) \wedge (\equiv\ `(a.r)\ zs)\\ & \supset (append^o\ xs\ ys\ zs).\end{aligned}$$

Here append$^o$ and ≡ both build atomic formulas. For example, $(append^o\ xs\ ys\ zs)$ and $(\equiv\ xs\ nil)$ are atomic formulas.

Authors' address: Weixi Ma, mvc@iu.edu; Kuang-chen Lu, kl13@iu.edu; Daniel P. Friedman, dfried00@gmail.com, Indiana University, USA.


miniKanren.org/workshop/2021/8-ART1

$$\begin{array}{rrcl} \text{Goals} & G & ::= & A \mid G \wedge G \mid G \vee G \mid \exists x\, G \\ \text{Definitions} & D & ::= & A \mid G \supset D \mid D \wedge D \mid \forall x\, D \\ \text{Atomic Formulas} & A & & \end{array}$$

Fig. 1. Horn Clause Formulas

$$\begin{array}{rrcl} \text{Goals} & G & ::= & A \mid G \wedge G \mid G \vee G \mid \exists x\, G \mid D \supset G \mid \forall x\, G \\ \text{Definitions} & D & ::= & A \mid G \supset D \mid D \wedge D \mid \forall x\, D \\ \text{Atomic Formulas} & A & & \end{array}$$

Fig. 2. Hereditary Harrop Formulas

*G formulas of fohc.* In $\mu$Kanren, a `run` query contains a *G formula*, e.g.,

```
(run 1
  (fresh (xs)
    (appendᵒ xs `(1 2) `(1 2))))
```

is formulated as

$$\exists xs\ (append^o\ xs\ \text{`}(1\,2)\ \text{`}(1\,2)).$$

*Formulas of hohh.* $\lambda$Prolog implements a more expressive logic, *higher-order hereditary Harrop formulas* (hohh) [6]. Shown in Figure 2, Hereditary Harrop formulas extend G formulas with implicational goals and forall-quantification. Also, with higher-order unification, the unification algorithm of $\lambda$Prolog identifies $\alpha\beta$-equivalence between binding structures (that are absent in $\mu$Kanren).

This paper presents $\lambda$Kanren. $\lambda$Kanren implements implicational goals and forall-quantification with two new operators, `assume-rel` and `all`, respectively. Also, $\lambda$Kanren incorporates higher-order pattern unification [4] for the binding structures (that are created by another two new operators, `tie` and `app`).

The rest of this paper demonstrates the uses of these four operators and their implementation details when appropriate. Our implementation of $\lambda$Kanren is available at https://github.com/mvcccccc/MK2020.

## 2 HIGHER-ORDER UNIFICATION

This section shows the power of higher-order pattern unification. By adapting Miller [4]'s unification algorithm, $\lambda$Kanren is equipped with two new operators: `tie` and `app`. `tie` expressions are abstractions and `app` is the shorthand for application. The ≡ operator in $\lambda$Kanren identifies $\alpha\beta$-equivalence between terms that involve `tie` and `app`.

Consider the following example that demonstrates $\alpha$-equivalence. This example, metaphorically, tests the equivalence between `(λ (a b) (a b))` and `(λ (x y) (x y))`.

```
> (run 1 q
    (== (tie (a b) (app a b))
        (tie (x y) (app x y))))
'(_0)
```

`tie` is implemented as the following macro. It takes a list of variable names and a term. It then creates a `Tie` structure that is internally used for curried binders. Hereafter, we call a variable that is introduced by `fresh` a *unification variable* and a variable that is introduced by `tie` a *binding variable*.

```
(define-syntax tie
  (syntax-rules ()
    [(_ () t) body]
    [(_ (x₀ x ...) body)
     (let ([x₀ (Var 'x₀)])
       (Tie x₀ (tie (x ...) body)))]))
```

app is implemented as the following macro that elaborates a list of terms to an App structure that is internally used for curried applications.

```
(define-syntax app
  (syntax-rules ()
    [(_ rator rand) (App rator rand)]
    [(_ rator rand₀ rand ...)
     (app (App rator rand₀) rand ...)]))
```

Next, consider the following example that queries for two instantiations of f. This example demonstrates (1) $\beta$-conversions during unification and (2) how binding structures are reified.

```
> (run 2 f
    (== (tie (a b) (app a b))
        (tie (x y) (app f x y))))
'((tie (_0) (tie (_1) (app _0 _1))))
```

There is only one instantiation: f is a function (a Tie structure) of two inputs and f outputs an application form (a App structure) that applies its first input on the second one.

The internal structures, Tie and App, are reified as tagged lists. These tagged lists reflect their corresponded user interfaces, tie and app. During reification, binding variables and unification variables are both converted to underscore-digit symbols.

The power of higher-order unification, however, comes in with limits. To ensure decidability, $\beta$-conversion in Miller [4]'s algorithm restricts application forms: when the operator of an app is a unification variable, its operands must be distinct binding variables, otherwise unification fails. For example, the following query has no solution because the operands, the two bs, of the unification variable f are not distinct. In this case, with f being a function of two input bs, we cannot decide which b takes control.

```
> (run 1 f
    (== (tie (a) a)
        (tie (b) (app f b b))))
'()
```

To enforce this restriction, Miller [4]'s algorithm imposes another restriction on variable scopes: the instantiation of a unification variable may only contain its *visible* binding variables. A binding variable x is visible to a unification variable q if the introduction of x lexically precedes that of q. Given the following example, it seems that q can be instantiated by y. Unfortunately, y is not visible to q and the query has no solution.

```
> (run 1 q
    (== (tie (a b) (app a b))
        (tie (x y) (app x q))))
'()
```

In our implementation, the unifier extends higher-order pattern unification and adapts it for conventional miniKanren programming style. In Miller [4]'s algorithm, a unification variable must be an operator of an application form. The operands of the application must be distinct binding variables. That means, Miller [4]'s algorithm reports unsovability when a unification variable is simply unified against a constant. (In fact, constants are not in the term language of Miller [4]'s algorithm).

In miniKanren, we often unify a single unification variable and a constant. So, it is compelling that we extend the term language and the unification variable, as we have done in our implementation.

## 3 IMPLICATIONAL GOALS

This section introduces the `assume-rel` operator that implements implicational goals ($D \supset G$). An `assume-rel` operator takes two inputs: (1) the hypothesis in the form of a D formula and (2) the goal in the form of a G formula. The `assume-rel` operator then uses the hypothesis as a fact and moves on to the goal.

Implementing `assume-rel` is subtle with shallow embedding. Because the definitions of $\lambda$Kanren are kept in the run-time environment of its host language, extending these definitions requires updating the run-time environment. This problem is illustrated in the following example, liberally adapted from Miller and Nadathur [5, p. 80].

```
(defrel (taken name class)
  (conde
    [(== 'Josh name) (== 'B521 class)]
    [(== 'Josh name) (== 'B522 class)]))
(defrel (pl-major name)
  (taken name 'B521)
  (taken name 'B523)
  (taken name 'B522))
```

One may complete `pl-major` after taking three classes: `B521`, `B522`, and `B523`. And Josh currently has `taken` `B521` and `B522`. In the following query, the `assume-rel` operator extends the definition of `taken` with `(taken 'Josh q)` and then moves on to the goal `(pl-major 'Josh)`.

```
> (run 1 q
(assume-rel [(taken name class)
             (== 'Josh name)
             (== 'q class)]
  (pl-major 'Josh)))
'(B523)
```

From the implementation aspect, because the host language is lexically scoped, the definition of `pl-major` is fixed. This means that, the free variable in the definition of `pl-major`, namely `taken`, always uses the original definition of Josh taking `B521` and `B522`. To extend definitions on the fly, we need to create dynamic scope so that the free variables may use the latest, updated definitions.

Our approach is to add an extra layer between $\lambda$Kanren and the host language (Racket). This extra layer redirects function definitions.

We introduce two global maps, `name->idx` and `idx->def`. Each `defrel` extends these two maps by creating a new index, putting the name-idx pair and the idx-def pair in the two maps respectively. The `idx->def` map is global. And the `name->idx` map is threaded through during the execution of a query (an invocation of a `run`).

To invoke a definition, one follows `name->idx` and `idx->def`, i.e., first retrieving the index using `name->idx` and then getting the definition using `idx->def`. For example, the user interface

```
(pl-major 'Josh)
```

is macro-expanded to

```
((cdr (assv idx->def (cdr (assv name->idx 'pl-major))))
 'Josh).
```

When an `assume-rel` operator is invoked, the two maps are extended again: (1) a new index is created; (2) `idx->def` contains the pair of the new index and the extended function; and (3) `name->idx` now has a new pair of the definition name and the new index, this new pair shadows the previous one.

In the previous example, let's use $t_1$ for the `taken` definition that knows Josh has taken `B521` and `B522`, use $t_2$ for the extended `taken` (where we `assume-rel` Josh has taken `B523`), and use `p` for the definition of `pl-major`. With `taken` and `pl-major` first defined, `name->idx` is `((pl-major . 2) (taken . 1))` and `idx->body` is `((2 . p) (1 . t1))`. Then, after assuming `(taken 'Josh q)`, the query `(pl-major 'Josh)` runs in an updated environment where `name->idx` is `((taken . 3) (pl-major . 2) (taken . 1))` and `idx->body` is `((3 . t2) (2 . p) (1 . t1))`. The more recent pair in `name->idx` shadows the previous one. And therefore, when `taken` is invoked, we use $t_2$.

Many interesting examples only make hypothesis on atomic formulas. And thus we provide the `assume` operator that is a shorter version of the `assume-rel` operator. Instead of any D formula, the `assume` operator only takes an atomic hypothesis.

As an example, we define the `eq` relation to be reflexive, transitive, and symmetric as follows.

```
(defrel (eq x y)
  (cond^e
    [(== x y)]
    [(fresh (z)
       (eq x z)
       (eq z y))]
    [(eq y x)]))
```

Obviously `apple` and `orange` are by no means `eq`. In fact, the following query does not terminate in a naive $\mu$Kanren implementation because the third cond$^e$ line is very recursive.

```
> (run 1 q
    (eq 'apple 'orange))
```

Using `assume`, we may temporarily extend the definition of `eq` as follows.

```
> (run 5 q
    (assume (eq 'orange 'apple)
      (assume (eq 'orange 'dog)
        (eq 'orange q))))
'(dog apple orange orange dog)
```

Because λKanren runs backward, as in the following, the hypothesis can be inferred as well.

```
> (run 1 q
    (assume (eq 'orange q)
      (eq 'apple 'orange))))
'(apple)
```

## 4 FORALL-QUANTIFICATION

This section introduces the `all` operator ($\forall x G$) that takes a list of symbols and a goal. These symbols are used to create special variables that are virtually constants (eigenvariables).

Continuing with `taken` and `pl-major`, we create a random person `x` using the `all` operator.

```
> (run 1 q
    (all (x)
      (assume-rel (taken x 'B521)
        (assume-rel (taken x 'B522)
          (assume-rel (taken x 'B523)
            (pl-major x))))))
'(_0)
```

Like the `fresh` operator, the `all` operator creates a new variable in the scope. Unlike the `fresh` operator, the `all` operator effectively creates a constant. This semantics is similar to the proof technique of a for-all goal in first-order logic: to prove $\forall x.P$, we fix a constant $x$ and then prove $P$.

Consider the next example that synthesizes the identity function using the `all` operator.

```
> (run 1 f
    (all (x)
      (== x (app f x))))
'((tie (_0) _0))
```

The implementation of the `all` operator follows that of the `fresh` operator, except that the created variable is a constant. In our implementation, we create an `all` variable as a free binding variable. Thus, the `all` variable cannot be unified with anything but itself.

## 5 λKANREN AS A THEOREM PROVER

λProlog is often regarded as a proof system. With hohh, λKanren suits a theorem prover as well. This section shows examples that use λKanren to prove intuitionistic style theorems.

We start with the definition of `proved`. At this moment, only `'trivial` is proved.

```
(defrel (proved x)
   (== x 'trivial))
```

Obviously, not everything is proved.

```
> (run 1 g
    (all (p)
      (proved p)))
'()
> (run 1 g
    (proved g))
'(trivial)
```

Next, we prove the commutativity of conjuction. I.e., $\forall p, q\ (p \wedge q) \supset (q \wedge p)$.

```
> (run 1 g
    (all (p q)
      (assume ((proved p) (proved q))
        (proved q)
        (proved p))))
'(_0)
```

The introduction rule of disjunction can be proved: $\forall p, q\ p \supset (p \vee q)$

```
  (run 1 goal
    (all (p q)
      (assume ((proved p))
        (conde
          [(proved p)]
          [(proved q)]))))
'(_0)
```

## 6 CONCLUSION

λKanren is based on higher-order hereditary Harrop formulas. It extends μKanren with four operators, `tie`, `app`, `assume-rel`, and `all`. In addition, unification (`==`) identifies $\alpha\beta$-equivalence between the binding operators.

Our implementation of λKanren is written in Racket by adding about 40 lines to μKanren. Overall, we appreciate the simplicity provided by the shallow embedding techniques.

## REFERENCES


[1] Krzysztof R. Apt and M. H. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM (JACM)*, 29(3):841–862, July 1982. ISSN 0004-5411. doi: 10.1145/322326.322339. URL https://doi.org/10.1145/322326.322339.

[2] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. The Reasoned Schemer, Second Edition, 2018.

[3] Jason Hemann and Daniel P. Friedman. μKanren: A Minimal Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme'13)*, volume 6, 2013.

[4] Dale Miller. A Logic Programming Language with Lambda-abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, September 1991.

[5] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, USA, 1st edition, 2012. ISBN 978-0-521-87940-8.

[6] Dale Miller, Gopalan Nadathur, and Andre Scedrov. HEREDITARY HARROP FORMULAS AND UNIFORM PROOF SYSTEMS. *Unknown Host Publication Title*, pages 98–105, January 1987. URL https://experts.umn.edu/en/publications/hereditary-harrop-formulas-and-uniform-proof-systems. Publisher: IEEE.

[7] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1):125–157, March 1991. ISSN 0168-0072. doi: 10.1016/0168-0072(91)90068-W. URL http://www.sciencedirect.com/science/article/pii/016800729190068W.

# Kanren Light

A Dynamically Semi-Certified Interactive Logic Programming System

MARCO MAGGESI, University of Florence, Italy
MASSIMO NOCENTINI, University of Florence, Italy

We present an experimental system strongly inspired by miniKanren, implemented on top of the tactics mechanism of the HOL Light theorem prover. Our tool is at the same time a mechanism for enabling the logic programming style for reasoning and computing in a theorem prover, and a framework for writing logic programs that produce solutions endowed with a formal proof of correctness.

CCS Concepts: • **Theory of computation** → *Interactive computation*; *Streaming models*; **Logic and verification**; **Automated reasoning**.

Additional Key Words and Phrases: miniKanren, HOL Light, Certified, Theorem Proving, Program verification

## 1 INTRODUCTION

In straightforward terms, the computation of a logic program evolves by refining a substitution seeking for solutions of a unification problem. This has been made explicit in the Kanren approach [1, 5, 7], where programs are described by composing (higher-order) operators that act on streams of substitutions. Such a methodology allows for a streamlined approach to logic programming; however, the intended semantics and the correctness of a Kanren program rest entirely on the meta-theoretic level.

We propose a framework that extends the Kanren approach to a system that computes both candidate substitutions and corresponding certificates of correctness with respect to a given specification. Such certificates will be formally verified logical truths synthesized using a theorem prover.

Our setup is based on the HOL Light theorem prover [6], in which we extend the currently available tactic mechanism with three basic features: (i) the explicit use of meta-variables, (ii) the ability to backtrack during the proof search, (iii) a layer of tools and facilities for interfacing with the underlying proof mechanism.

The basic building block of our framework are ML procedures that we call *solvers*, which are a generalization of HOL tactics and are –as well as tactics– meant to be used compositionally in order to define arbitrarily complex proof search strategies.

We say that our approach is *semi-certified* because

- on the one hand, the synthesized solutions are formally proved theorems, hence their validity is guaranteed by construction;
- on the other hand, the completeness of the search procedure cannot be enforced in our framework and consequently has to be ensured by a meta-reasoning.

Authors' addresses: Marco Maggesi, University of Florence, Italy, marco.maggesi@unifi.it; Massimo Nocentini, University of Florence, Italy, massimo.nocentini@unifi.it.


miniKanren.org/workshop/2021/8-ART5

Moreover, we say that our system is *dynamically* semi-certified, because the proof certificate is built at run-time.

At the present stage, our implementation is intended to be a testbed for experiments and further investigation on this reasoning paradigm. Section 6 gives some further information on our code.

## 2 A WORD ABOUT THE HOL LIGHT THEOREM PROVER

In the HOL system, there are two fundamental datatypes called *term* and *theorem*. Terms model fragments of (well-formed) mathematical expressions. Theorems are Boolean terms that are proved correct according to a fixed set of logical rules. Examples of both a term and a theorem in the concrete syntax of HOL Light are

`` `2 + 2` `` and `|- 2 + 2 = 4`.

Notice that terms are written enclosed in backquotes while theorems use the entailment symbol `|-`.

A theorem `as |- b` is a first-class value composed by a list of terms `as` and a body `b`, for instance

```
# ARITH_SUC;;
val it : thm =
  |- (!n. SUC (NUMERAL n) = NUMERAL (SUC n)) /\
     SUC _0 = BIT1 _0 /\
     (!n. SUC (BIT0 n) = BIT1 n) /\
     (!n. SUC (BIT1 n) = BIT0 (SUC n))
```

is synthetized via regular function call of `prove` that actually consumes the theorem's body and the corresponding proof,

```
let ARITH_SUC = prove
 (`(!n. SUC(NUMERAL n) = NUMERAL(SUC n)) /\
   (SUC _0 = BIT1 _0) /\
   (!n. SUC (BIT0 n) = BIT1 n) /\
   (!n. SUC (BIT1 n) = BIT0 (SUC n))`,
  REWRITE_TAC[NUMERAL; BIT0; BIT1; DENUMERAL ADD_CLAUSES]);;
```

The Boolean connectives '$\wedge$', '$\vee$', '$\Longrightarrow$' are represented in ASCII encoding `/\`, `\/` and `==>`, respectively. Universal and existential quantifier $\forall x.\, Px$ and $\exists x.\, Px$ are denoted with exclamation and interrogation marks: `!x. P x` and `?x. P x`. Other syntactic elements are borrowed from the ML world, such us the notation for concrete lists `[x1;...]`.

As the name suggests, HOL (*Higher-Order Logic*) implements a higher-order language based on a variant of the typed lambda calculus. Hence, in a rough comparison with classical logic programming languages, our system is closer to $\lambda$Prolog [9] than the usual (first-order) Prolog.

Interactive proofs in HOL Light are performed by running *tactics* that operate on a context called *goal*, which represents the intermediate status of the current logical reasoning. There are simple tactics that model basic logical inference steps as well as sophisticated tactics that implement powerful decision procedures.

From the theorem proving perspective, our work consists of extending the HOL Light's tactic mechanism by introducing specific ideas coming from the miniKanren methodology. The resulting system allows the user to build proof scripts either with tactics or solvers, and the resulting theorems will be available to the programming environment regardless of which proof mechanism has been utilized. In particular, the new theorems can be built on top of the standard library of HOL Light, populated by several thousand theorems.

## 3 A SIMPLE EXAMPLE

To give the flavor of our framework, we show an example of how to perform simple computations on lists. Let us consider the problem of computing the concatenation of two lists `[1; 2]` and `[3]`. One idiomatic way to

approach this problem in HOL is by using *conversions* [11]. Conversions are ML procedures that receive as input a term $t$ and output a theorem of the form `|- t = t'`. The term $t'$ is the *result* of the computation, and the theorem itself is the *certificate* that guarantees its correctness. Let us show first how conversions are used before describing how one can perform the same task using our framework.

In HOL Light, one has the constant `APPEND` and the equational theorem (of the same name) that characterize it

```
|- (!l. APPEND [] l = l) /\
   (!h t l. APPEND (h :: t) l = h :: APPEND t l)
```

We can then use the conversion `REWRITE_CONV` which performs the rewriting. The ML command is

```
# REWRITE_CONV [APPEND] `APPEND [1;2] [3]`;;
```

which produces the theorem

```
|- APPEND [1; 2] [3] = [1; 2; 3]
```

Our implementation allows us to address the same problem from a logical point of view. We start by recalling two theorems that are proved – via list structural induction – during the bootstrap procedure of HOL Light, namely

```
# APPEND_NIL;;
val it : thm = |- !l. APPEND [] l = l
```

and

```
# APPEND_CONS;;
val it : thm =
  |- !x xs ys zs. APPEND xs ys = zs
                  ==> APPEND (x :: xs) ys = x :: zs
```

to give the logical rules, in form of Horn clauses, that characterize the `APPEND` operator. Then we define a *solver*

```
let APPEND_SLV : solver =
  REPEAT_SLV (CONCAT_SLV (ACCEPT_SLV APPEND_NIL)
                         (RULE_SLV APPEND_CONS));;
```

which implements the most obvious strategy for proving a relation of the form `` `APPEND x y = z` `` by structural analysis on the list `` `x` ``. The precise meaning of the above code will be clear later; however, this can be seen as the direct translation of the Prolog program

```
append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Then, the problem of concatenating the two lists is described by the term

```
`??x. APPEND [1;2] [3] = x`
```

where the binder `` `(??)` `` is a syntactic variant of the usual existential quantifier `` `(?)` ``, which introduces the *meta-variables* of the *query*.

The following command

```
list_of_stream
  (solve APPEND_SLV
         `??x. APPEND [1; 2] [3] = x`);;
```

runs the search process where (i) the `solve` function starts the proof search and produces a stream (i.e., a lazy list) of *solutions* and (ii) the outermost `list_of_stream` transforms the stream into a list.

The output of the previous command is a single solution which is represented by a pair where the first element is the instantiation for the meta-variable `` `x` `` and the second element is a HOL theorem

```
val it : (term list * thm) list =
  [([`x = [1; 2; 3]`], |- APPEND [1; 2] [3] = [1; 2; 3])]
```

Since the theorem is the instantiation of the original query term, it certifies the correctness of the solution.

Now comes the interesting part: as in logic programs, our search strategy (i.e., the `APPEND_SLV` solver) can be used for backward reasoning. Consider the variation of the above problem where we want to enumerate all possible splits of the list `[1; 2; 3]`. This can be done by simply changing the goal term in the previous query:

```
# list_of_stream
    (solve APPEND_SLV
           `??x y. APPEND x y = [1;2;3]`);;

val it : (term list * thm) list =
  [([`x = []`; `y = [1; 2; 3]`],
    |- APPEND [] [1; 2; 3] = [1; 2; 3]);
   ([`x = [1]`; `y = [2; 3]`],
    |- APPEND [1] [2; 3] = [1; 2; 3]);
   ([`x = [1; 2]`; `y = [3]`],
    |- APPEND [1; 2] [3] = [1; 2; 3]);
   ([`x = [1; 2; 3]`; `y = []`],
    |- APPEND [1; 2; 3] [] = [1; 2; 3])]
```

The system finds the above solutions by filtering and refining a stream of substitutions, precisely in the same way it is done in any typical miniKanren implementation; eventually, the interesting part is the associated theorems that are synthesized.

## 4 A LIBRARY OF SOLVERS

Our framework is based on ML procedures called *solvers* which generalize classical HOL tactics in two ways: (i) they facilitate the manipulation of meta-variables (and their associated substitutions) in the goal[1] and (ii) they allow the proof search to backtrack. Before digging into the description of what a solver is, we warn the reader that the word *goal* has a different meaning in miniKanren and HOL. For the former, a goal is a function that consumes a substitution and produces a stream of substitutions; for the latter, a goal is a pair of (already proved) assumptions and a term that still has to be proved. From now on, we will use the word *goal* in the sense of HOL.

For the sake of completeness, it is worth to describe the differences among goals, tactics, and solvers.

On the one hand, the refinements that a miniKanren goal does on substitutions are performed by a HOL tactic which takes a HOL goal apart into a tuple $(\mathcal{M}, \mathcal{S}, f)$, where $\mathcal{M}$ is a set collecting the introduced meta-variables so far, $\mathcal{S}$ is a list of (sub)goals, and $f$ is a function that certifies the performed refinement. The usual HOL routine is to push and pop those tuples in a stack that represents the steps left to prove the claimed term – whenever the stack gets empty, the proof is completed.

On the other hand, a *solver* is a function that consumes a HOL goal as well as a tactic does, and produces a *stream* of such tuples that actually allows us to equip HOL Light with backtracking. To tie the knot, solvers extend tactics in the sense that every HOL tactic can be "promoted" into a solver using the ML function

```
TACTIC_SLV : tactic -> solver
```

We provide a library of basic solvers, usually having a name that ends in `_SLV`. For the rest of the paper, the following elementary solvers

[1]The tactic mechanism currently implemented in HOL Light already provides basic support for meta-variables in goals. However, it seems to be used only internally in the implementation of the intuitionistic tautology prover `ITAUT_TAC`.

- `RULE_SLV : thm -> solver`, that implements the backward chaining rule;
- `ACCEPT_SLV : thm -> solver`, that solves a goal by unifying with the supplied theorem;
- `CONJ_SLV : solver`, that splits a goal using the introduction rule of the conjunction;
- `REFL_SLV : solver`, that solves a goal which is an equation by unifying of the left- and right-hand sides;
- `ALL_SLV : solver`, that leaves the goal unmodified.

Please note that, as in miniKanren systems, the unification procedure employed is not hard-wired by our framework, and each solver can implement its own unification strategy. We see two main interesting variants that one would have at disposal. The first one is to use pattern matching instead of unification; this would allow for a mechanism of input/output modes as in certain Prolog implementations. The second one would be to use a higher-order unification algorithm to unleash the full expressivity of the underlying higher-order language.

*Solvers are highly compositional*, as tactics in HOL and goals in miniKanren are, and complex solvers can be built from simpler ones using high-order functions. For instance, given two solvers $s_1$ and $s_2$ the solver combinator `CONCAT_SLV` make a new solver that collect sequentially all solutions of $s_1$ followed by all solutions of $s_2$. This is the most basic construction for introducing backtracking into the proof strategy. The solver `COLLECT_SLV` iterates `CONCAT_SLV` over a list of solvers. Two other high-order solvers are (i) `THEN_SLV : solver -> solver -> solver` which combines sequentially two solvers and (ii) `REPEAT_SLV : solver -> solver` that keeps applying a given solver. Unlike Prolog, miniKanren uses a complete search strategy by default and that is provided in our system as well by the solver

```
let INTERLEAVE_SLV (slvl:solver list) : solver =
  if slvl = [] then NO_SLV else
  mergef_stream slvl [];;
```

that relies on the stream combinator

```
mergef_stream : ('b -> 'a stream) list -> (unit -> 'a stream) list -> 'b -> 'a stream
```

which merges two lists of streams by interleaving each one of them.

*Solvers (as for classical HOL tactics) can be used interactively* by means of the following essential commands:

- `gg` ⟨*term*⟩ starts a new goal;
- `ee` ⟨*solver*⟩ applies a solver to the current goal state;
- `bb ()` restores the previous goal state (i.e., undo the previous `ee` command);
- `top_thms ()` returns the stream of solutions found.

Here is an example of interaction. We first introduce the goal, notice the use of the binder `(??)` for the meta-variable `x`:

```
# gg `??x. 2 + 2 = x`;;
val it : mgoalstack =
`2 + 2 = x`
Metavariables: `x`,
```

one possible solution is by using reflexivity that closes the proof

```
# ee REFL_SLV;;
val it : mgoalstack = No sub(m)goals
```

and allows us to form the resulting theorem

```
# list_of_stream(top_thms());;
val it : (instantiation * thm) option list =
  [Some (([], [(`2 + 2`, `x`)], []), |- 2 + 2 = 2 + 2)]
```

Now, if one want to find a different solution, we can restore the initial state

```
# bb();;
val it : mgoalstack =
`2 + 2 = x`
Metavariables: `x`,
```

then use a different solver, for instance by unifying with the equational theorem `|- 2 + 2 = 4`, which can be automatically proved using the HOL procedure `ARITH_RULE`,

```
# ee (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`));;
val it : mgoalstack = No sub(m)goals
```

and, again, take the resulting theorem

```
# list_of_stream(top_thms());;
val it : (instantiation * thm) option list =
  [Some (([], [(`4`, `x`)], []), |- 2 + 2 = 4)]
```

Finally, we can change the proof strategy to find both solutions by using backtracking

```
# bb();;
val it : mgoalstack =
`2 + 2 = x`
Metavariables: `x`,

# ee (CONCAT_SLV REFL_SLV (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`)));;
val it : mgoalstack = No sub(m)goals

# list_of_stream(top_thms());;
val it : (instantiation * thm) option list =
  [Some (([], [(`2 + 2`, `x`)], []), |- 2 + 2 = 2 + 2);
   Some (([], [(`4`, `x`)], []), |- 2 + 2 = 4)]
```

The function `solve : solver -> term -> (term list * thm) stream` runs the proof search non interactively and produces a list of solutions as already shown in Section 3. In this last case it would be

```
# list_of_stream (
    solve (CONCAT_SLV REFL_SLV (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`)))
          `??x. 2 + 2 = x`);;
val it : ((term * term) list * thm) list =
  [([(`2 + 2`, `x`)], |- 2 + 2 = 2 + 2); ([(`4`, `x`)], |- 2 + 2 = 4)]
```

## 5 CASE STUDY: EVALUATION FOR A LISP-LIKE LANGUAGE

The material in this section is strongly inspired by the ingenious work of Byrd, Holk, and Friedman about the miniKanren system [3], where the authors work with the semantics of the Scheme language. Here we target a dynamically scoped variant of the LISP language –not unlike it is done in [2]– formalized as an object language inside the HOL prover. The HOL prover could be a powerful tool for a formal study of the meta-theory of a programming language such as LISP. In this perspective, this section may have a scientific interest beyond the entertaining nature of the example it is going to present.

First, we need to extend our HOL Light environment with an object datatype `sexp` for encoding S-expressions according to the following BNF grammar

```
sexp ::= Symbol string
       | List (sexp list)
```

For instance, the sexp `(list a (quote b))` is represented as HOL term with

```
`List [Symbol "list";
       Symbol "a";
       List [Symbol "quote";
             Symbol "b"]]`
```

This syntactic representation can be hard to read and gets quickly cumbersome as the size of the terms grows. Hence, we also introduce a notation for concrete sexp terms, which is activated by the syntactic pattern `'(...)`. For instance, the above example is written in the HOL concrete syntax for terms as

```
`'(list a (quote b))`
```

With this setup, we can easily specify the evaluation rules for our minimal LISP-like language. This is a ternary predicate `` `EVAL e x y` `` which satisfies the following clauses:

(1) quoted expressions

```
# EVAL_QUOTED;;
|- !e q. EVAL e (List [Symbol "quote"; q]) q
```

(2) variables

```
# EVAL_SYMB;;
|- !e a x. RELASSOC a e x ==> EVAL e (Symbol a) x
```

(3) lambda abstractions

```
# EVAL_LAMBDA;;
|- !e l. EVAL e (List (CONS (Symbol "lambda") l))
                 (List (CONS (Symbol "lambda") l))
```

(4) lists

```
# EVAL_LIST;;
|- !e l l'. ALL2 (EVAL e) l l'
            ==> EVAL e (List (CONS (Symbol "list") l)) (List l')
```

(5) unary applications

```
# EVAL_APP;;
|- !e f x x' v b y.
     EVAL e f (List [Symbol "lambda"; List[Symbol v]; b]) /\
     EVAL e x x' /\ EVAL (CONS (x',v) e) b y
     ==> EVAL e (List [f; x]) y
```

The predicate `` `EVAL` `` is inductively definided, i.e., it is (informally) the *smallest* predicate that satisfies the above rules.

We now use our framework for running a certified evaluation process for this language. First, we define a solver for a single step of computation

```
let STEP_SLV : solver =
  COLLECT_SLV
    [CONJ_SLV;
     ACCEPT_SLV EVAL_QUOTED;
     THEN_SLV (RULE_SLV EVAL_SYMB) RELASSOC_SLV;
     ACCEPT_SLV EVAL_LAMBDA;
     RULE_SLV EVAL_LIST;
     RULE_SLV EVAL_APP;
```

```
    ACCEPT_SLV ALL2_NIL;
    RULE_SLV ALL2_CONS];;
```

In the above code, we collect the solutions of several different solvers. Other than the five rules of the EVAL predicate, we include specific solvers for conjunctions and the two predicates REL_ASSOC and ALL2.

Let us mention that the definition of solvers such us STEP_SLV above could be automatically derived from the set of clauses by performing a syntactical analysis. However, we did not invest time so far on this kind of improvements, since we are still experimenting with the basis of the system.

The top-level recursive solver for the whole evaluation predicate is now easy to define:

```
let rec EVAL_SLV : solver =
   fun g -> CONCAT_SLV ALL_SLV (THEN_SLV STEP_SLV EVAL_SLV) g;;
```

Let us make a simple test. The evaluation of the expression

```
((lambda (x) (list x x x)) (list))
```

can be obtained as follows:

```
# get (solve EVAL_SLV
             `??ret. EVAL []
                         '((lambda (x) (list x x x)) (list))
                         ret`);;

val it : term list * thm =
  ([`ret = '(() () ())`],
   |- EVAL [] '((lambda (x) (list x x x)) (list)) '(() () ()))
```

Again, we can use the declarative nature of logic programs to run the computation backwards. For instance, one intriguing exercise is the generation of quine programs, that is, programs that evaluate to themselves. In our formalization, they are those terms $q$ satisfying the relation `` `EVAL [] q q` ``. The following command computes the first two quines found by our solver.

```
# let sols = solve EVAL_SLV `??q. EVAL [] q q`);;
# take 2 sols;;

val it : (term list * thm) list =
  [([`q = List (Symbol "lambda" :: _3149670)`],
    |- EVAL [] (List (Symbol "lambda" :: _3149670))
       (List (Symbol "lambda" :: _3149670)));
   ([`q =
      List
      [List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
       List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]]]`],
    |- EVAL []
       (List
       [List
        [Symbol "lambda"; List [Symbol _3220800];
```

```
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
        List
        [Symbol "lambda"; List [Symbol _3220800];
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]]])
       (List
       [List
        [Symbol "lambda"; List [Symbol _3220800];
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
        List
        [Symbol "lambda"; List [Symbol _3220800];
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]]])]
```

One can easily observe that any lambda expression is trivially a quine for our language. This is indeed the first solution found by our search:

```
([`q = List (Symbol "lambda" :: _3149670)`],
 |- EVAL []
         (List (Symbol "lambda" :: _3149670))
         (List (Symbol "lambda" :: _3149670)))
```

The second solution is more interesting. Unfortunately, it is presented in a form that is hard to decipher. A simple trick can help us to present this term as a concrete sexp term: it is enough to replace the HOL generated variable (`_3149670`) with a concrete string. This can be done by an ad-hoc substitution:

```
# let [_; i2,s2] = take 2 sols;;
# vsubst [`"x"`,hd (frees (rand (hd i2)))] (hd i2);;

val it : term =
  `q = '((lambda (x) (list x x)) (lambda (x) (list x x)))`
```

If we take one more solution from `sols` stream, we get a new quine which, interestingly enough, is precisely the one obtained in [3]:

```
val it : term =
  `q =
   '((quote (lambda (x) (list x (list (quote quote) x))))
     (quote (quote (lambda (x) (list x (list (quote quote) x))))))`
```

## 6 DESCRIPTION OF OUR CODE

The HOL Light theorem prover and our extension are written in OCaml and, more precisely, in a rather minimal and conservative subdialect of it, which should be understandable to everyone that has some familiarity with any of the languages of the ML family. Our code is available from a public repository, in particular, a release has been created at https://github.com/massimo-nocentini/kanren-light/releases/tag/miniKanren2020 .

Besides the code presented in this article, the above repository contains some other experiments of various nature, including the following:

- An implementation of the Quicksort algorithm. The procedure outputs the sorted list together with a formal proof that such list is indeed sorted and in bijection with the input lists.
- A solver for the *Monte Carlo Lock*, a brain teaser by Smullyan [13], where one has to unlock a *safe* whose *key* is the fixed point of an abstract machine. The interesting thing is that the solver is essentially derived from the formal specification in HOL of the puzzle.

- An intuitionistic first-order tautology prover `ITAUT_SLV`. This is inspired by a similar tactic `ITAUT_TAC` already available in HOL Light.[2] However, HOL tactics cannot backtrack, which implies that `ITAUT_TAC` is incomplete. Our solver `ITAUT_SLV` is coded in pretty much the same way as `ITAUT_TAC`, but it is complete (although this latter fact can be claimed only via a meta-theoretical analysis).

With respect to the existing framework of HOL Light, our effort didn't apply any change to both existing structures and computation flow, it just adds a parallel way of proving things. The connection point is the `type mgoal = term list * goal` that enhances a `goal` with a list of meta-variables and, eventually, all the complexity of the presented work lies in their correct bookkeeping and in the handling of goal streams.

Our code is conceived for experimenting, and very little or no attention has been paid to optimizations. Despite this, the OCaml runtime and the HOL Light implementation have an established reputation of being time- and memory-efficient systems (compared with similar tools). From our informal tests, it seems that this efficiency is, at least partially, inherited by our implementation.

## 7 FUTURE AND RELATED WORK

We presented a rudimentary framework inspired by miniKanren systems implemented on top of the HOL Light theorem prover that enables a logic programming paradigm for proof searching. More specifically, it facilitates the use of meta-variables in HOL goals and permits backtracking during the proof construction. Despite the simplicity of the present implementation, we have already shown the implementation of some paradigmatic examples of logic-oriented proof strategies.

It would be interesting to enhance our framework with more features:

- Implement higher-order unification as Miller's higher-order patterns, so that our system can enable higher-order logic programming in the style of $\lambda$Prolog [4].
- Support constraint logic programming [8], e.g., by adapting the data structure that represents goals.

Besides extending our system with new features, we plan to test it on further examples. One natural domain of applications would be the development of decision procedures. While HOL Light already offers some remarkable tools for automatic theorem proving, our system could offer new alternatives leaning to simplicity and compositionality. For instance, we could try to translate in our system the approach of $\alpha$lean*TAP* [10] for implementing an automatic procedure for first-order classical logic in HOL Light analogous to the `blast` tactic of Paulson [12] in Isabelle.

## REFERENCES

[1] William E. Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. USA. Advisor(s) Friedman, Daniel P.

[2] 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

[3] 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* (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

[4] Amy P. Felty, Elsa L. Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. 1988. Lambda-Prolog: An Extended Logic Programming Language. In *Proceedings of the 9th International Conference on Automated Deduction*. Springer-Verlag, Berlin, Heidelberg, 754–755.

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

[6] John Harrison. 1996. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*. Springer-Verlag, Berlin, Heidelberg, 265–269.

[2] The tactic `ITAUT_TAC` has a peculiar role in the HOL system. It is used during the *bootstrap* of the system to prove several basic logical lemmas. After the intial stages, a much more powerful and faster procedure for (classical) first-order logic `MESON_TAC` is installed in the system, and the `ITAUT_TAC` becomes superfluous.

[7] Jason Hemann and Daniel P. Friedman. 2013. microKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming* (Alexandria, VA) *(Scheme '13)*.

[8] Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. *Electronic Proceedings in Theoretical Computer Science* 234 (Jan 2017), 135–149. https://doi.org/10.4204/eptcs.234.10

[9] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1 (1991), 253–281.

[10] Joseph P. Near, William E. Byrd, and Daniel P. Friedman. 2008. $\alpha$leanTAP: A Declarative Theorem Prover for First-Order Classical Logic. In *Logic Programming*, Maria Garcia de la Banda and Enrico Pontelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–252.

[11] Lawrence Paulson. 1983. A higher-order implementation of rewriting. *Science of Computer Programming* 3, 2 (1983), 119 – 149. https://doi.org/10.1016/0167-6423(83)90008-4

[12] Lawrence C. Paulson. 1999. A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science* 5, 3 (mar 1999), 73–87. http://www.jucs.org/jucs_5_3/a_generic_tableau_prover

[13] Raymond M. Smullyan. 2009. *The Lady Or the Tiger?: And Other Logic Puzzles*. Dover Publications.

# Certified Semantics for Disequality*

DMITRY ROZPLOKHAS, Higher School of Economics, JetBrains Research, Russia
DMITRY BOULYTCHEV, Saint Petersburg State University, JetBrains Research, Russia

We present an extension of our prior work on certified semantics for core miniKanren, introducing disequality constraints in the language. Semantics is parameterized by an exact definition of constraint stores, allowing us to cover different implementations, and we provide a list of sufficient conditions on this definition for search completeness. We also give two examples of concrete implementations of constraint stores that satisfy those sufficient conditions. The description and proofs for parameterized semantics and both implementations are certified in Coq and two correct-by-construction interpreters are extracted.

## 1 INTRODUCTION

In its initial form [Friedman et al. 2005; Hemann and Friedman 2013] miniKanren introduces a single form of constraint — unification of terms. While from a theoretical standpoint unification together with other primitive constructs (conjunction, disjunction, and fresh variable introduction) form a Turing-complete basis, in practice of relational programming a number of extensions are often used to make specifications more expressive, concise or efficient. One of the most important extensions is *disequality constraint*.

A generic concept of domain-specific constraints in logic programming is studied in details in [Jaffar et al. 1998]; more specifically, disequality constraint [Comon-Lundh 1991] introduces one additional type of base goal — a disequality of two terms

$$t_1 \not\equiv t_2$$

The informal semantics of disequality constraint is complementary to that of unification: it puts certain restrictions on free variables in the terms which prevent them from turning into syntactically equal. Similarly to unification, whose evaluation results in a substitution, which is then threaded through the rest of computations, the effect of disequality constraint is recorded in a *constraint store* which is later used to check the violation of disequalities [Alvis et al. 2011].

We present an extension of our prior work on certified semantics for core miniKanren [Rozplokhas et al. 2019]. In that work, we defined denotational and operational semantics and proved the soundness and completeness of the latter w.r.t. the former. The main advantage of the operational semantics introduced there over the ones developed before [Kumar 2010; Lozov et al. 2017; Rozplokhas and Boulytchev 2018] was its ability to capture the conventional for miniKanren *interleaving search* [Kiselyov et al. 2005] procedure. This allowed us to give the first to our knowledge formal proof of completeness of the interleaving search as the capability to reach all the solutions from denotational semantics (proof of completeness as the fairness of steams interleaving for a

*The reported study was funded by RFBR, project number 18-01-00380

Authors' addresses: Dmitry Rozplokhas, Higher School of Economics, JetBrains Research, Russia, rozplokhas@gmail.com; Dmitry Boulytchev, Saint Petersburg State University, JetBrains Research, Russia, dboulytchev@math.spbu.ru.


miniKanren.org/workshop/2021/8-ART6

specific implementation was given in [Hemann et al. 2016]). The development was formally certified in Coq proof assistant [Bertot and Castéran 2004], and a correct-by-construction interpreter was extracted.

To some extent our work follows the conventional roadmap of adding constraints to a pure logic/relational language [Jaffar et al. 1998]; the difference is that, first, we use a specific constraint and a concrete solver, and second, we prove all the results with regard to conventional for miniKanren interleaving search (versus a very generic and abstract breadth-first search in [Jaffar et al. 1998]).

The contribution of our current work is as follows:

- we extend our denotational semantics to handle disequality constraints;
- we introduce a new abstraction layer (a constraint store with a number of abstract operations) in our operational semantics;
- we formulate a set of *sufficient conditions for completeness*, expressed as algebraic properties of constraint store and abstract operators, and prove the soundness and completeness of the extended operational semantics w.r.t. the denotational one;
- we present two concrete implementations of constraint store and abstract operators and show that they satisfy the sufficient conditions; thus, the soundness and completeness of the implementation with disequality constraints follow immediately, and correct-by-construction interpreter for miniKanren with disequality constraints can be extracted
- we demonstrate how our framework can be used to prove some properties of implementations of disequality constraints.

The paper is organized as follows. In Section 2 we recall our framework from the previous paper [Rozplokhas et al. 2019], which is extended in this work. The following sections describe the new results. Section 3 contains the description of the extensions in semantics and sufficient conditions on abstract definitions for search completeness. Section 4 contains two examples of implementations of constraint stores that satisfy the sufficient conditions for completeness. Section 5 presents some applications of the extended semantics. The final section concludes.

## 2 THE SYNTAX AND SEMANTICS OF THE CORE LANGUAGE

In this section, we recall existing definitions of the syntax and the two semantics for the core language without disequality constraints and the main result — the equivalence of these two semantics [Rozplokhas et al. 2019].

### 2.1 The Syntax of Core Language

The syntax of the language is shown in Fig. 1. First, we fix a set of constructors $C$ with known arities and consider a set of terms $\mathcal{T}_X$ with constructors as functional symbols and variables from $X$. We parameterize this set with an alphabet of variables since in the semantic description we will need *two* kinds of variables. The first kind, *syntactic* variables, is denoted by $\mathcal{X}$. We also consider an alphabet of *relational symbols* $\mathcal{R}$ which are used to name relational definitions. The central syntactic category in the language is a *goal*. In our case, there are five types of goals: *equality* of terms, conjunction and disjunction of goals, fresh variable introduction, and invocation of some relational definition. Thus, equality is used as a constraint, and multiple constraints can be combined using conjunction, disjunction, and recursion. For the sake of brevity we abbreviate immediately nested "**fresh**" constructs into the one, writing "**fresh** $x\ y\ldots.\ g$" instead of "**fresh** $x$ . **fresh** $y$ . $\ldots\ g$". The final syntactic category is *specification* $\mathcal{S}$. It consists of a set of relational definitions and a top-level goal. A top-level goal represents a search procedure which returns a stream of substitutions for the free variables of the goal. The language we defined is first-order, as goals can not be passed as parameters, returned or constructed at runtime.

As an example consider the specification for the standard `append`$^o$ relation and a query which splits a list containing three constants `A`, `B` and `C` into two parts in every possible way:

$$
\begin{array}{lcll}
\mathcal{C} & = & \{C_i^{k_i}\} & \text{constructors with arities} \\
\mathcal{T}_X & = & X \cup \{C_i^{k_i}(t_1, \ldots, t_{k_i}) \mid t_j \in \mathcal{T}_X\} & \text{terms over the set of variables } X \\
\mathcal{D} & = & \mathcal{T}_\varnothing & \text{ground terms} \\
\mathcal{X} & = & \{x, y, z, \ldots\} & \text{syntactic variables} \\
\mathcal{A} & = & \{\alpha, \beta, \gamma, \ldots\} & \text{semantic variables} \\
\mathcal{R} & = & \{R_i^{k_i}\} & \text{relational symbols with arities} \\
\mathcal{G} & = & \mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X} & \text{equality} \\
 & & \mathcal{G} \wedge \mathcal{G} & \text{conjunction} \\
 & & \mathcal{G} \vee \mathcal{G} & \text{disjunction} \\
 & & \textbf{fresh } \mathcal{X} \text{ . } \mathcal{G} & \text{fresh variable introduction} \\
 & & R_i^{k_i}(t_1, \ldots, t_{k_i}),\ t_j \in \mathcal{T}_\mathcal{X} & \text{relational symbol invocation} \\
\mathcal{S} & = & \{R_i^{k_i} = \lambda\ x_1^i \ldots x_{k_i}^i \text{ . } g_i;\}\ g & \text{specification}
\end{array}
$$

Fig. 1. The syntax of core language

```
append^o = λ x y xy .
  ((x ≡ Nil) ∧ (xy ≡ y)) ∨
  (fresh h t ty .
     (x  ≡ Cons (h, t))  ∧
     (xy ≡ Cons (h, ty)) ∧
     (append^o t y ty)
  );
append^o x y (Cons (A, Cons (B, Cons (C, Nil))))
```

## 2.2 Denotational sematics

For denotational semantics, we use a simple set-theoretic approach which can be considered analogous to the least Herbrand model for definite logic programs [Lloyd 1984].

Intuitively, the mathematical model for every goal should be a relation between semantic variables that occur free in this goal. We represent this relation as a set of total functions

$$\mathfrak{f} : \mathcal{A} \to \mathcal{D}$$

from semantic variables to ground terms. We call these functions *representing functions*.

Then, the semantic function for goals is parameterized over environments which prescribe semantic functions to relational symbols:

$$\Gamma : \mathcal{R} \to ({\mathcal{T}_\mathcal{A}}^* \to 2^{\mathcal{A} \to \mathcal{D}})$$

An environment associates with relational symbol a function that takes a string of terms (the arguments of the relation) and returns a set of representing functions. The signature for semantic brackets for goals is as follows:

$$[\![\bullet]\!]_\Gamma : \mathcal{G} \to 2^{\mathcal{A} \to \mathcal{D}}$$

It maps a goal into the set of representing functions w.r.t. an environment $\Gamma$.

$$
\begin{array}{rcll}
[\![t_1 \equiv t_2]\!]_\Gamma & = & \{\mathfrak{f} : \mathcal{A} \to \mathcal{D} \mid \bar{\mathfrak{f}}\,(t_1) = \bar{\mathfrak{f}}\,(t_2)\} & [\textsc{Unify}_D] \\
[\![g_1 \wedge g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cap [\![g_1]\!]_\Gamma & [\textsc{Conj}_D] \\
[\![g_1 \vee g_2]\!]_\Gamma & = & [\![g_1]\!]_\Gamma \cup [\![g_1]\!]_\Gamma & [\textsc{Dis}_D] \\
[\![\textbf{fresh}\ x\,.\,g]\!]_\Gamma & = & ([\![g\,[\alpha/x]]\!]_\Gamma) \uparrow \alpha,\ \alpha \notin FV(g) & [\textsc{Fresh}_D] \\
[\![R(t_1, \ldots, t_k)]\!]_\Gamma & = & (\Gamma\, R)\, t_1 \ldots t_k & [\textsc{Invoke}_D]
\end{array}
$$

Fig. 2. Denotational semantics of goals

We formulate the following important *completeness condition* for the semantics of a goal $g$: for any goal $g$ and two representing functions $\mathfrak{f}$ and $\mathfrak{f}'$, such that $\mathfrak{f}|_{FV(g)} = \mathfrak{f}'|_{FV(g)}$

$$\mathfrak{f} \in [\![g]\!] \Leftrightarrow \mathfrak{f}' \in [\![g]\!]$$

In other words, representing functions for a goal $g$ restrict only the values of free variables of $g$ and do not introduce any "hidden" correlations. This condition guarantees that our semantics is complete in the sense that it does not introduce artificial restrictions for the relation it defines. We proved that the semantics of goals always satisfy this condition.

To define the semantic function we need a few operations for representing functions:

- A homomorphic extension of a representing function

  $$\bar{\mathfrak{f}} : \mathcal{T}_\mathcal{A} \to \mathcal{D}$$

  which maps terms to terms:

  $$
  \begin{array}{rcl}
  \bar{\mathfrak{f}}\,(\alpha) & = & \mathfrak{f}\,(\alpha) \\
  \bar{\mathfrak{f}}\,(C_i^{k_i}\,(t_1, \ldots . t_{k_i})) & = & C_i^{k_i}\,(\bar{\mathfrak{f}}\,(t_1), \ldots \bar{\mathfrak{f}}\,(t_{k_i}))
  \end{array}
  $$

- A pointwise modification of a function

  $$f\,[x \leftarrow v]\,(z) = \begin{cases} f\,(z) & , \quad z \neq x \\ v & , \quad z = x \end{cases}$$

- A *generalization* operation:

  $$\mathfrak{f} \uparrow \alpha = \{\mathfrak{f}\,[\alpha \leftarrow d] \mid d \in \mathcal{D}\}$$

  Informally, this operation generalizes a representing function into a set of representing functions in such a way that the values of these functions for a given variable cover the whole $\mathcal{D}$. We extend the generalization operation for sets of representing functions $\mathfrak{F} \subseteq \mathcal{A} \to \mathcal{D}$:

  $$\mathfrak{F} \uparrow \alpha = \bigcup_{\mathfrak{f} \in \mathfrak{F}} (\mathfrak{f} \uparrow \alpha)$$

The semantics for goals is shown on Fig. 2.

The final component is the semantics of specifications. Given a specification

$$\{R_i = \lambda\, x_1^i \ldots x_{k_i}^i\,.\,g_i;\}_{i=1}^n\ g$$

we construct a correct environment $\Gamma_0$ and then take the semantics of the top-level goal:

$$[\![\{R_i = \lambda\, x_1^i \ldots x_{k_i}^i\,.\,g_i;\}_{i=1}^n\ g]\!] = [\![g]\!]_{\Gamma_0}$$

As the set of definitions can be mutually recursive we apply the fixed point approach and define $\Gamma_0$ as the least fixed point of a specific function $F$ that takes an environment $\Gamma$ and returns new environment in which semantics of a body of each definition is evaluated with environment $\Gamma$.

## 2.3 Operational sematics

The operational semantics of miniKanren, which we described, corresponds to the known implementations with interleaving search. The semantics is given in the form of a labeled transition system (LTS) [Keller 1976].

The states in the transition system have the following shape:

$$S = \mathcal{G} \times \Sigma \times \mathbb{N} \mid S \oplus S \mid S \otimes \mathcal{G}$$

A state has a tree-like structure with intermediate nodes corresponding to partially-evaluated conjunctions ("$\otimes$") or disjunctions ("$\oplus$"). A leaf in the form $\langle g, \sigma, n\rangle$ determines a goal in a context, where $g$ — a goal, $\sigma$ — a substitution accumulated so far, and $n$ — a natural number, which corresponds to a number of semantic variables used to this point. For a conjunction node, its right child is always a goal since it cannot be evaluated unless some result is provided by the left conjunct.

We also need extended states

$$\overline{S} = \diamond \mid S$$

where $\diamond$ symbolizes the end of the evaluation.

The set of labels is defined as follows:

$$L = \circ \mid \Sigma \times \mathbb{N}$$

The label "$\circ$" is used to mark those steps which do not provide an answer; otherwise, a transition is labeled by a pair of a substitution and a number of allocated variables. The substitution is one of the answers, and the number is threaded through the derivation to keep track of the allocated variables.

The transition rules are shown in Fig. 3. The introduced transition system is completely deterministic.

A derivation sequence for a certain state $s$ determines a *trace* $\mathcal{T}r_s$ — a finite or infinite sequence of answers. The trace corresponds to the stream of answers in the reference miniKanren implementations.

## 2.4 Semantics Equivalence

After we defined two different kinds of semantics for miniKanren we related them and showed that the results given by these two semantics are the same for any specification. By proving this equivalence we established the *completeness* of the search which means that the search will get all answers satisfying the described specification and only those.

To do it we had to relate the answers produced by these two semantics as they have different forms: a trace of substitutions (along with numbers of allocated variables) for operational and a set of representing functions for denotational. There is a natural way to extend any substitution to a representing function: composing it with an arbitrary representing function will preserve all variable dependencies in the substitution. So we defined a set of representing functions corresponding to substitution as follows:

$$[\![\sigma]\!] = \{\bar{\mathfrak{f}} \circ \sigma \mid \mathfrak{f} : \mathcal{A} \to \mathcal{D}\}$$

And *denotational analog* of an operational semantics (a set of representing functions corresponding to answers in the trace) for given extended state $s$ is then defined as a union of sets for all substitution in the trace:

$$[\![s]\!]_{op} = \cup_{(\sigma,n)\in\mathcal{T}r_s} [\![\sigma]\!]$$

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{\circ} \diamond,\ \nexists\, mgu\,(t_1\sigma, t_2\sigma) \qquad \text{[UnifyFail]}$$

$$\langle t_1 \equiv t_2, \sigma, n\rangle \xrightarrow{(mgu\,(t_1\sigma, t_2\sigma)\circ\sigma),\ n)} \diamond \qquad \text{[UnifySuccess]}$$

$$\langle g_1 \vee g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \oplus \langle g_2, \sigma, n\rangle \qquad \text{[Disj]}$$

$$\langle g_1 \wedge g_2, \sigma, n\rangle \xrightarrow{\circ} \langle g_1, \sigma, n\rangle \otimes g_2 \qquad \text{[Conj]}$$

$$\langle \textbf{fresh}\ x\,.\,g, \sigma, n\rangle \xrightarrow{\circ} \langle g\,[\alpha_{n+1}\,/_x], \sigma, n+1\rangle \qquad \text{[Fresh]}$$

$$\frac{R_i^{k_i} = \lambda\, x_1 \ldots x_{k_i}\,.\,g}{\left\langle R_i^{k_i}\,(t_1, \ldots, t_{k_i}), \sigma, n\right\rangle \xrightarrow{\circ} \left\langle g\,[t_1\,/_{x_1}\ldots t_{k_i}\,/_{x_{k_i}}], \sigma, n\right\rangle} \qquad \text{[Invoke]}$$

$$\frac{s_1 \xrightarrow{\circ} \diamond}{(s_1 \oplus s_2) \xrightarrow{\circ} s_2} \qquad \text{[DisjStop]}$$

$$\frac{s_1 \xrightarrow{r} \diamond}{(s_1 \oplus s_2) \xrightarrow{r} s_2} \qquad \text{[DisjStopAns]}$$

$$\frac{s \xrightarrow{\circ} \diamond}{(s \otimes g) \xrightarrow{\circ} \diamond} \qquad \text{[ConjStop]}$$

$$\frac{s \xrightarrow{(\sigma, n)} \diamond}{(s \otimes g) \xrightarrow{\circ} \langle g, \sigma, n\rangle} \qquad \text{[ConjStopAns]}$$

$$\frac{s_1 \xrightarrow{\circ} s_1'}{(s_1 \oplus s_2) \xrightarrow{\circ} (s_2 \oplus s_1')} \qquad \text{[DisjStep]}$$

$$\frac{s_1 \xrightarrow{r} s_1'}{(s_1 \oplus s_2) \xrightarrow{r} (s_2 \oplus s_1')} \qquad \text{[DisjStepAns]}$$

$$\frac{s \xrightarrow{\circ} s'}{(s \otimes g) \xrightarrow{\circ} (s' \otimes g)} \qquad \text{[ConjStep]}$$

$$\frac{s \xrightarrow{(\sigma, n)} s'}{(s \otimes g) \xrightarrow{\circ} (\langle g, \sigma, n\rangle \oplus (s' \otimes g))} \qquad \text{[ConjStepAns]}$$

Fig. 3. Operational semantics of interleaving search

This allowed us to state the theorem relating two semantics.

Theorem 1 (Operational semantics soundness and completeness). *For any specification* $\{\ldots\}\ g$*, for which the indices of all free variables in* $g$ *are limited by some number* $n$

$$[\![\langle g, \epsilon, n\rangle]\!]_{op} =_n [\![\{\ldots\}\ g]\!].$$

Where '$=_n$' means that we compare representing functions of these sets only on the semantic variables from $\{\alpha_1, \ldots, \alpha_n\}$:

$$S_1 =_n S_2 \overset{def}{\iff} \{\bar{\mathfrak{f}}|_{\{\alpha_1,\ldots,\alpha_n\}} \mid \bar{\mathfrak{f}} \in S_1\} = \{\bar{\mathfrak{f}}|_{\{\alpha_1,\ldots,\alpha_n\}} \mid \bar{\mathfrak{f}} \in S_2\}.$$

We can not use the usual equality of sets instead of this one, the sets from the theorem statement are actually not equal. The reason for this is that denotational semantics encodes only dependencies between the free variables of a goal, which is reflected by the completeness condition, while operational semantics may also contain dependencies between semantic variables allocated in "**fresh**". Therefore we have to restrict representing functions on the semantic variables allocated in the beginning (which includes all free variables of a goal). This does not compromise our promise to prove the completeness of the search as miniKanren provides the result as substitutions only for queried variables, which are allocated in the beginning.

The proof of this main theorem was certified in Coq.

## 3 EXTENSION WITH DISEQUALITY CONSTRAINTS

In this section, we present extensions of our two semantics for the language with disequality constraints and revised versions of the soundness and completeness theorems.

Disequality constraint introduces one additional type of base goal — a disequality of two terms: $t_1 \not\equiv t_2$

The extension of denotational semantics is straightforward (as disequality constraint is complementary to equality):

$$[\![t_1 \not\equiv t_2]\!] = \{\bar{\mathfrak{f}} \in \mathcal{R} \mid \bar{\mathfrak{f}}(t_1) \neq \bar{\mathfrak{f}}(t_2)\},$$

This definition for a new type of goals fits nicely into the general inductive definition of denotational semantics of an arbitrary goal and preserves its properties, such as completeness condition.

In the operational case we deviate from describing one specific search implementation since there are several distinct ways to embed disequality constraints in the language and we would like to be able to give semantics (and subsequently prove correctness) for all of them. Therefore we base the extended operational semantics on a number of abstract definitions concerning constraint stores for which different concrete implementations may be substituted.

We assume that we are given a set of constraint store objects, which we denote by $\Omega_\sigma$ (indexing every constraint store with some substitution $\sigma$ and assuming the store and the substitution are consistent with each other), and three following operations:

(1) Initial constraint store $\Omega_\epsilon^{init}$ (where $\epsilon$ is empty substitution), which does not contain any constraints yet.
(2) Adding a disequality constraint to a store $\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2)$, which may result in a new constraint store $\Omega'_\sigma$ or a failure $\perp$, if the new constraint store is inconsistent with the substitution $\sigma$.
(3) Updating a substitution in a constraint store $\mathbf{update}\,(\Omega_\sigma, \delta)$ to intergate a new substitution $\delta$ into the current one, which may result in a new constraint store $\Omega'_{\sigma\delta}$ or a failure $\perp$, if the constraint store is inconsistent with the new substitution.

The change in operational semantics for the language with disequality constraints is now straightforward: we add a constraint store to a basic (leaf) state $\langle g, \sigma, \Omega_\sigma, n\rangle$, as well as in the label form $(\sigma, \Omega_\sigma, n)$, and this store is simply threaded through all the rules, except those for equality. We change the rules for equality using **update** operation and add the rules for disequality constraint using **add**. In both cases, the search in the current branch is pruned if these primitives return $\perp$.

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \diamond,\ \nexists\ mgu\,(t_1, t_2, \sigma) \qquad \text{[UnifyFailMGU]}$$

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \diamond,\ mgu\,(t_1, t_2, \sigma) = \delta,\ \mathbf{update}\,(\Omega_\sigma, \delta) = \bot \qquad \text{[UnifyFailUpdate]}$$

$$\langle t_1 \equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{(\sigma\delta,\ \Omega'_{\sigma\delta},\ n)} \diamond,\ mgu\,(t_1, t_2, \sigma) = \delta,\ \mathbf{update}\,(\Omega_\sigma, \delta) = \Omega'_{\sigma\delta} \qquad \text{[UnifySuccess]}$$

$$\langle t_1 \not\equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{\circ} \diamond,\ \mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \bot \qquad \text{[DiseqFail]}$$

$$\langle t_1 \not\equiv t_2, \sigma, \Omega_\sigma, n\rangle \xrightarrow{(\sigma,\ \Omega'_\sigma,\ n)} \diamond,\ \mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega'_\sigma \qquad \text{[DiseqSucess]}$$

The initial state naturally contains an initial constraint store $\left\langle g, \varepsilon, \Omega_\epsilon^{init}, n\right\rangle$.

To state the soundness and completeness result now we need to revise our definition of the denotational analog of an answer $(\sigma, \Omega_\sigma, n)$ since we have to take into account the restrictions which a constraint store $\Omega_\sigma$ encodes. To do this we need one more abstract definition — a denotational interpretation of a constraint store $[\![\Omega_\sigma]\!]$ as a set of representing functions. We prove the soundness and completeness w.r.t. this interpretation and expect it to adequately reflect how the restrictions of constraint stores in the answers are presented. The denotational analog of operational semantics for an arbitrary extended state is then redefined as follows.

$$[\![s]\!]_{op} = \cup_{(\sigma,\Omega_\sigma,n)\in\mathcal{T}r_s}[\![\sigma]\!] \cap [\![\Omega_\sigma]\!]$$

The statement of the soundness and completeness theorem stays the same with regard to this updated definitions of semantics and denotational analog.

Theorem 2 (Operational semantics soundness and completeness for extended language). *For any specification* $\{\dots\}$ *g, for which the indices of all free variables in g are limited by some number n*

$$[\![\left\langle g, \epsilon, \Omega_\epsilon^{init}, n\right\rangle]\!]_{op} =_n [\![\{\dots\}\ g]\!].$$

To be able to prove it we, of course, need certain requirements for the given operations on constraint stores. We came up with the following list of sufficient conditions for soundness and completeness.

(1) $[\![\Omega_\epsilon^{init}]\!] = \{\tilde{\mathfrak{f}} : \mathcal{A} \to \mathcal{D}\}$;
(2) $\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega'_\sigma \implies [\![\Omega_\sigma]\!] \cap [\![t_1 \not\equiv t_2]\!] \cap [\![\sigma]\!] = [\![\Omega'_\sigma]\!] \cap [\![\sigma]\!]$;
(3) $\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \bot \implies [\![\Omega_\sigma]\!] \cap [\![t_1 \not\equiv t_2]\!] \cap [\![\sigma]\!] = \varnothing$;
(4) $\mathbf{update}\,(\Omega_\sigma, \delta) = \Omega'_{\sigma\delta} \implies [\![\Omega_\sigma]\!] \cap [\![\sigma\delta]\!] = [\![\Omega'_{\sigma\delta}]\!] \cap [\![\sigma\delta]\!]$;
(5) $\mathbf{update}\,(\Omega_\sigma, \delta) = \bot \implies [\![\Omega_\sigma]\!] \cap [\![\sigma\delta]\!] = \varnothing$.

These conditions state that given denotational interpretation and given operations on constraint stores are adequate to each other.

Condition 1 states that interpretation of the initial constraint store is the whole domain of representing function since it does not impose any restrictions.

Condition 2 states that when we add a constraint to a store $\Omega_\sigma$ the interpretation of the result contains exactly those functions which simultaneously belong to the interpretation of the store $\Omega_\sigma$ and satisfy the constraint if we consider only extensions of the substitution $\sigma$.

Condition 3 states that addition could fail only if no such functions exist.

Conditions 4 state that the result of updating a store with an additional substitution should have the same interpretation if we consider only extensions of the updated substitution.

Condition 5 states that update could fail only if no such functions exist.

The conditions 2-5 describe exactly what we need to prove the soundness and completeness for base goals (equality and disequality); at the same time, since these conditions have relatively simple intuitive meaning in terms of these two operations they are expected to hold naturally in all reasonable implementations of constraint stores.

We can prove that this is enough for soundness and completeness to hold for an arbitrary goal. However, contrary to our expectations, the existing proof can not be just reused for all non-basic types of goals and has to be modified significantly in the case of **fresh**. Specifically, we need one additional condition on constraint store in state $(\sigma, n, \Omega_\sigma)$: only the values on the first $n$ fresh variables determine whether a representing function belongs to the denotational semantics $[\![\sigma]\!] \cap [\![\Omega_\sigma]\!]$ of the state (note the similarity to the completeness condition). Luckily, we can infer this property for all states that can be constructed by our operational semantics from the sufficient conditions above.

Thus for an arbitrary implementation, we need to give a formal definition of constraint store object and its denotational interpretation, provide three operations for it and prove five conditions on them, and by this, we ensure that for arbitrary specification the interpretations of all solutions found by the search in this version of MiniKanren will cover exactly the mathematical model of this specification.

As well as our previous development this extension is certified in Coq[1]. We describe operational semantics and its soundness and completeness as modules parametrized by the definitions of constraint stores and proofs of the sufficient conditions for them.

## 4 CONCRETE IMPLEMENTATIONS

In this section, we define two concrete implementations of constraint stores which can be incorporated in operational semantics: the trivial one and the one, which is close to existing real implementation in a certain version of miniKanren [Alvis et al. 2011]. We prove that they satisfy the sufficient conditions for search completeness from the previous section. Both implementations are certified in Coq, which allowed us to extract two correct-by-construction interpreters for miniKanren with disequality constraints.

### 4.1 Trivial Implementation

This trivial implementation simply stores all pairs of terms, which the search encounters, in a multiset and never uses them:

$$\Omega_\sigma \subset_m \mathcal{T} \times \mathcal{T}$$

$$\Omega_\epsilon^{init} = \varnothing$$

$$\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \Omega_\sigma \cup \{(t_1, t_2)\}$$

$$\mathbf{update}\,(\Omega_\sigma, \delta) = \Omega_\sigma$$

The interpretation of such constraint store is the set of all representing functions that does not equate terms in any pair:

$$[\![\Omega_\sigma]\!] = \{\bar{\mathsf{f}} \colon \mathcal{A} \to \mathcal{D} \mid \forall (t_1, t_2) \in \Omega_\sigma,\ \bar{\mathsf{f}}\,(t_1) \neq \bar{\mathsf{f}}\,(t_2)\}$$

This is a correct implementation (although for the full implementation we should find a way to present restrictions stored this way in answers adequately) and it satisfies the sufficient conditions for completeness

[1] https://github.com/dboulytchev/miniKanren-coq/tree/disequality

trivially, but it is not very practical. In particular, it does not use information acquired from disequalities to halt the search in case of contradiction and it can return contradictory answers with the final disequality constraint violated by the final substitution (such as $([\alpha_0 \to 5], [\alpha_0 \neq 5], 1)$): since such answers have empty interpretations, their presence does not affect search completeness.

### 4.2 Realistic Implementation

This implementation is more similar to those in existing miniKanren implementations and takes an approach that is close to one described is [Alvis et al. 2011].

In this version, every constraint is represented as a substitution containing variable bindings which should *not* be satisfied.

$$\Omega_\sigma \subset_m \Sigma$$

So if a constraint store $\Omega_\sigma$ contains a substitution $\omega$ the set of representing functions prohibited by it is $[\![\sigma\omega]\!]$, which provides the following denotational interpretation for a constraint store:

$$[\![\Omega_\sigma]\!] = \bigcap_{\omega \in \Omega_\sigma} \overline{[\![\sigma\omega]\!]}$$

We start with an empty store

$$\Omega_\epsilon^{init} = \varnothing$$

When we encounter a disequality for two terms we try to unify them and update constraint store depending on the result of unification:

$$\mathbf{add}\,(\Omega_\sigma, t_1 \not\equiv t_2) = \begin{cases} \Omega_\sigma & \nexists mgu(t_1\sigma, t_2\sigma) \\ \bot & mgu(t_1\sigma, t_2\sigma) = \epsilon \\ \Omega_\sigma \cup \{\omega\} & mgu(t_1\sigma, t_2\sigma) = \omega \neq \epsilon \end{cases}$$

If the terms are not unifiable, there is no need to change the constraint store. If they are unified by current substitution the constraint is already violated and we signal a failure. Otherwise, the most general unifier is an appropriate representation of this constraint.

When updating a constraint store with an additional substitution $\delta$ we try to update each individual constraint substitution by treating it as a list of pairs of terms that should not be unified (the first element of each pair is a variable), applying $\delta$ to these terms and trying to unify all pairs simultaniously:

$$\mathbf{update}_{\mathbf{constr}}\,([x_1 \to t_1, \ldots, x_k \to t_k], \delta) = mgu([\delta(x_1), \ldots, \delta(x_k)], [t_1\delta, \ldots, t_k\delta])$$

We construct the updated constraint store from the results of all constraint updates:

$$\mathbf{update}\,(\Omega_\sigma, \delta) = \begin{cases} \bot & \exists \omega \in \Omega_\sigma : \mathbf{update}_{\mathbf{constr}}\,(\omega, \delta) = \epsilon \\ \{\omega' \mid \mathbf{update}_{\mathbf{constr}}\,(\omega, \delta) = \omega' \neq \bot,\ \omega \in \Omega_\sigma\} & \textit{otherwise} \end{cases}$$

If any constraint is violated by the additional substitution we signal a failure, otherwise we take in the store the updated constraints (and some constraints are thrown away as they can no longer be violated).

We proved the sufficient conditions for completeness for this implementation, too, but it required us to prove first that all substitutions constructed by miniKanren search have a specific form. Namely, a current subsitution $\sigma$ at any point of the search (started from the initial state) is always *narrowing* — which means

that $\mathcal{VR}an(\sigma) \cap \mathcal{D}om(\sigma) = \varnothing$ — and every time a current substitution $\sigma$ is updated by composing with some substitution $\delta$ (in rule [UnifySuccess]) this substitution is *extending* — which means that $\mathcal{D}om(\delta) \cap \mathcal{D}om(\sigma) = \varnothing \wedge \mathcal{VR}an(\delta) \cap \mathcal{D}om(\sigma) = \varnothing$.

## 5 APPLICATIONS

In addition to verification of correctness of different implementations of disequality constraints we can use our framework to formally state and prove some of its other important properties. Thanks to our completeness result, we can do it in the denotational context, where the reasoning is much easier.

For example, we can specify contradictory answers with empty interpretation, which we pointed out for the trivial implementation from the previous section, and prove that there are no such answers in the realistic implementation if and only if there are infinitely many constructors in the language. So, for the realistic implementation the following holds iff the set of constructors is infinite:

Lemma 1. *For any goal $g$, if all free variables in it belong to the set $\{\alpha_1, \ldots, \alpha_n\}$, then*

$$\forall(\sigma, \Omega_\sigma, n_r) \in Tr_{\langle g, \epsilon, \Omega_\epsilon^{init}, n\rangle}, \quad [\![\sigma]\!] \cap [\![\Omega_\sigma]\!] \neq \varnothing.$$

The proof is based on the following lemma about combining constraints, which we can prove we can prove when there are infinitely many constructors (and otherwise it is not true).

Lemma 2. *If for a finite constraint store $\Omega_\sigma$*

$$\forall \omega \in \Omega_\sigma, [\![\sigma]\!] \cap [\![\omega]\!] \neq \varnothing,$$

*then*

$$[\![\sigma]\!] \cap [\![\Omega_\sigma]\!] \neq \varnothing.$$

Another example of application is the justification of optimizations in constraint store implementation. For example, the following obvious (in denotational context) statement allows deleting subsumed constraints in the realistic implementation.

Lemma 3. *For any constraint store $\Omega_\sigma$ and two constraint substitutions $\omega$ and $\omega'$, if*

$$\exists \tau, \omega' = \omega\tau$$

*then*

$$[\![\Omega_\sigma \cup \{\omega, \omega'\}]\!] = [\![\Omega_\sigma \cup \{\omega\}]\!].$$

## 6 CONCLUSION

In this paper we presented an extended version of formal semantics for miniKanren which supports disequality constraints. The semantics is parametrized by an exact implementation of constraint stores and allows us to ensure the correctness of different implementations in a unified way, using the given set of sufficient conditions.

## REFERENCES

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.*

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer. https://doi.org/10.1007/978-3-662-07964-5

Hubert Comon-Lundh. 1991. Disunification: A Survey. In *Computational Logic - Essays in Honor of Alan Robinson.*

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

Jason Hemann and Daniel P. Friedman. 2013. $\mu$Kanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming*.

Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. 96–107. https://doi.org/10.1145/2989225.2989230

Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. 1998. The semantics of constraint logic programs. *The Journal of Logic Programming* 37, 1 (1998), 1 – 46. https://doi.org/10.1016/S0743-1066(98)10002-X

Robert M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (1976), 371–384. https://doi.org/10.1145/360248.360251

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). (2005), 192–203. https://doi.org/10.1145/1086365.1086390

Ramana Kumar. 2010. Mechanising Aspects of miniKanren in HOL. Bachelor Thesis, The Australian National University.

John W. Lloyd. 1984. *Foundations of Logic Programming, 1st Edition*. Springer.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed Relational Conversion. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. 39–58. https://doi.org/10.1007/978-3-319-89719-6_3

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. 18:1–18:13. https://doi.org/10.1145/3236950.3236958

Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2019. Certified Semantics for miniKanren. In *miniKanren and Relational Programming Workshop*.

# mediKanren: a System for Biomedical Reasoning

WILLIAM E. BYRD, GREGORY ROSENBLATT, MICHAEL J. PATTON, and THI K. TRAN-NGUYEN, Hugh Kaul Precision Medicine Institute, University of Alabama at Birmingham, USA
MARISSA ZHENG, Harvard University, USA
APOORV JAIN, Indian Institute of Technology Delhi, India
MICHAEL BALLANTYNE, Programming Research Laboratory, Northeastern University, USA
KATHERINE ZHANG, Harvard University, USA
MEI-JAN CHEN*, JORDAN WHITLOCK, MARY E. CRUMBLEY†, and JILLIAN TINGLIN‡, Hugh Kaul Precision Medicine Institute, University of Alabama at Birmingham, USA
KAIWEN HE, Department of Computer Science, University of Alabama at Birmingham, USA
YIZHOU ZHANG§, Harvard University, USA
JEREMY D. ZUCKER and JOSEPH A. COTTAM, Pacific Northwest National Laboratory, USA
NADA AMIN, Harvard University, USA
JOHN OSBORNE, Informatics Institute, University of Alabama at Birmingham, USA
ANDREW CROUSE and MATTHEW MIGHT, Hugh Kaul Precision Medicine Institute, University of Alabama at Birmingham, USA

mediKanren is a reasoning engine over biomedical knowledge, based on miniKanren. This paper gives a system description, along with examples of queries in low-level and medium-level query languages. We describe the implementation of mediKanren. We also illustrate the utility of mediKanren by describing how query results have resulted in treatment for a patient with a rare genetic disease. We also describe ways in which mediKanren might be improved or extended in the future. mediKanren is being developed as one of the reasoning engines for the NIH NCATS Biomedical Data Translator Program, and is compliant with standards and knowledge graphs produced by that program.

Additional Key Words and Phrases: relational programming, miniKanren, Racket, Scheme, relational programming, functional programming, precision medicine, drug repurposing, NCATS Biomedical Data Translator

## 1 INTRODUCTION

With over 1.5 million publications per year and more than 50 million total peer-reviewed articles, the rate and volume of novel discoveries has surpassed our ability to fully utilize and understand what is known [Jinha 2010]. This problem, described as the "unknown known," is characterized by two specific features: 1) "forgotten facts"—facts that are published but not widely known; and 2) "uninferred facts"—facts that have not been deduced

*now independent (current email: mjchen0098@gmail.com)
†now at University of Michigan Medical School (current email: mcrumble@med.umich.edu)
‡now at University of Alabama at Birmingham School of Medicine
§now at University of Waterloo (current email: yizhou.zhang@uwaterloo.ca)



*miniKanren 2020, August 27 2020, Online*

from existing published research. While all professional fields are subject to the effects of the "forgotten" or "uninferred" facts, the cost of the unknown known for healthcare providers is measured in human lives.

To tackle the issue of the uninferred medical knowledge, the University of Alabama at Birmingham's Hugh Kaul Precision Medicine Institute (UAB-HKPMI) has developed a software reasoning tool called *mediKanren* as a part of a multi-institutional grant funded by the National Center for Advancing Translational Science (NCATS). Since the tool's inception, mediKanren has been successful in finding novel FDA-approved therapeutic recommendations for disorders ranging from undiagnosed and purely symptomatic disease to genetically diagnosed metabolic disorders [Ross et al. 2019; Shepard 2019].

In this paper we give an overview of mediKanren, along with details of mediKanren's implementation and real-life example precision medicine queries. In Section 2 we provide background through an extensive glossary of all the main terms used in the paper. In Section 3 we give an example real-world scientific question, in which we want to change—or "budge"—the expression of a gene implicated in a disease. In Section 4 we answer this gene budging question using low-level and medium-level query languages, and using the mediKanren graphical user interface (GUI). In Section 5 we expand on the simple gene budging query to handle a set of genes implicated in Acute Respiratory Distress Syndrome (ARDS) in patients with COVID-19. In Section 6 we discuss how mediKanren performs concept normalization (sometimes called "synonymization") in order to find related concepts automatically, and to allow for cross-knowledge-graph queries. We also discuss tooling to automatically create maps of knowledge graphs. In Section 7 we discuss the core implementation of mediKanren. In Section 8 we discuss future directions. In Section 9 we conclude.

The example knowledge graphs and query answers in this paper are subject to change, as the knowledge graphs, mediKanren code, and underlying datamodel are improved.

## 2 BACKGROUND

mediKanren is being developed as part of the NIH NCATS Biomedical Data Translator Program. The following glossary defines common terms used in the NCATS Biomedical Data Translator Program, and in this paper. This glossary is designed to be read in order, and describes the essential background of the paper.

**NCATS Biomedical Data Translator Program** Multi-institutional program run by the National Center for Advancing Translational Sciences (NCATS), which is part of the National Institutes of Health (NIH). The program's goal is to combine structured knowledge with reasoning software that can be used by scientists/physician-scientists to accelerate the pace of *translational science*: translating basic scientific knowledge into treatments for patients.

**Translator System** The overall software system, structured knowledge, standards, and practices being developed by the NCATS Biomedical Data Translator Program.

**Translator Architecture** The architecture of the Translator System. The main components of the Translator Architecture are the autonomous relay service (ARS), autonomous reasoning agents (ARAs), and knowledge providers (KPs). The Translator Architecture also includes associated standards, including the Biolink Model, the Translator API (TRAPI), the KGX (Knowledge Graph eXchange) standard, and the *Smart API*.

**NCATS Biomedical Data Translator Consortium** A consortium of NIH NCATS-led research groups at multiple government institutions and universities working together to develop the Translator Architecture and System.

**precision medicine** Sometimes called *personalized medicine*. An approach to medicine in which treatment is tailored to each patient's genetic, environmental and lifestyle factors. For example, precision oncology might take into account specific mutations in the genome of the patient's tumors, along with the gene expression profiles in the cancer cells, using DNA and RNA sequencing. This information might be used to discover—or more likely, repurpose—drugs that might be especially effective for that patient.

**drug discovery** The process of finding a new chemical compound that treats one or more diseases. This process may involve laboratory bench work, computer simulations, large-scale robotic testing of thousands of compounds, or other expensive techniques.

**clinical trial** In the United States, a new drug must pass through multiple phases of clinical trials before being approved by the federal Food and Drug Administration (FDA). Bringing a newly discovered candidate drug to market is extremely expensive and usually takes many years, partly because many drugs are found to be unsafe in humans (fail the phase I of the clinical trial), or are found to be safe but ineffective in treating the target disease (fail phase II or phase III of the clinical trial).

**drug repurposing** The process of taking an existing drug that has been shown to be safe in humans (passed phase I of a clinical trial), and using that drug to treat a disease other than for which it was originally intended. A drug that passes phase I of a clinical trial, but fails phase II or phase III of that trial, might still be effective for treating some other disease that was not studied in the clinical trial. Since the drug has already passed phase I, and is considered safe, the time and cost to bring the drug to market can be greatly reduced. Furthermore, it may be possible for a physician to prescribe the drug to an individual patient even if the drug has not been shown to be effective at treating any specific disease, as long as there is evidence that the patient might benefit from the treatment, and that the risk of treatment is low.

**gene expression level** The amount of messenger RNA (mRNA) transcript produced for a given gene in a cell. This can be measured by techniques such as RNA sequencing, which can give an entire *gene expression profile*, showing which transcripts are being overexpressed or underexpressed in the cells of a patient, relative to a typical healthy cell. mRNA transcripts can be translated into proteins, which perform most of the functions within the cell.

**protein expression level** The amount of protein translated from a given gene in a cell. This can be measured by techniques such as proteomics, which can give an entire *protein expression profile*, showing which proteins are being overexpressed or underexpressed in the cells of a patient, relative to a typical healthy cell. Protein expression level is necessary, but not sufficient, for protein activity.

**protein activity** The rate at which a protein performs its function. Protein activity can be directly measured by bioassays, but these are, in general, not high throughput, so it is difficult to measure which proteins are active or inactive in the cells of a patient, relative to a typical healthy cell. The change in activity of proteins can cause disease symptoms (also known as *disease pheonotypes*), which can be treated in some cases through gene budging.

**gene budging** Using a drug to increase or decrease the activity of one or more proteins implicated in a disease or disease phenotypes, in order to try to treat the disease or its symptoms. Finding drugs that are safe in humans, and which can budge the activities of one or more target proteins in the desired direction (increasing or decreasing activity) is one approach to drug repurposing that we demonstrate in this paper, using mediKanren.

**Semantic Web** A set of related technologies, standards, and practices to develop interconnected knowledge sources that support automatic semantic reasoning. While originally proposed as an improvement to the World Wide Web, Semantic Web technologies and techniques can be used more generally. The NCATS Biomedical Data Translator Program has adopted many Semantic Web-related standards, such as CURIEs. The Semantic Web philosophy has influenced the development of new standards that are part of the Translator Architecture.

**CURIE** Compact Uniform Resource Identifiers (CURIEs) [W3C 2010] are machine-readable identifiers of the form `<knowledge-source>:<id>`. For example, `UMLS:C1425762` is a CURIE representing the human gene RHOBTB2. The prefix `UMLS` refers to the Unified Medical Language System (UMLS), which provides mappings between many different controlled vocabularies. The suffix `C1425762` is an identifier that is

unique within the UMLS knowledge source. CURIEs can be used to unambiguously refer to a concept or predicate, and to link concepts between different knowledge graphs.

**concept** An entity in the real world—for example, the drug *imatinib*, along with associated information about the drug. A concept may also contain metadata, such as provenance and context. A concept can be compactly represented as a *CURIE*.

**concept category** The type of a concept. For example, the concept *diabetes* might have the category *Disease*.

**predicate** A verb linking two concepts (the subject and the object). For example, the predicate *treats* might connect a *Drug* concept to a *Disease* concept.

**edge** A subject-predicate-object triple, where the subject and object are both concepts, along with associated data, representing an assertion—for example, the assertion *imatinib treats cancer*. An edge may also contain metadata, such as provenance and context.

**knowledge graph (KG)** A graph containing edges describing the relationship between concepts (nodes).

**map of a knowledge graph** A high-level graph showing only the concept categories, and predicates connecting concept categories, in a concrete knowledge graph. For example, if a concrete knowledge graph were to contain edges *imatinib treats cancer* and *insulin treats diabetes*, a map of that concrete knowledge graph might include the abstract edge *Chemical treats Disease*.

**node normalization** Also called *concept normalization*, *node synonymization*, or *concept synonymization*. The process of determining which distinct CURIEs actually represent the same concept in the real world.

**edge normalization** The process of determining which distinct subject-predicate-object edges actually represent the same assertion.

**controlled vocabulary** A curated set of terms describing concepts and/or predicates, usually from some specialized field, such as chemistry or medicine.

**ontology** A controlled vocabulary that also includes structured relationships between concepts, such as *is-a* (parent-child relationship) or *is-sibing-of* predicates.

***n*-hop query** A query between two concepts with precisely *n* unknown edges in the path between those concepts.

**Biolink Model** A standard datamodel for biomedical data used in the Translator System. The knowledge graphs used by mediKanren are in Biolink Model format.

**knowledge provider (KP)** A component of the overall Translator System that can provide structured knowledge to an autonomous reasoning agent (ARA), either dynamically in response to a API-based query, or as a bulk transfer of a knowledge graph via KGX (Knowledge Graph eXchange). Different KPs may specialize in different types of knowledge—for example, genomic information.

**autonomous reasoning agent (ARA)** A component of the overall Translator System that can perform reasoning over the structured knowledge provided by one or more knowledge providers (KPs) in response to a query. The mediKanren system described in this paper is one of several ARAs being developed as part of the Translator System. An ARA can be run in "stand-alone" mode, in which the ARA directly interacts with user queries, and directly provides responses to the user. For example, the GUI described in Section 4.3 allows a user to directly interact with mediKanren in stand-alone mode. In the context of the complete Translator System, a query would normally be sent to an ARA from the central autonomous relay service (ARS), or perhaps from another ARA, using Translator API (TRAPI) messages. Similarly, an ARA would respond to the query with a TRAPI-compliant message. Different ARAs may specialize in different types of reasoning, and may be implemented using any technology, as long as they abide by the standards of the Translator Architecture.

**autonomous relay service (ARS)** A component of the overall Translator System that acts as intermediary between an end user and the autonomous reasoning agents (ARAs). The ARS interprets a user query,

turning that query into Translator API (TRAPI) messages that are sent to one or more ARAs. Unlike the multiple ARAs and multiple knowledge providers (KPs) in the Translator System, there is a single ARS.

**mediKanren** One of several autonomous reasoning agents (ARAs) being developed as part of the NCATS Biomedical Data Translator System. mediKanren is the ARA described in this paper.

**Translator API (TRAPI)** An application programming interface (API) standard being developed as part of the NCATS Biomedical Data Translator Program. The TRAPI allows for the ARS, ARAs, and KGs to exchange information about queries.

**KGX (Knowledge Graph eXchange)** An NCATS Biomedical Data Translator Program standard—and set of tools—for bulk data exchange of knowledge graphs between KPs and ARAs.

Knowledge graphs (KGs) provide a unified way of organizing relations between concepts that exist in disparate locations and have been utilized in a variety of fields ranging from medicine to biodiversity and ecology [RDM 2019; Rotmensch et al. 2017].

As a part of a collaboration between UAB's Hugh Kaul Precision Medicine Institute and the NCATS Biomedical Data Translator Consortium, mediKanren make use of several knowledge graphs provided by other teams. Any knowledge graph compliant with the Translator KGX (Knowledge Graph eXchange) format can be readily imported into mediKanren. To give an example of a knowledge graph used by mediKanren, the Semantic MEDLINE Database (SemMedDB) [**?**] knowledge graph has more than 107 million assertions summing up more than 31 million publications (as of mid-2020). mediKanren also incorporates the *RTX*, *ROBOKOP*, and *Orange* knowledge graphs; the names of these knowledge graphs are derived from the team names in the early stage of the Translator project.

Figures 10 and 11 on pages 23 and 24 outline the mapped edges contained within the RTX and ROBOKOP KGs.

Knowledge graphs consist of medical concepts, represented as graph vertices, and relationships between them, represented as directed graph edges. Edges between concepts have a subject-predicate-object structure, as in the Resource Description Framework (RDF) data model. Both concepts and edges are attributed. Edge attributes include metadata about provenance and evidence supporting the relationship claim. For example, a triple of subject-predicate-object is "imatinib inhibits KIT gene" and the evidence would link to publications supporting that claim.

A subgraph containing further claims around "imatinib" is shown in Figure 12 on page 25. A bigger subgraph containing further claims around "imatinib" is shown in Figure 13 on page 26. In both cases, we are inferring that "imatinib treats asthma," a new finding based on known mechanisms.[1]

## 3 AN EXAMPLE SCIENTIFIC QUESTION: RHOBTB2 2-HOP QUERY

The question attempts to find a therapeutic option for UAB-HKPMI participants with rare mutations resulting in over-expression/gain-of-function of the RHOBTB2 gene. The clinical manifestations of RHOBTB2 over-expression include seizures, severe global developmental delay, movement disorders, ataxia, and generalized decrease in muscle tone. Thus, the query aims to find FDA-approved drugs `X` that directly (1-hop) or indirectly (2-hop, through some gene or protein `Y`) inhibit the over-production of the RHOBTB2 gene.

To find a safe drug, we go through an edge, `drug-safe`, that encapsulates whether a drug has a tradename. This is not a property of the drug, but is in the form of an edge/claim with the subject being the drug, the predicate being "has tradename," and the object being the trademark.

We will see how to answer this query in the next section, using low-level and medium-level languages.

[1] NCATS provided the example problem of trying to find mechanistic connections between imatinib and asthma during the initial Translator Feasibility Phase.

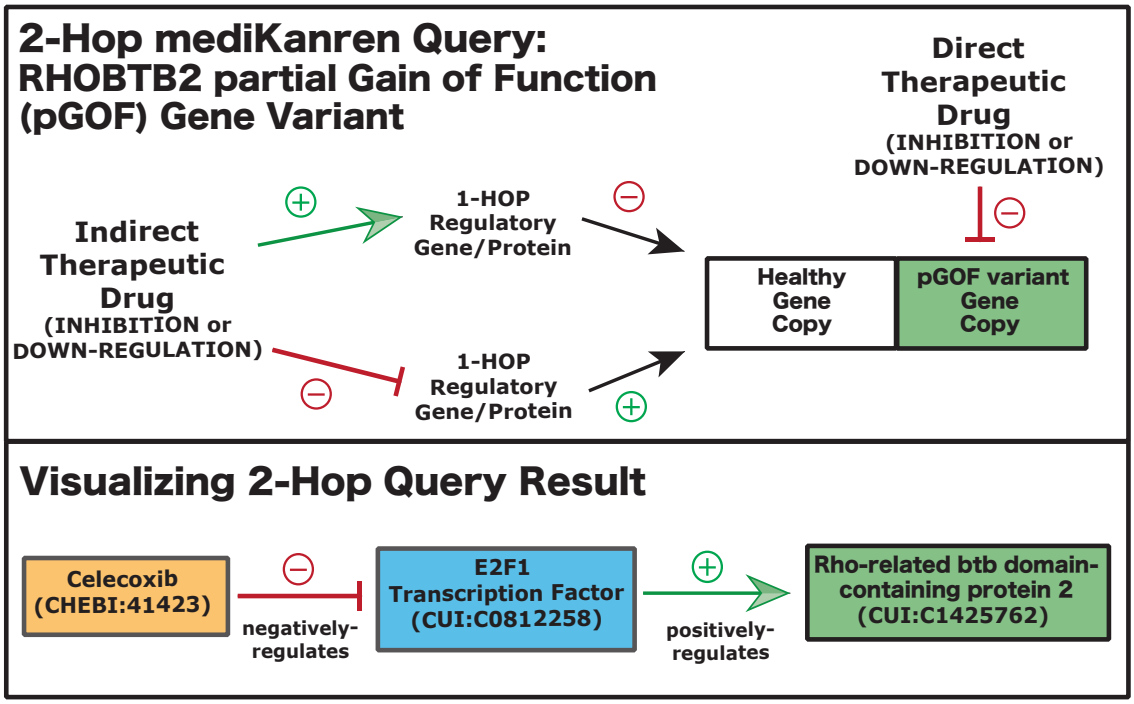


Fig. 1. The top panel of this diagram shows the therapeutic strategy for partial gain-of-function (pGOF) genetic variants, such as the RHOBTB2 case from UAB-HKPMI. The bottom panel serves as a visualization aid for the 2-hop mediKanren query result. The FDA-approved drug celecoxib was found to be an indirect inhibitor of the E2F1 transcription factor, which is responsible for expression of the RHOBTB2 gene. Concept CURIEs are notated in parentheses.

## 4 QUERYING KNOWLEDGE GRAPHS WITH MEDIKANREN

mediKanren currently supports two query languages: a low-level language that is essentially raw miniKanren, extended with relations to traverse knowledge graphs; and a more abstract interface, `query/graph`, that makes standard queries easier to write and to read.[2]

We assume the reader is familiar with the basics of miniKanren and Racket. For readers unfamiliar with miniKanren, the language is described in Friedman et al. [2018] and Byrd [2009]. A high-level overview of Racket is given by Felleisen et al. [2015].

### 4.1 Low-level Query Language (miniKanren)

Knowledge graphs can be queried directly in miniKanren, with the help of relations in mediKanren's miniKanren/graph database interface library. Graph database relations `~cui-concepto` and `edgeo` can be used to find

[2]mediKanren also supports `run/graph`, which is effectively a variant of `query/graph` without concept/edge normalization.

concepts (drugs, genes, diseases, etc.) and to find edges in a knowledge graph that unify against a given subject-predicate-object triple, for example.[3]

While edgeo is a pure relation, ~cui-concepto requires that its first argument be a ground string. This is typical of the design of mediKanren—while we try to preserve relationality when possible, in some cases it doesn't seem useful or scalable to make a relation operate in "all modes."

As a first step towards answering the scientific question of Section 3, we ask what are the concepts that positively or negatively regulate RHOBTB2. This is a 1-hop query. We will see later how to do 2-hop queries.

```
(define rhobtb2-curie "UMLS:C1425762")

(run* (q)
  (fresh (db edge eid subject object pred eprops
          scid scui sname sdetails
          pred-id pred-name
          ocid ocui oname odetails)
    (== q `(,db ,sname ,pred-name ,oname))
    (== edge `(,eid ,subject ,object ,pred . ,eprops))
    (== pred `(,pred-id . ,pred-name))
    (== subject `(,scid ,scui ,sname . ,sdetails))
    (== object `(,ocid ,ocui ,oname . ,odetails))
    (conde
     ((== pred-name "positively_regulates"))
     ((== pred-name "negatively_regulates")))
    (~cui-concepto rhobtb2-curie `(,db . ,object))
    (edgeo `(,db . ,edge))))
```

The query uses miniKanren's unification to pattern match on an edge, a predicate, a subject, and an object. The matching uses the underlying list representation of these entities. Because the object is known when the edge query edgeo is called, the interface to the database will use the "by object" index.

This query produces 12 answers:

```
'((uab-pmi "E2F1 gene" "positively_regulates" "RHOBTB2 gene")
  (uab-pmi "E2F1 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "Tumor Suppressor Genes" "negatively_regulates" "RHOBTB2 gene")
  (semmed "E2F1 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "E2F1 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "UBE2L3 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "muristerone" "positively_regulates" "RHOBTB2 gene")
  (semmed "ERVK-10 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "PRH2 gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "PR@ gene cluster" "negatively_regulates" "RHOBTB2 gene")
  (semmed "PGR gene" "negatively_regulates" "RHOBTB2 gene")
  (semmed "TMEM37 wt Allele" "negatively_regulates" "RHOBTB2 gene"))
```

This query is tedious and error-prone to write by hand—we will see a more abstract interface in the next section.

[3]The ~cui-concepto relation should really be named ~curie-concepto, since it is used to find concepts corresponding to a given CURIE (Compact Uniform Resource Identifier). In contrast to a CURIE, a *CUI* is a UMLS (Unified Medical Language System) concept identifier, which is used in SemMedDB. The name ~cui-concepto is a holdover from when mediKanren only used SemMedDB.

### 4.2 Medium-level Query Language

As can be seen in the code above, raw miniKanren is a low-level, and often clumsy, interface for expressing biomedical queries. To make it easier, faster, and less error-prone to write and read queries, mediKanren also supports a higher-level query interface, `query/graph`.

With this higher-level language, we can concisely express the two-hop query described in Section 3:

```
(query/graph
  (;; concepts
   (X       drug)
   (Y       gene-or-protein)
   (rhobtb2 "CUI:C1425762")
   (T       #f))
  (;; edges
   (X->Y       negatively-regulates)
   (Y->rhobtb2 positively-regulates)
   (X->T       drug-safe))
  ;; paths
  (X X->Y Y Y->rhobtb2 rhobtb2)
  (X X->T T))
```

The query asks for a safe drug that negatively regulates a gene or protein which in turn positively regulates the RHOBTB2 gene. The query is structured in three parts: declarations of concepts, edges, and paths. Concept and edge declarations include a name which will be used in path declarations, as well as a constraint. A path is comprised of concepts and edges, alternating between them.

Here we indicate that the `rhobtb2` concept set consists of a single element, identified by its CURIE. The concepts `X` and `Y` are constrained by sets of concept categories that abstract over the variation between category names in the various datasets (see Section 6.2). Similarly, we constrain each edge using a set of edge predicate names. The query uses two paths: one for the two-hops of downregulation and upregulation, and another for the drug-safe edge relating the drug to its trademark with FDA approval.

In general, a use of `query/graph` has the following structure:

```
(query/graph
  ((<concept-identifier> <curie-or-category>) ...)
  ((<predicate-identifier> <predicate> [<filter-procedure>]) ...)
  (<concept-identifier> <predicate-identifier> <concept-identifier>
   [<predicate-identifier> <concept-identifier>] ...) ...)
```

A `<curie-or-category>` must be a `<curie-string>`, or a list of `<category-string>`s, or `#f`, where `#f` stands for the full set of categories. A `<predicate>` must be a list of `<predicate-string>`s, or `#f`, where `#f` stands for the full set of predicates. A `<predicate>` may be followed by an optional procedure for filtering edges. The other substructures follow the definitions in Section 2.

The query above returns 11 results. Of the 11 "safe" drugs, 8 of them are scored with a "confidence" score of 0.925, including celecoxib. (This confidence score should not be taken too literally—this does not mean we have 92% confidence in the results! However, we can use relative confidence scores to help rank results.)

```
(0.925
  (("CHEBI:41423" . "CUI:C0812258") ("CUI:C0812258" . "CUI:C1425762"))
  (("celecoxib" . "E2f transcription factor 1")
   ("E2f transcription factor 1" . "Rho-related btb domain-containing protein 2")))
```

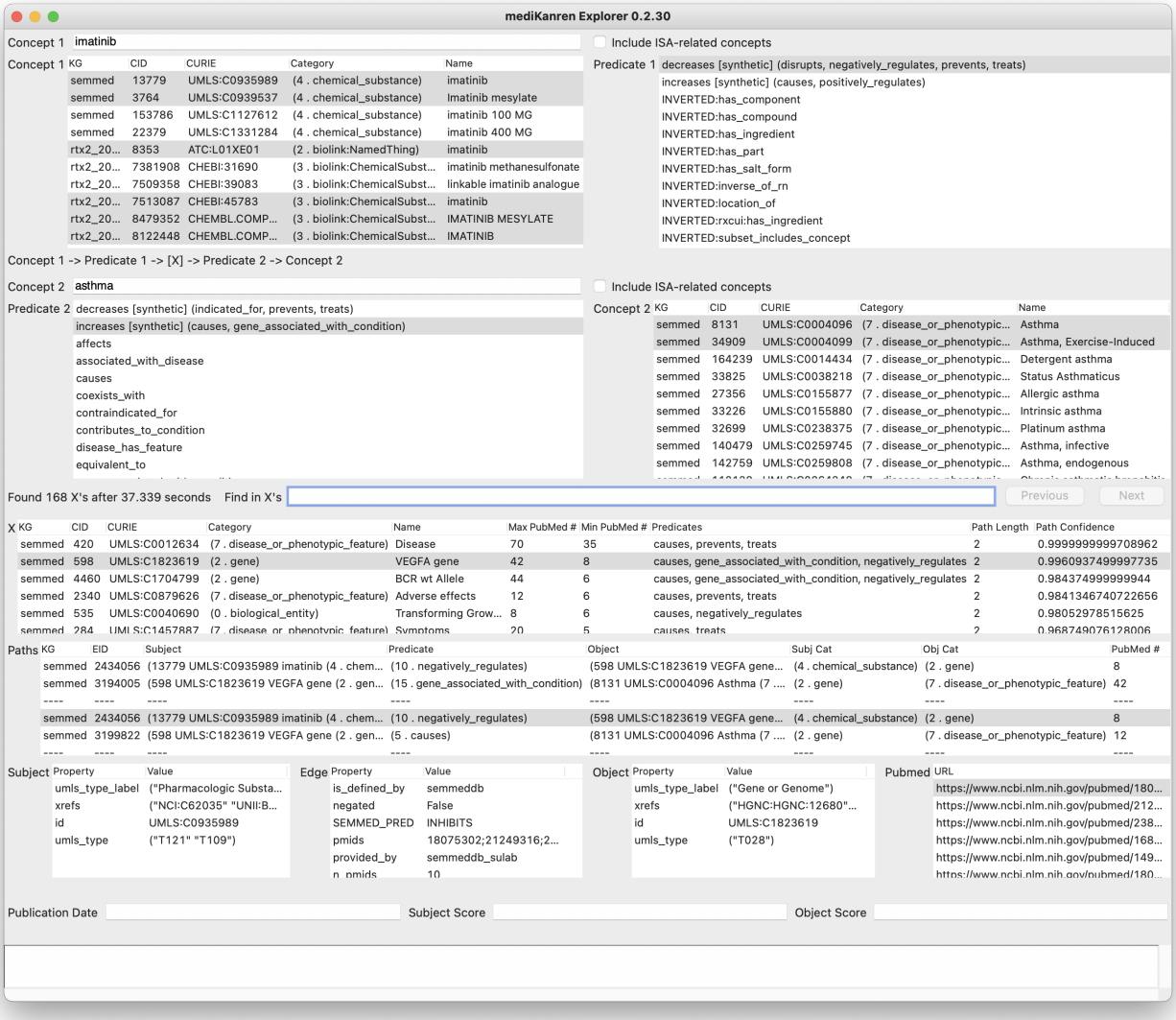


Fig. 2. mediKanren Graphical User Interface, showing a 2-hop query to determine if the cancer drug imatininb might treat asthma, through regulation of some unknown concept, "X." The user enters "imatinib" in the "Concept 1" text field, which populates the "Concept 1" list box with concepts containing the string "imatinib." The user selects several of these concepts, along with a set of predicates representing the notion of "Concept 1" "decreasing" the unknown concept "X." The user then enters "asthma" in the "Concept 2" text field, and selects two related concepts from the "Concept 2" list box, along with a set of predicates representing the notion of unknown concept "X" "increasing" "Concept 2." The list box "X" is then populated with 168 concepts that satisfy the query. The user selects a candidate "X" to inspect—in this case, the concept "VEGFA gene." The "Paths" list box is then populated with all the 2-hop paths that connect the selected imatinib-related concepts with the selected asthma-related concepts, through "VEGFA gene." The user selects the first edge in one of the 2-hop results: "imatinib negatively_regulates VEGFA gene." (The second edge in this 2-hop result is "VEGFA gene causes Asthma.") The boxes below the "Paths" list box include information about the edge, including clickable URLs for publications supporting that edge.

### 4.3 Graphical User Interface

As can be seen in Section 4, the query languages supported by mediKanren require knowledge of Racket and of relational programming. It is easy to make mistakes when hand-writing queries in these languages. Also, while expressive, these programmatic queries can take a while to write.

To support users who are non-programmers (or non-Racket/non-miniKanren programmers), we have created a graphical user interface (GUI) that is easy and fast to use. This GUI only supports a limited subset of mediKanren queries:

(1) 1-hop queries of the form *known subject concept S is related to unknown object concept O through known predicate P*;
(2) 1-hop queries of the form *unknown subject concept S is related to known object concept O through known predicate P*;
(3) and 2-hop queries of the form *known subject concept S1 is related to unknown concept X through predicate P1, and concept X is also related to object O2 through known predicate P2.*

Figure 2 is a screenshot of the mediKanren GUI, showing a simple 2-hop drug repurposing query provided by NCATS, which was described at the end of Section 2. This query is trying to determine if the cancer drug imatinib may also treat asthma through some unknown mechanism. Effectively, the query asks, "is there some concept $X$ such that imatinib reduces the severity of/inhibits $X$, and where $X$ causes/increases the severity of asthma?" One of the $X$'s in this case is the gene VEGFA. The user can click on one of the PubMed identifier links in the lower-right of the GUI window to open up that paper in a Web browser.

## 5 RANKING GENE BUDGING RESULTS FOR A SET OF GENES

A common use of mediKanren in the Hugh Kaul Precision Medicine institute is to find FDA-approved drugs that "budge" the expression of a gene, either increasing or decreasing the amount of protein produced by that gene. This task is referred to as *gene budging*.

### 5.1 Simple Gene Budging

Our running query from Section 3 is an example of simple gene budging: we are looking to regulate a gene via a drug.

As described in Figure 1, the FDA-approved compound celecoxib that was returned is an indirect inhibitor of RHOBTB2. Upon sharing this result with the participants' physician, celecoxib therapy was initiated by admission into clinical trials. As of January of 2020, the participants have reported a decreased number of ataxic events, with an increase in focus and attention span.

### 5.2 Ranking Gene Budging Results for Acute Respiratory Distress Syndrome (ARDS)

Gene budging finds drugs that upregulate or downregulate some gene. In practice, we might have a profile of genes that are over-expressed and under-expressed, and we want to find drugs that downregulate the over-expressed genes and up-regulate the under-expressed genes. Done naively, this process can result in an explosive cocktail of drugs.

Instead, it helps to know which genes have the most regulatory influence over the list of abnormally expressed genes. This is done by determining the number of abnormal genes that another gene, which may or may not be on the list of interest, influences. The genes that regulate the most other abnormal genes and fewest number of normal genes can be targeted to minimize the number of genes that need to be budged directly in order to return all abnormally expressed genes to normal levels while simultaneously minimizing the number of unintended adverse effects. Furthermore, this regulation should be done per cell tissue type to provide more precise anatomical context for targeting.

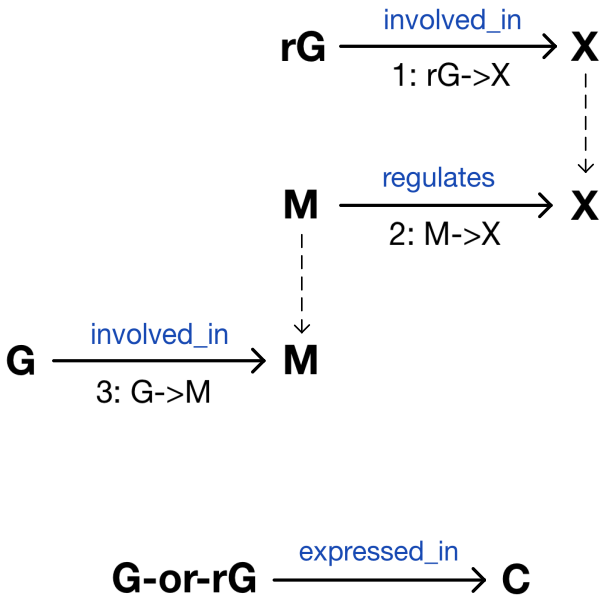


Fig. 3. Queries to find genes that regulate genes of interest. G are genes, M and X are pathways, rG are regulated genes (genes of interest), and C are cell/tissue types. Dotted arrows denote how one query feeds another. The flow starts at the top with the provided genes of interests rG. The first query out of the regulated genes rG finds the involved pathways X. The second query then finds the pathways M that regulate pathways X. Then the third query finds the genes G involved in pathways M. The indirection via pathways causes a loss of precision; however, the pathway information is very reliable, since it comes from the heavily curated Gene Ontology (GO). Finally, the independent bottom query finds the cell/tissue type C of each gene involved.

To implement ranked gene budging for a set of genes, we first use queries to extract the relevant data from the knowledge graphs. Subsequently we use Racket code to aggregate and sort information for each tissue type and regulating gene.

Figure 3 depicts the queries we use to extract the raw data. We start with a regulated gene *rG* of interest. We look at the pathways *X* involving the gene. We look at pathways *M* that regulate pathways *X*. We look at genes *G* that are involved in pathways *M*. We constrain per tissue type for each gene *G* or regulated gene *rG*.

The queries are as follow. We use `run/graph`, a variant of `query/graph` without concept normalization (see Section 6.2). We disable concept normalization to stay within the Gene Ontology (GO) [?], which contains curated information about genes, gene products, and pathways. We could have run all those queries in one big query, but it is more efficient to perform multiple smaller queries.

```
;; rG->X hop
(match-define
  (list rG/X=>concepts rG/X=>edges)
  (run/graph
    ((rG g) (X #f)) ;; g is a gene of interest
    ((rG->X involved_in))
    (rG rG->X X)))
```

```
;; M->X hop
(match-define
  (list M/X=>concepts M/X=>edges)
  (run/graph
    ((M #f)
     (X (hash-ref rG/X=>concepts 'X)))
    ((M->X (set-union positively_regulates negatively_regulates subclass_of)))
    (M M->X X)))
;; G->M hop
(match-define
  (list G/M=>concepts G/M=>edges)
  (run/graph
    ((G #f)
     (M Ms-in-GO))
    ((G->M involved_in))
    (G G->M M)))
;; query that finds cell expression information
(define q
  (query/graph
    ((G-or-rG g) ;; g is any regulator gene that was found by the previous query or
                 ;; any gene of interest in the given list
     (C #f))
   ((G-or-rG->C '("expressed_in")))
   (G-or-rG G-or-rG->C C)))
```

Loosely then, the genes *G* regulate the gene *rG*. For a particular gene *G*, we can do a more precise 1-hop query to confirm that it regulates gene *rG*, without the indirection of the pathways. Still, the multi-hop query through GO can find possible connections between genes and how they influence each other and/or a particular condition that may not have been studied before.

We process the query results using Racket code. For each tissue type, we create a map from each gene expressed in the tissue to a list of genes that the given gene regulates. If we sort the map by the number of genes regulated, this already gives us a good idea of the most regulating genes. Based on this map, we can rank gene budging results based on a number of criteria.

When we take the initial set of genes to be the genes involved in ARDS (Acute Respiratory Distress Syndrome), these ranked gene budging results has the potential to help COVID-19 patients, for whom ARDS can be fatal.

## 6 HARMONIZATION OF KNOWLEDGE GRAPHS

Multiple knowledge graphs (RTX, Orange, SemMedDB, ROBOKOP) each of which may reference overlapping source databases must be “harmonized” such that equivalent concepts and relations are identified. This yields a more computationally tractable graph and allows queries to traverse all accessible links that may have otherwise been missed due to source knowledge graph naming conventions. Currently mediKanren supports harmonization through the manual inspection of knowledge graph maps and the effective normalization of knowledge graph concepts and predicates. In the future, a complete harmonization will include meta-data such as the provenance of all input relations. For example, if an asserted gene-disease relationship is found in 2 input graphs (but based on the same publication) the harmonized knowledge graph should give such a relation a lower weight than if each knowledge graph derived its assertion from different publications.

### 6.1 Automatic Generation of Knowledge Graph Maps

To facilitate harmonization of input knowledge graphs, we assess the relevant concepts and relation classes of each input knowledge graph using maps. Initially, knowledge graph maps such as those in Figures 10 and 11 were hand drawn. While they look nice, they are time-consuming to create and they might be selective with respect to to the underlying knowledge graph triples.

For any knowledge graph, we can generate a map of triple types (subject prefix, predicate, object prefix), where a concept prefix takes into account the CURIE name up to the column. Now, we have a complete specification of what triple types are allowed by the knowledge graphs. Now, the question is how do we visualize the resulting map of triple types?

We are still looking into options, but we can compare a hand-drawn map of the Orange knowledge graph and a complete automatically generated map of the same knowledge graph in Figures 4 and 5. As we can see, the hand-drawn map is more readable; however, it is missing many concepts and edges. Inspection of these maps and the underlying knowledge graph data allows the creation of rules for concept and predicate expansion to effectively normalized graphs.

### 6.2 Normalization by CURIE and Predicate Expansion

Concept normalization traditionally involves the selection of a canonical representation of a concept or predicate to represent equivalent identifiers or synonyms for that concept. We achieve the same effect in mediKanren by collecting all known equivalent (but differently represented) CURIEs for a single concept by finding all identifiers for that concept across all publicly available databases. Comparing the two KGs, RTX and ROBOKOP, reveals a design similarity in using the HGNC CURIE to index gene concepts (see green highlighted CURIES in Figures 10 and 11); however, it also reveals critical differences. The ROBOKOP KG omitted cross-reference indexes to other commonly used gene indexes—NCBIGene, Ensembl, CUI, NCIT and MESH (Figure 11). Moreover, the ROBOKOP KG chose the PANTHER-FAMILY CURIEs to index protein concepts instead of UniProtKB. We effectively normalize all concepts in mediKanren's knowledge graphs by adding these omitted cross references.

Beyond concept CURIE differences, the graphs differ significantly with respect to predicate designations. ROBOKOP's `part_of` predicate (as opposed to RTX's encodes predicate) is used to traverse between a specific gene and its respective coding gene/protein family. Similarly with concepts, these equivalent predicates are added to the underlying knowledge graph. These differences may at first glance appear trivial, but the underlying differences in these graphs make cross-knowledge graph queries inoperable. We discuss further how concept and predicate normalization are currently performed in the implementation section (Section 7.1).

## 7 IMPLEMENTATION

Rather than build on an existing relational or graph database, we chose to implement our system using miniKanren and Racket, which are both flexible and expressive languages. We made this choice to reduce the risk of painting ourselves into a corner as we discovered requirements during development, as we were more confident in our ability to adapt miniKanren and Racket as necessary. In particular, the faster-miniKanren[4] implementation is fairly small and simple, making it quite hackable.

We use Racket to implement a simple graph database using a mixture of files containing either plain text s-expressions or data in a custom binary format. These files represent the underlying data, as well as indices supporting fast lookup needed for common queries over the graph structure. Aside from indices for graph structure lookup, we also include indices for full-text search over concept names, supporting responsive lookup in our GUI.

[4]https://github.com/michaelballantyne/faster-miniKanren

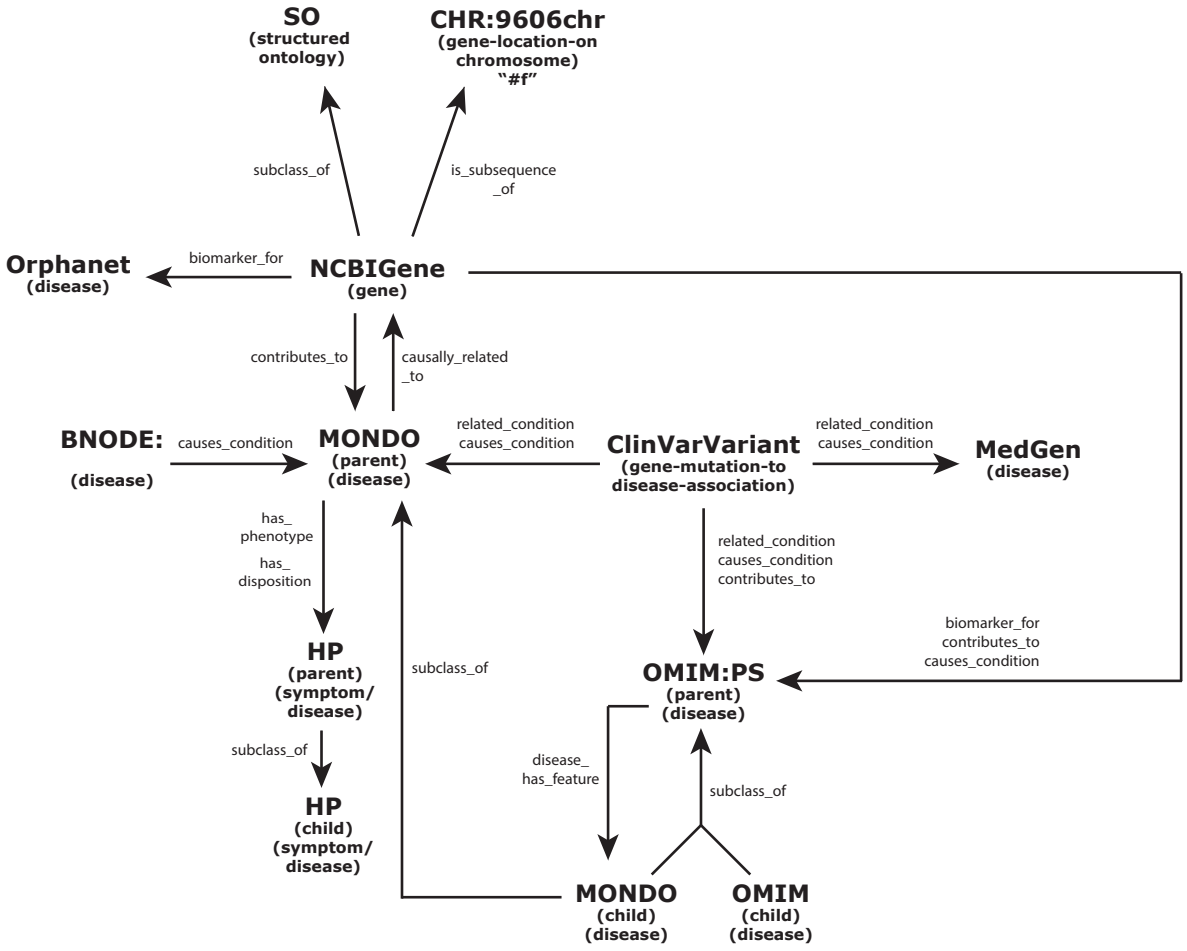


Fig. 4. Hand-drawn map of Orange knowledge graph.

The datasets we deal with, while many GB in size, are still small enough that an expensive computing cluster is unnecessary. In fact, mediKanren has been designed to run on fairly modest laptops to allow scientists to use their personal machines to explore the data without needing internet access or other supports. For this reason, our custom database implementation performs most data retrievals directly from disk without prefetching into RAM. This choice reduces both memory usage and startup time. Despite this tradeoff, mediKanren is still fast enough for interactive use during case reviews or other situations where it's necessary to perform many simple queries quickly. The simplicity of this database implementation has allowed us to quickly prototype new features, but trades off some performance and compact data encoding.

To improve interoperability, the knowledge graphs we work with have had their structure standardized according to the Biolink Model [Team 2020b]. This standardization allows us to ingest and process new data from different sources using the same pipeline. This pipeline begins by converting a data source to CSV (or TSV) format using the Knowledge Graph eXchange (KGX) tool [Team 2020a], which was developed by a collaborative

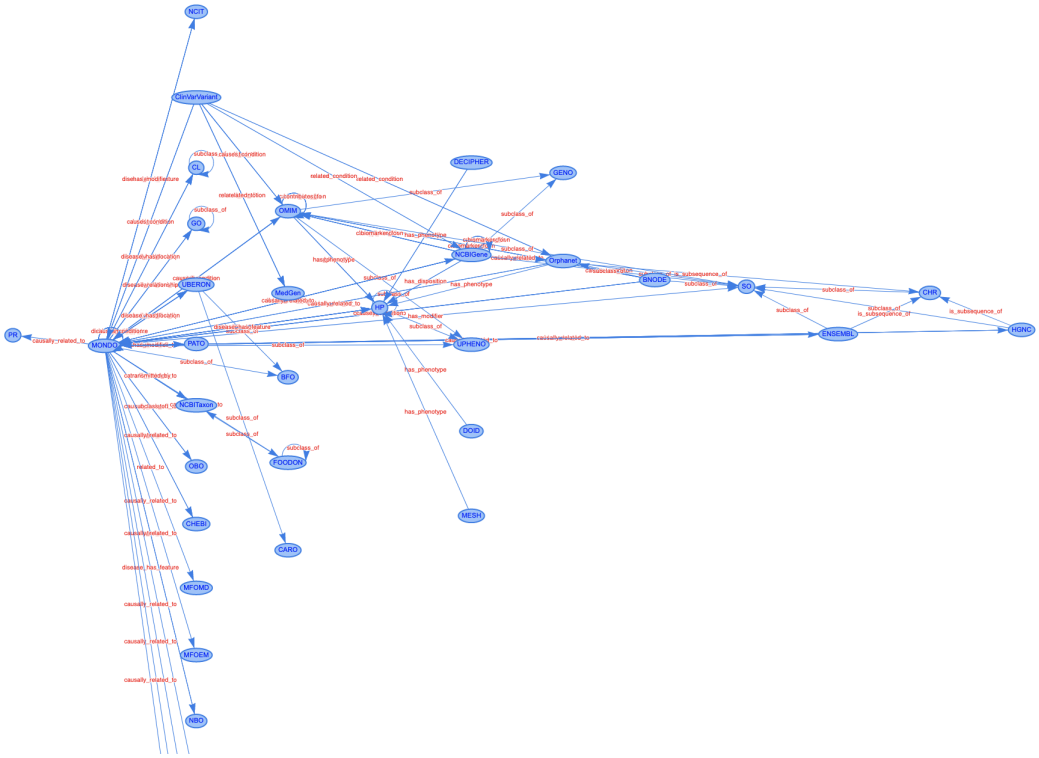


Fig. 5. Autogenerated map of the Orange knowledge graph. When compared with the hand-drawn map in Figure 4, it is obvious that we need to improve the layout of the map, and also need better ways to navigate the map. In practice, it is more useful at the moment to look at the text table summarizing the knowledge graph by type, from which the map is generated.

effort for general use. Given the resulting CSVs (or TSVs), we then run a second processing step specific to our implementation, to convert this input data to our database representation.

We expose concept and edge information to a miniKanren program by defining the relations `concepto` and `edgeo`. These relations are defined using miniKanren's `project` construct, which allows us to implement them with Racket code that accesses our database representation, choosing an appropriate index to use based on groundness of different portions of the concept or edge argument.

## 7.1 Concept Normalization Implementation

As described in Section 6.2, we want to be able to write queries that connect data across knowledge graphs. Unfortunately, we cannot do this naively because many of the concepts we work with are given different CURIEs in each knowledge graph. Rather than normalizing each input graph to a canonical CURIE for each entity type (for example, using *only* HGNC CURIEs for all genes) we normalize concepts to equivalence classes. Each concept is assigned an equivalence class by computing the set of identifiers reachable from a concrete concept either through special properties or by following special edges (such as edges containing the Biolink predicate `same_as`), guided by manual inspection of the underlying source knowledge graph or generated knowledge graph map. These special properties and edges are defined using easy-to-change rules that allow us to experiment with different

notions of equivalence. Because concept normalization is an expensive operation, we support pre-building a cache of each concept's equivalence class.

### 7.2 Propagators

The `query/graph` DSL is implemented on top of a constraint propagation architecture inspired by Radul and Sussman [2009]. Concept and edge sets specified in a query are represented as cells containing sets of intermediate results coming from underlying miniKanren queries. These cells are connected by propagators that define constraints on those cells. Whenever a cell's constraint is updated, the set of results the cell contains may shrink. Anytime a set shrinks, any propagators watching that set's cell will be scheduled to update its constraints. This update process continues until quiescence, by which point all constraints will be arc-consistent. Final results are then extracted from cells by finding all globally consistent combinations.

## 8 FUTURE WORK

### 8.1 dbKanren

We have started a new implementation of our database with a more compact data representation, higher performance, and a more direct integration with miniKanren, including the ability to precompute and persist finite relations. The new implementation will also support stratified aggregation and computing fixed points, in the sense of Datalog, allowing us to reduce the amount of ad hoc Racket code we have to write when expressing complex queries. Data ingestion and processing will be possible to express relationally, rather than requiring ad hoc Racket code. At some point, we would also like to support temporally stratified relations, allowing us to describe dynamic systems, such as the UI layer, relationally as well, reducing the effort needed to prototype new UIs.

### 8.2 Ontologies

We aim to improve on intelligent uses of ontology data sources. Ontologies have been used extensively in biomedical and translational science to organize and structure knowledge with child-parent relationships (for example, symptoms comprising a disease) and/or functional class information (for example, a drug or gene classified by its mechanism of action or function) [Denny et al. 2018; Martínez-Romero et al. 2017]. A key issue in gathering information from ontologies are the inherent asymmetries that exists between data sources. Figure 6 outlines the structural differences in the Medical Subject Headings (MESH) and the Chemicals Entities of Biological Interest (CHEBI) ontologies [de Matos et al. 2010] with respect categorizing anticoagulant drug concepts. To date, mapping concepts across ontologies remains an ongoing struggle and area of active research [Groß et al. 2016]. In the future, we hope to develop a systematic way of identifying and reconciling differences across ontologies during import without losing data integrity.

### 8.3 High-Level Query DSL

We are creating a more intuitive query language based on feedback from biomedical researchers. Our existing query DSL designs are not ideal for biomedical researchers without experience in Racket or logic programming. Based on such users' feedback, we are improving the `query/graph` interface in several ways. We have designed a new syntax reminiscent of SQL, with a "select" clause for accessing node and edge attributes and a "where" clause for filtering results based on these attributes. Concepts and edges are specified in named clauses to make the syntax more readable to novices. The output of a query is structured as a list of rows for easy import to Excel for further analysis. The new DSL also checks syntax more carefully to provide informative error messages. We expect this DSL to elaborate to the next-generation mediKanren logic language. Experts will be able to mix

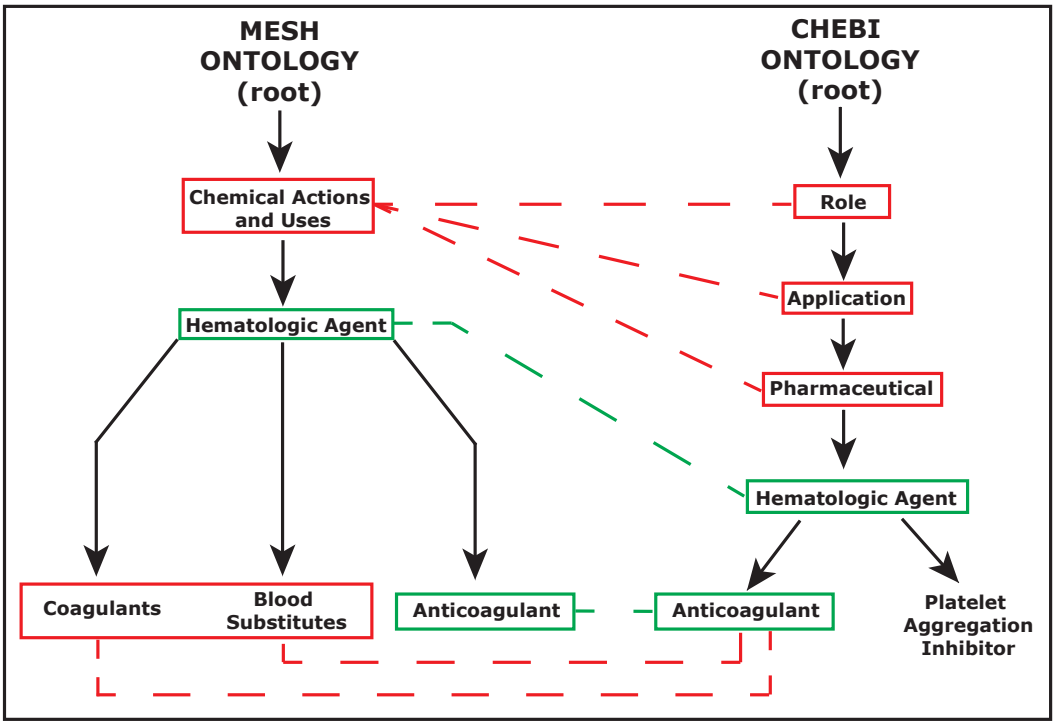


Fig. 6. MESH and CHEBI chemical ontologies display some similarities (dashed green) but many more structural differences (dashed red) for organization of drug class architecture.

high-level queries with lower-level relational code to express complex queries. Figure 7 shows how we imagine the query from Section 4.2 will be expressed.

Using the query language, we can search for over-represented biological motifs in the knowledge graph:

- confounders($A$,$B$) = $A \longleftarrow ?X \longrightarrow B$
- colliders($A$,$B$) = $A \longrightarrow ?X \longleftarrow B$
- feedback($A$) = $A \longrightarrow ?X \longrightarrow A$
- feedForward($A$,$B$) = $A \xrightarrow{v_1} ?X \xrightarrow{v_2} B, A \xrightarrow{v_3} B$

and enable some ranking over the query results, for example with a belief maintenance system introduced next.

```
(query/graph
  (select (curie X) (curie Y) (pubmed X->Y) (pubmed Y->rhobtb2))
  (concepts
    (X       drug)
    (Y       gene-or-protein)
    (rhobtb2 "CUI:C1425762"))
  (edges
    (X negatively-regulates Y       #:as X->Y)
    (Y positively-regulates rhobtb2 #:as Y->rhobtb2))
  (where (safe-drug X)))
```

Fig. 7. RHOBTB2 query using our future query language design. Sections 4.1 and 4.2 show the same query expressed using mediKanren's existing low-level and medium-level query languages, respectively.

### 8.4 Truth Maintenance Systems and Causal Reasoning

We are looking at non-monotonic reasoning based on truth maintenance systems [Doyle 1979; Forbus and de Kleer 1993] and belief maintenance systems [Falkenhainer 1988; Ramoni and Riva 1993].

*Truth maintenance systems* (TMS) have been incorporated in mediKanren for modeling the knowledge base as a directed graph with possible benefits of introducing the "beliefs" for an edge $A \rightarrow B$ (where $A$ and $B$ are concepts) in terms of the antecedents. These beliefs are essentially the support for and against a given assertion (for example, "imatinib cures cancer") and are incorporated by adding a layer of *belief maintenance* (BMS) on top of the TMS. The TMS, with its ability to identify the conclusions of a set of antecedents, helps in easily updating the beliefs. This has applications in drug ranking. For example, a query such as $A \rightarrow_{\text{increases}} X \rightarrow_{\text{cures}} B$ ($A$ increases some intermediary $X$ which cures $B$) could be ranked by the BMS in decreasing order of support, which would thus help identify the $X$s having the greatest support for the assertion above.

In addition, we have extended truth maintenance systems to support deterministic and probabilistic causal reasoning so that we can leverage knowledge of regulatory mechanisms to draw inferences about causal effects [Pearl 2009].

To illustrate the kinds of reasoning that this extension permits, let's examine an example from biochemistry.Consider the following diamond motif in Figure 8, which was shown to be over-represented in mammalian signal transduction pathways [Ma'ayan et al. 2005]. In this motif, $U$ is a stimulus, such as a signaling ligand, and $C$ is a receptor. $A$ and $B$ are downstream signaling molecules, both of which are activated in response to $U$ binding to $C$, and $D$ is a transcription factor that can be activated by the activity of either $A$ or $B$.

In this case, we would like to perform the following kinds of deterministic causal reasoning:

- **Prediction** ($\neg A \Longrightarrow \neg D$): If signaling molecule $A$ is not active, then transcription factor $D$ is not active.
- **Abduction** ($\neg D \Longrightarrow \neg C$): If transcription factor $D$ is not active, then the receptor $C$ is not bound to the ligand.
- **Transduction** ($A \Longrightarrow B$): If signaling molecule $A$ is active, then so is signaling molecule $B$.
- **Action** ($\neg C \Longrightarrow D_A \wedge \neg B_A$): If the receptor $C$ is not bound, but we intervene to activate $A$ with a drug, signaling molecule $B$ will not be active but the transcription factor $D$ will still be activated.
- **Counterfactual** ($D \Longrightarrow D_{\neg A}$): If transcription factor $D$ is active, then it will still be active if we intervene to inhibit $A$ with a drug.

We have further extended the TMS to handle probabilistic causal reasoning. For example suppose $U$ can bind to $C$ with probability $p$ and $A$ can spontaneously activate with probability $q$. We now wish to know $P(\neg D_{\neg A}|D)$,

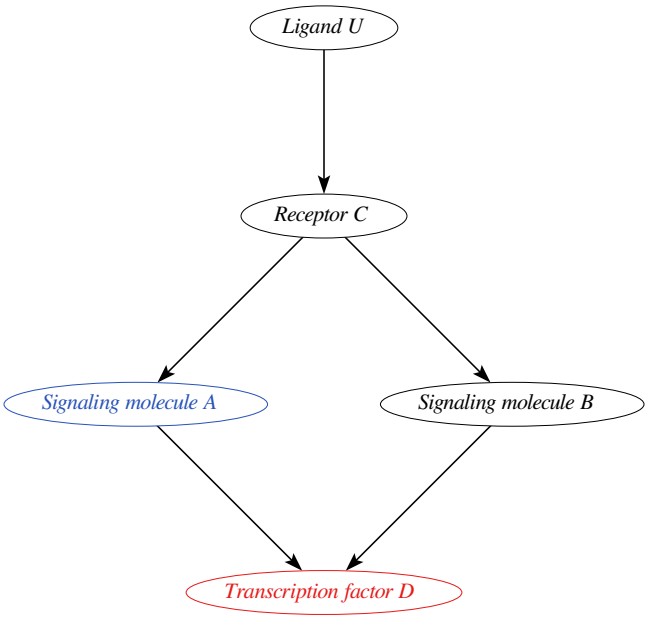


Fig. 8. Causal diagram for diamond signaling motif.

that is, the probability that if transcription factor $D$ was observed to be active, it would not be active if we were to inhibit $A$.

As a further extension of the TMS, we would like to reason about cases where there may be unmeasured common causes of the variables in our model.

For example, consider the case in Figure 9 where we wish to predict the causal effect of some observable phenotype, such as cholesterol serum levels, on a disease such as cancer. Unfortunately, naive attempts to predict this causal effect may be biased because there are many common causes of both cholesterol levels and risk for cancer, such as diet, that could introduce confounding. However, if it is also known that certain alleles of a gene, such as apoE, are genetic determinants of low serum levels, then this knowledge can be used to predict an unbiased causal effect of cholesterol on cancer using a methodology called *Mendelian randomization* [Katan 2004].

To generalize this reasoning capability, we implemented sound and complete algorithms in the TMS that can identify which causal effects can be identified given a causal knowledge graph containing a set of observed and latent nodes [Shpitser and Pearl 2008]. In those cases where the causal effect is identified, the TMS will derive a formula for estimating the unbiased causal effect from observational data.

## 8.5 A Methodology for Improving COVID-19 Clinical Outcomes

COVID-19 has a complex etiology that starts in the lungs, and—through cytokine-induced inflammation and virus-induced endothelial tissue damage—activates the coagulation cascade, leading to thrombotic events [Voutouri

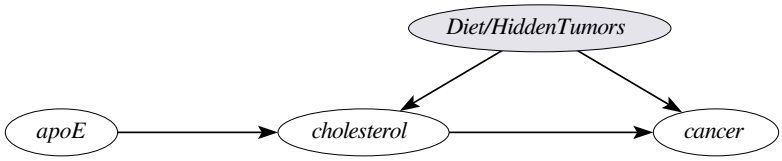


Fig. 9. A causal model with unobserved confounders (grey). Mendelian Randomization is one method for using prior knowledge networks to predict causal effects in the presence of unobserved confounders.

et al. 2021]. Doctors calibrate an implicit model of the COVID-19 disease pathway using clinical lab values to select medical countermeasures that guide the patient back to health. Clinical data from hundreds of thousands of de-identified patients has been collected and recently made available through NCATS National COVID Cohort Collaborative (N3C) [n3c [n.d.]]. We are currently using causal knowledge extracted from mediKanren to connect these clinical lab values to an explicit model of the COVID-19 disease pathway to account for potential sources of bias in the clinical data. By applying causal reinforcement learning to the N3C dataset, we aim to learn optimal dynamic treatment regimes that result in improved clinical outcomes [Zhang and Bareinboim 2020].

## 9 CONCLUSION

In this paper we introduced *mediKanren*, a combination of miniKanren, a graph database, knowledge graphs describing relationships between medical concepts, multiple query languages of varying degrees of abstraction, and a graphical user interface (GUI) to simplify data exploration and common queries. All features of the faster-miniKanren implementation are available for queries, including typical constructs like `run`, `conde`, `fresh`, various constraints, and (potentially recursive) user-defined relations. We provide the database as a set of miniKanren relations. To make queries fast, we represent the data backing these relations as specially formatted files on disk, with indexes for fast retrieval. We use the miniKanren `project` syntax to write Racket code that interfaces with this representation. Performance is sufficient for the GUI to support low-latency querying in the common case.

## ACKNOWLEDGMENTS

Support for this work was provided by the National Center for Advancing Translational Sciences, National Institutes of Health, through the Biomedical Data Translator Program, awards OT2TR003435 and OT2TR002517. Any opinions expressed in this document are those of the Translator community at large and do not necessarily reflect the views of NCATS, individual Translator team members, or affiliated organizations and institutions.

Support for Marissa Zheng and Katherine Zhang was provided by the Harvard College Research Program (HCRP) through the Harvard College Office of Undergraduate Research and Fellowships.

Support for Jeremy Zucker and Joseph Cottam was provided by the PNNL Laboratory-directed R&D Data-Model Convergence Initiative. PNNL is operated for the DOE by Battelle Memorial Institute under Contract DE-AC05-76RLO1830.

We thank Jim Cimino and Jake Chen of UAB's Informatics Institute for their ideas and suggestions during the Translator Feasibility Phase.

We also thank Chris Austin, Christine Colvis, Noel Southall, and the rest of the NCATS for their vision and leadership of the Biomedical Data Translator Program. We also thank the other Translator teams for their support

and friendly working relationships. We have benefited especially from the advice of: Andrew Su at Scripps; Maureen Hoatlin and Matt Brush at OSHU; Chris Mungall at LBNL; Steve Ramsey at Oregon State University; Chris Bizon at RECNI/UNC, Melissa Haendel at the University of Oregon; and Paul Clemons at the Broad Institute. We are also grateful to all of the teams that have provided us knowledge graphs, beginning with Andrew Su and Greg Stupp, who supplied the initial Biolink-compatible versions of SemMedDB. The ROBOKOP knowledge graph from the ROBOKOP team at RENCI has also been invaluable. Steve Ramsey and his team at Oregon State University have created and refined multiple versions of their RTX 2 knowledge graph specifically for our needs.

Will Byrd, Greg Rosenblatt, and Michael Patton greatly benefited from visiting other Translator teams; we thank everyone for their hospitality, especially: Maureen Hoatlin, Melissa Haendel, and Matt Brush for our OSHU visit; Chris Mungall, Marcin Joachimiak, Deepak Unni, and Seth Carbon for our LBNL visit; and Paul Clemons, Jason Flannick, Vlado Dancik, Mathias Wawer, and Marcin von Grotthuss for our visits to the Broad Institute. We also thank Paul Clemons, Steve Ramsey, Melissa Haendel, Chris Bizon, and Eugene Muratov for visiting us at UAB. We thank the teams and institutions that hosted in-person Translator hackathons for their hospitality, as well: Scripps (La Jolla), RENCI (Chapel Hill), OSHU (Portland), NCATS (Bethesda), and ISB (Seattle).

We thank George Bonheyo for helpful edits. We also thank the anonymous reviews from the miniKanren Workshop for their helpful suggestions.

Nada Amin thanks Elena Glassman and Grzegorz Kossakowski for brainstorming.

Nada Amin and Will Byrd thank Gerald Jay Sussman at MIT for validating that mediKanren is a fine application for exploring and pushing the boundaries of truth maintenance systems.

We thank everyone who has used mediKanren and given us feedback over the past 3 years. The Biomedical Data Translator Programs's subject-matter experts have been especially helpful, especially Maureen Hoatlin from OHSU. The online mini-hackathons arranged by Karamarie Fecho from RENCI also helped us understand the strengths and limitations of our tools and knowledge sources.

## REFERENCES

[n.d.]. About the National COVID Cohort Collaborative |National Center for Advancing Translational Sciences. https://ncats.nih.gov/n3c/about

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

P de Matos, R Alcántara, and A Dekker et al. 2010. Chemical Entities of Biological Interest: An update. *Nucleic Acids Res* (2010), D249-D254. Issue 38. doi:10.1093/nar/gkp886.

P Denny, M Feuermann, DP Hill, RC Lovering, H Plun-Favreau, and P Roncaglia. 2018. Exploring autophagy with Gene Ontology. *Journal Biomedical Semantics* 14 (2018), 419–436. doi:10.1080/15548627.2017.1415189.

Jon Doyle. 1979. *A Truth Maintenance System.* Technical Report. MIT AI Memo 521.

Brian Falkenhainer. 1988. Towards a General-Purpose Belief Maintenance System. In *Uncertainty in Artificial Intelligence*, John F. LEMMER and Laveen N. KANAL (Eds.). Machine Intelligence and Pattern Recognition, Vol. 5. North-Holland, 125 – 131. https://doi.org/10.1016/B978-0-444-70396-5.50017-0

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

Kenneth D. Forbus and Johan de Kleer. 1993. *Building Problem Solvers.* MIT Press, Cambridge, MA, USA.

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (2nd ed.). The MIT Press, Cambridge, MA, USA.

A Groß, C Pruski, and E Rahm. 2016. Evolution of biomedical ontologies and mappings: Overview of recent approaches. *Comput Struct Biotechnol J.* 14 (2016), 333–340. doi:10.1016/j.csbj.2016.08.002.

Arif E. Jinha. 2010. Article 50 million: an estimate of the number of scholarly articles in existence. 23 (2010), 258–263. doi:10.1087/20100308.

M B Katan. 2004. Apolipoprotein E isoforms, serum cholesterol, and cancer. 1986. *International Journal of Epidemiology* 33, 1 (feb 2004), 9. https://doi.org/10.1093/ije/dyh312

Avi Ma'ayan, Sherry L Jenkins, Susana Neves, Anthony Hasseldine, Elizabeth Grace, Benjamin Dubin-Thaler, Narat J Eungdamrong, Gehzi Weng, Prahlad T Ram, J Jeremy Rice, Aaron Kershenbaum, Gustavo A Stolovitzky, Robert D Blitzer, and Ravi Iyengar. 2005.

Formation of regulatory patterns during signal propagation in a Mammalian cellular network. *Science* 309, 5737 (12 aug 2005), 1078–1083. https://doi.org/10.1126/science.1108876

Marcos Martínez-Romero, Clement Jonquet, Martin J. O'Connor, John Graybeal, Alejandro Pazos, and Mark A. Musen. 2017. *Journal Biomedical Semantics* 8 (2017), 21. doi: 10.1186/s13326-017-0128-y PMCID: PMC5463318.

Judea Pearl. 2009. *Causality: Models, Reasoning and Inference* (2nd ed.). Cambridge University Press, USA.

Alexey Radul and Gerald Jay Sussman. 2009. The art of the propagator. In *Proceedings of the 2009 International Lisp Conference*. 1–10.

Marco Ramoni and Alberto Riva. 1993. Belief maintenance with probabilistic logic. In *Proceedings of the AAAI Fall Symposium on Automated Deduction in Non Standard Logics, Raleigh, NC*.

Page RDM. 2019. Ozymandias: a biodiversity knowledge graph. *Scientific Reports* 7 (2019), e6739. doi:10.7717/peerj.6739.

C Ross, H Empinado, and R Robbins. 2019. An AI expert's toughest project: writing code to save his son's life - STAT. https://www.statnews.com/2019/07/25/ai-expert-writing-code-save-son/.

M Rotmensch, Y Halpern, A Tlimat, S Horng, and D Sontag. 2017. Learning a Health Knowledge Graph from Electronic Medical Records. *Scientific Reports* 7, 1 (2017), 5994. doi:10.1038/s41598-017-05778-z.

B. Shepard. 2019. Diagnosis in 2.127 seconds: Solving a years-long vomiting mystery using AI, research and brain power - News. https://www.uab.edu/news/health/item/10703-diagnosis-in-2-127-seconds-solving-a-years-long-vomiting-mystery-using-ai-research-and-brain-power.

Ilya Shpitser and Judea Pearl. 2008. Complete Identification Methods for the Causal Hierarchy. *Journal of Machine Learning Research* 9, 64 (2008), 1941–1979. http://jmlr.org/papers/v9/shpitser08a.html

Biomedical Data Translator Tangerine Team. 2020a. Knowledge Graph Exchange tools for Biolink Model compliant graphs. https://github.com/NCATS-Tangerine/kgx.

Biolink Model Team. 2020b. Biolink Model. https://biolink.github.io/biolink-model/.

Chrysovalantis Voutouri, Mohammad Reza Nikmaneshi, C Corey Hardin, Ankit B Patel, Ashish Verma, Melin J Khandekar, Sayon Dutta, Triantafyllos Stylianopoulos, Lance L Munn, and Rakesh K Jain. 2021. In silico dynamics of COVID-19 phenotypes for optimizing clinical management. *Proceedings of the National Academy of Sciences of the United States of America* 118, 3 (19 jan 2021). https://doi.org/10.1073/pnas.2021642118

W3C. 2010. CURIE Syntax 1.0: A syntax for expressing Compact URIs. https://www.w3.org/TR/curie/.

Junzhe Zhang and Elias Bareinboim. 2020. Designing Optimal Dynamic Treatment Regimes: A Causal Reinforcement Learning Approach. PMLR. http://proceedings.mlr.press/v119/zhang20a.html

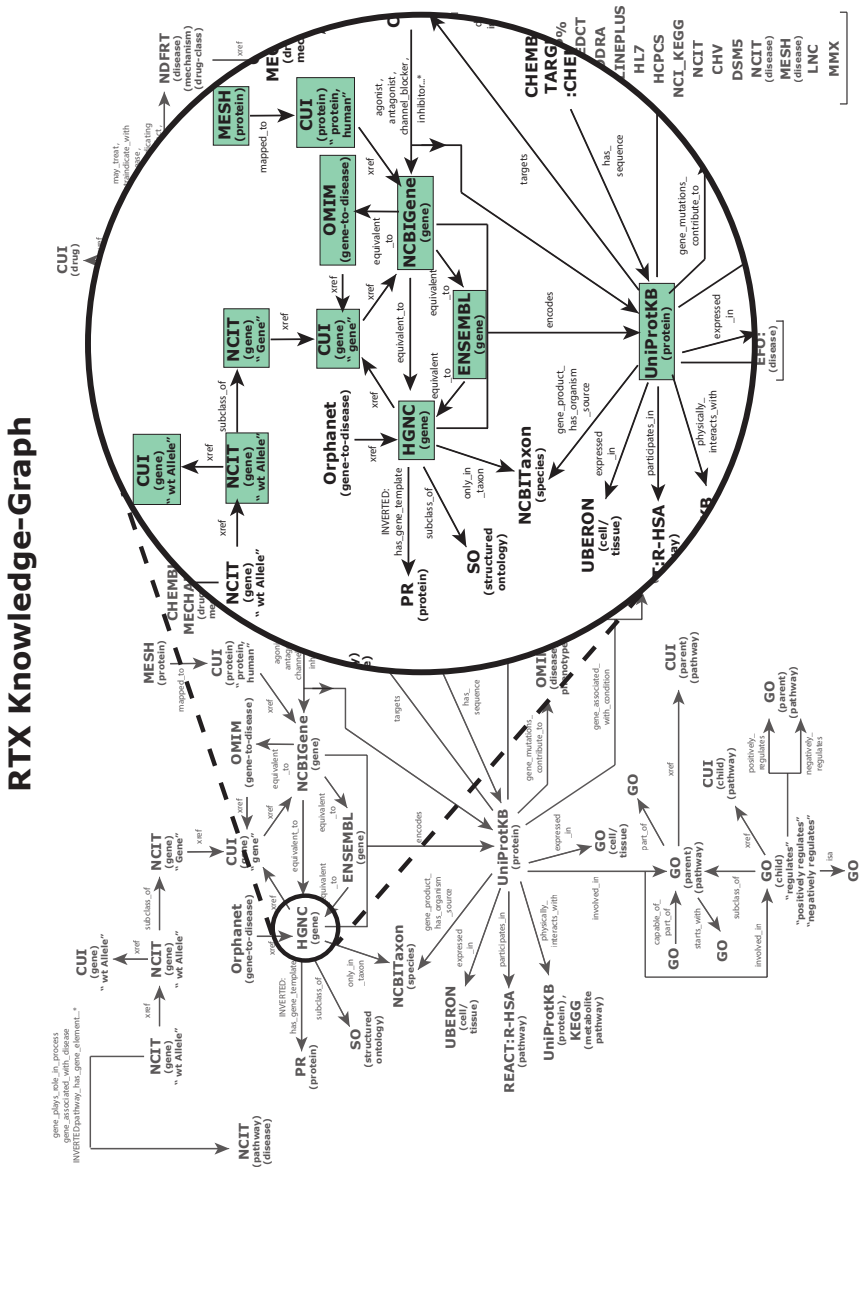


Fig. 10. Zoomed-in and green highlighted concepts represent gene and protein concept specific CURIES, and the predicate paths that connect them. Concept types are noted in parenthesis below each database name (bold). Edge predicates are located above each arrow, directing the relation between concepts.

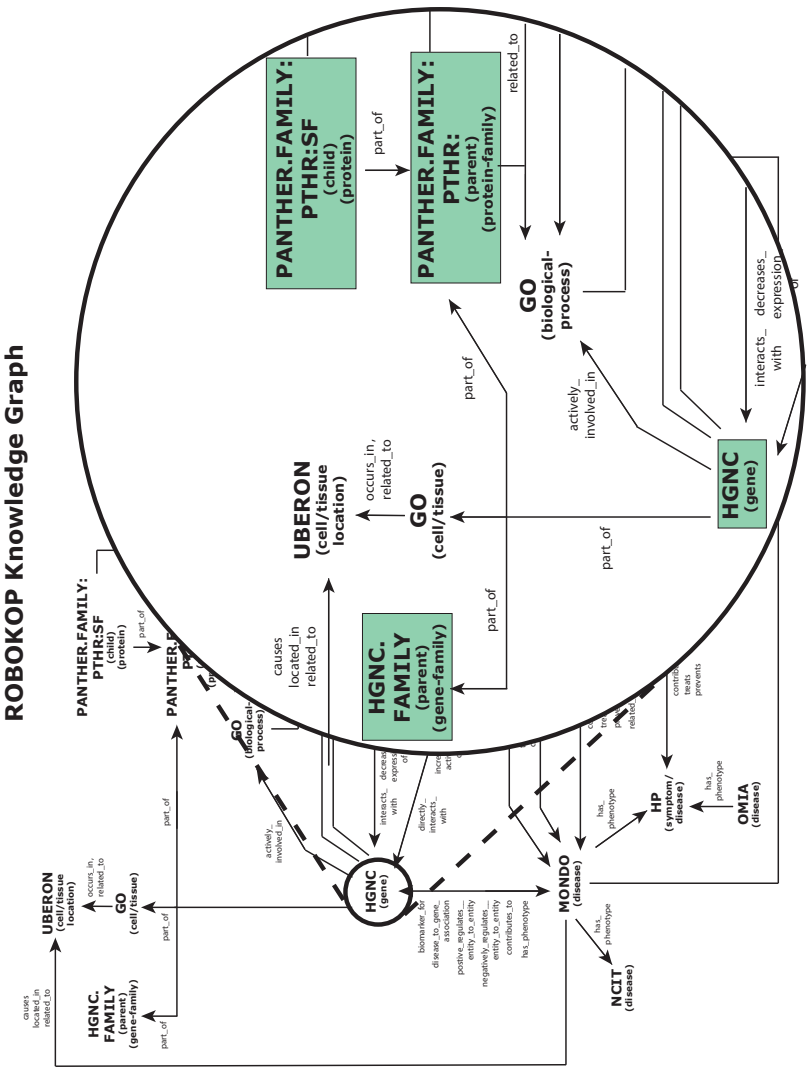


Fig. 11. Zoomed-in and green highlighted concepts represent gene and protein concept specific CURIES, and the predicate paths that connect them. Concept types are noted in parenthesis below each database name (bold). Edge predicates are located above each arrow, directing the relation between concepts.

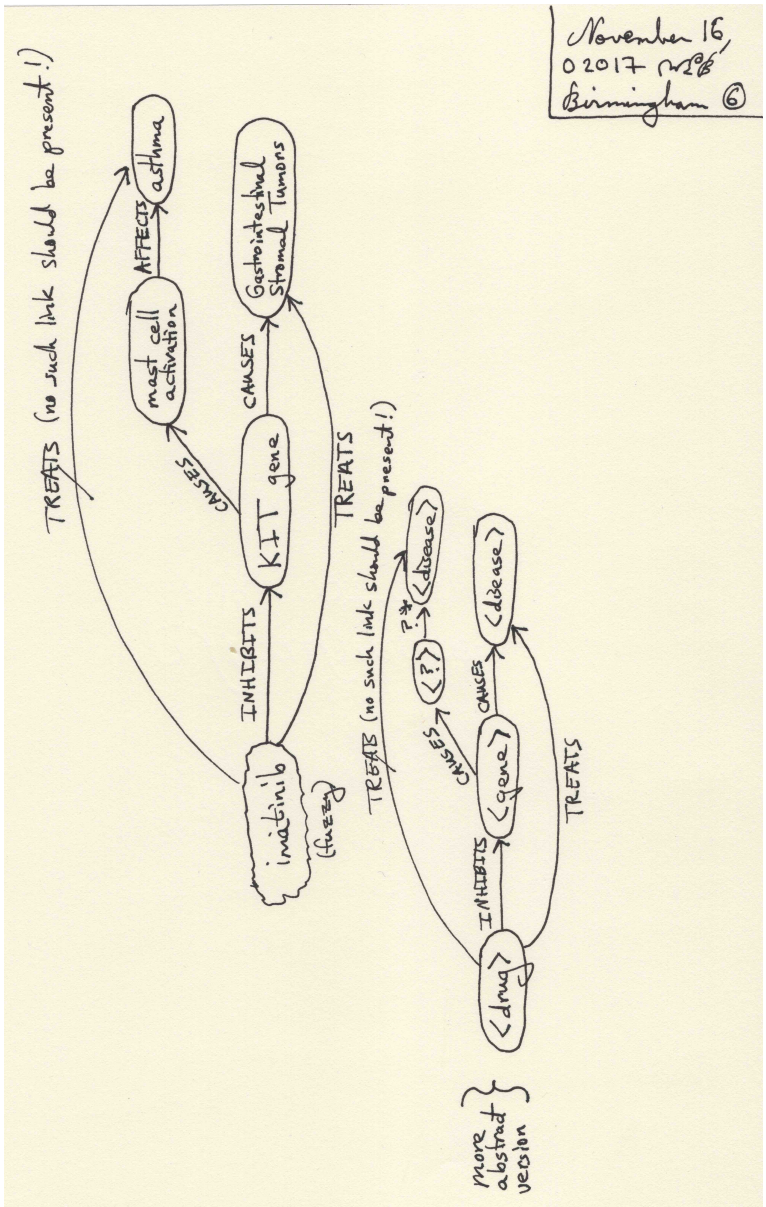


Fig. 12. Hand-drawn diagram from the early development of mediKanren. The upper-part of the diagram shows a subgraph from the Semantic MEDLINE Database (SemMedDB), based on the NCATS-provided drug-repurposing example query: "is there evidence that the cancer drug imatinib also treats asthma?" Even though imatininb is not known to treat asthma (at least in the medical literature), we can weigh the strength of the evidence that imatininb might treat asthma through some known mechanism of action, such as by inhibiting expression of a gene implicated in causing asthma. The subgraph shows that drugs in the "imatinib" class are known in the literature to treat gastrointestinal stromal tumors, and that (one) mechanism of action for this treatment is inhibition of KIT gene, whose over-expression is known to cause the tumors. KIT gene is also known to cause mast cell activation, which affects asthma. The overall subgraph provides evidence that imatinib might treat asthma, inferring potential new knowledge. Any algorithm weighing this evidence should take into account that the "affects" predicate is weaker than the "causes" predicate, and that the path from imatinib to asthma through KIT gene is one "hop" longer than the path from imatinib to gastrointestinal stromal tumors. The lower-part of the diagram shows a generalized schema of the concrete imatinib subgraph, which can be used as the basis of a query for discovering drug-repurposing candidates.

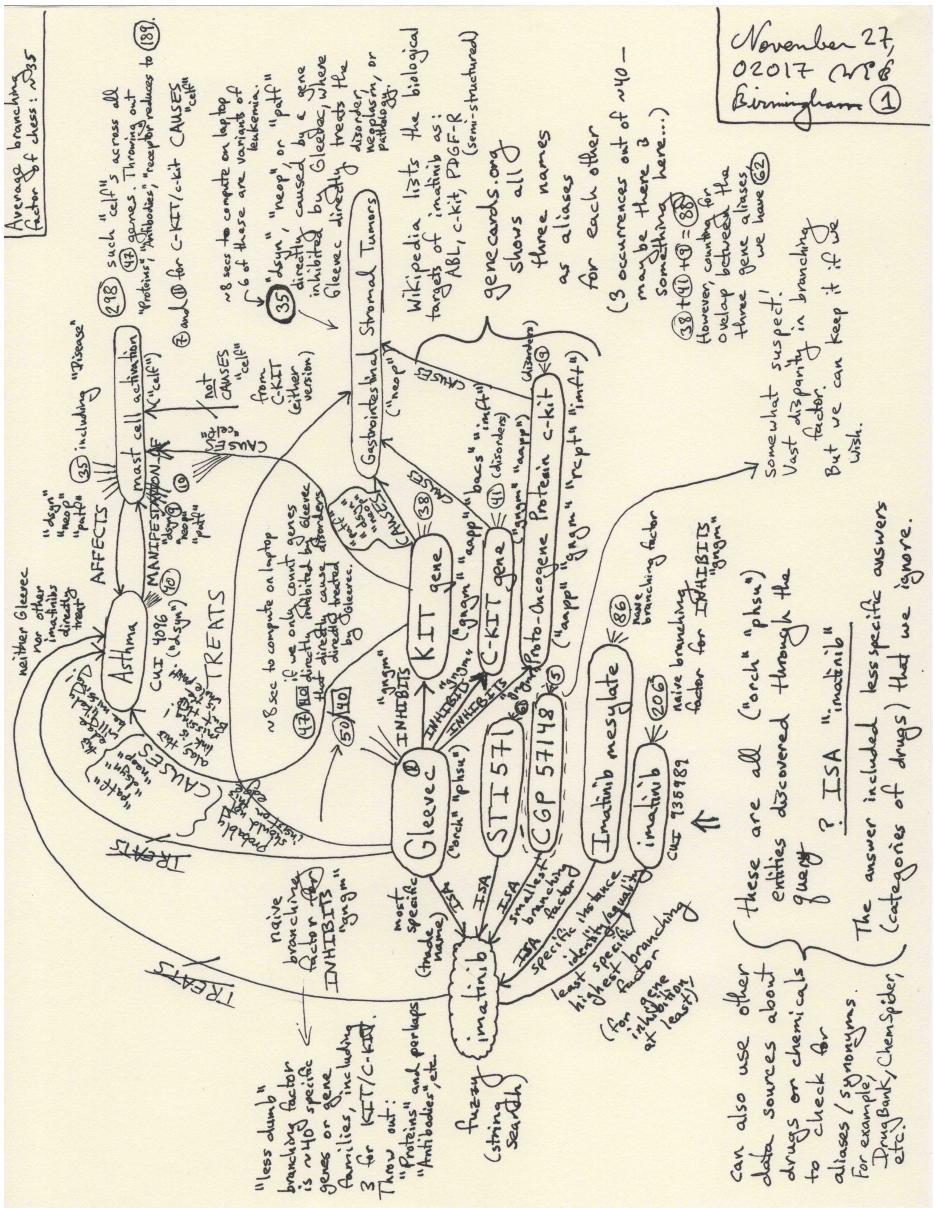


Fig. 13. Hand-drawn diagram from the early development of mediKanren. This diagram is a refined version of the Semantic MEDLINE Database (SemMedDB) subgraph from Figure 12, based on the NCATS-provided drug-repurposing example query: "is there evidence that the cancer drug imatinib also treats asthma?" This expanded subgraph shows different drugs that might match the search term "imatinib," along with multiple genes implicated in causing gastrointestinal stromal tumors. The figure also shows the branching factors of the edges that start with each node in the subgraph: for example, the C-KIT gene is known to cause 41 different disorders. For several nodes the branching factor is over 200; in comparison, the average branching factor in a game of chess is around 35.

# Relational Synthesis for Pattern Matching*

DMITRY KOSAREV, Saint Petersburg State University, Russia and JetBrains Research, Russia
DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We apply relational programming techniques to the problem of synthesizing efficient implementation for a pattern matching construct. Although in principle pattern matching can be implemented in a trivial way, the result suffers from inefficiency in terms of both performance and code size. Thus, in implementing functional languages alternative, more elaborate approaches are widely used. However, as there are multiple kinds and flavors of pattern matching constructs, these approaches have to be specifically developed and justified for each concrete inhabitant of the pattern matching "zoo." We formulate the pattern matching synthesis problem in relational terms and develop optimizations which improve the efficiency of the synthesis and guarantee the optimality of the result. Our approach is based on relational representations of both the high-level semantics of pattern matching and the semantics of an intermediate-level implementation language. This choice make our approach, in principle, more scalable as we only need to modify the high-level semantics in order to synthesize the implementation of a new feature. Our evaluation on a set of small samples, partially taken from existing literature shows, that our framework is capable of synthesizing optimal implementations quickly. Our in-depth stress evaluation on a number of artificial benchmarks, however, has shown the need for future improvements.

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

Additional Key Words and Phrases: relational programming, relational interpreters, pattern matching

## 1 INTRODUCTION

Algebraic data types (ADT) are an important tool in functional programming which deliver a way to represent flexible and easy to manipulate data structures. To inspect the contents of an ADT's values a generic construct — *pattern matching* — is used. Pattern matching can be considered as a generalization of conventional conditional control-flow construct "`if` .. `then` .. `else`" and in principle can be decomposed into a nested hierarchy of those; from this standpoint the problem of pattern matching implementation can be considered trivial. However, some decompositions are obviously better than others. We repeat here an example from [Maranget 2008] to demonstrate this difference (see Fig. 1). Here we match a triple of boolean values $x$, $y$, and $z$ against four patterns (Fig. 1a; we use OCaml [Leroy et al. 2020] as reference language). The naïve implementation of this example is shown on Fig. 1b; however if we decide to match $y$ first the result becomes much better (Fig. 1c).

The quality of a pattern matching implementation can be measured in various ways. One can either optimise the run time cost by minimizing the amount of checks performed, or the static cost by minimizing the size of the generated code. *Decision trees* are considered suitable for the first criterion as they check every subexpression no more than once. However, minimizing the size of decision tree is known to be NP-hard [Baudinet and MacQueen 1985], and as a rule various heuristics, using, for example, the number of nodes, the length of the longest path

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

Authors' addresses: Dmitry Kosarev, Saint Petersburg State University, Russia , JetBrains Research, Russia, Dmitrii.Kosarev@pm.me; Dmitry Boulytchev, Saint Petersburg State University, Russia , JetBrains Research, Russia, dboulytchev@math.spbu.ru.


miniKanren.org/workshop/2021/8-ART8

```
match x, y, z with
| _, F, T → 1
| F, T, _ → 2
| _, _, F → 3
| _, _, T → 4
```

(a) Pattern matching

```
if x then
  if y then
    if z then 4 else 3
  else
    if z then 1 else 3
else
  if y then 2
  else
    if z then 1 else 3
```

(b) A correct but non-optimal implementation

```
if y then
  if x then
    if z then 4 else 3
  else 2
else
  if z then 1 else 3
```

(c) Optimal implementation

Fig. 1. Pattern matching implementation example

and the average length of all paths are applied during compilation. In [Scott and Ramsey 2000] the results of experimental evaluation of nine heuristics for Standard ML of New Jersey are reported.

For minimizing the static cost *backtracking automata* can be used since they admit a compact representation but in some cases can perform repeated checks.

There is a certain difference in dealing with pattern matching in strict and non-strict languages. For strict languages checking sub-expressions of the scrutinee in any order is allowed. The pattern matching implementation for strict languages can operate in *direct* or *indirect* styles. In the direct style the construction of an implementation is done explicitly. In indirect the construction of implementation requires some post-processing, which can vary from easy simplifications to complicated supercompilation techniques [Sestoft 1996]. The main drawback of indirect approach is that the size of intermediate data structures can be exponentially large.

For non-strict languages pattern matching should evaluate only those sub-expressions which are necessary for performing pattern matching. If not done carefully pattern matching can change the termination behavior of the program. In general non-strict languages put more constraints on pattern matching and thus admit a smaller set of heuristics. A few approaches for checking sub-expressions in lazy languages have been proposed. In [Augustsson 1985] a simple left-to-right order of subexpression checking was proposed with a proof that this particular order doesn't affect termination. The backtracking automaton being built takes a form of a DAG to reduce the code size. A few refinements have been added in [Wadler 1987] as a part of textbook [Peyton Jones 1987] on the implementation of lazy functional languages. The approach from this book is used in the current version of GHC [Marlow and Peyton Jones 2012]. [Laville 1991] models values in lazy languages using *partial terms*, although it doesn't scale to types with infinite sets of constructors (like integers). The approach doesn't test all subexpressions from left to right as does [Augustsson 1985] but aims to avoid performing unnecessary checks by constructing *lazy automaton*. Pattern matching for lazy languages has been compiled also to decision trees [Maranget 1992] and later into *decision DAGs* which in some cases allows the compiler to make the code smaller [Maranget 1994].

The inefficiency of backtracking automata have been improved in [Le Fessant and Maranget 2001]. The approach utilizes a matrix representation for pattern matching. It splits the current matrix according to constructors in the first column and reduces the task to compiling matrices with fewer rows. The technique is indirect; in the end a few optimizations are performed by introducing special *exit* nodes to the compiled representation. The approach from this paper is used in the current implementation of the OCaml compiler.

The previous approach uses the first column to split the matrix. In [Maranget 2008] the *necessity* heuristic has been introduced which recommends which column should be used to perform the split. Good decision trees which are constructed in this work can perform better in corner cases than [Le Fessant and Maranget 2001], but for practical use the difference is insignificant.

While existing approaches deliver appropriate solutions for certain forms of pattern matching constructs, they have to be extended in an *ad hoc* manner each time the syntax and semantics of pattern matching construct changes. For example, besides a simple conventional form of pattern matching there are a number of extensions: guards (first appeared in KRC language [Turner 2013]), disjunctive patterns [Leroy et al. 2020], non-linear patterns [McBride et al. 1969], active patterns [Syme et al. 2007] and pattern matching for polymorphic variants [Garrigue 1998] which require a separate customized algorithms to be developed.

We present an approach to pattern matching implementation based on application of relational programming [Byrd 2009; Friedman et al. 2005] and, in particular, relational interpreters [Byrd et al. 2017] and relational conversion [Lozov et al. 2017]. Our approach is based on relational representation of the source language pattern matching semantics on the one hand, and the semantics of the intermediate-level implementation language on the other. We formulate the condition necessary for a correct and complete implementation of pattern matching and use it to construct a top-level goal which represents a search procedure for all correct and complete implementations. We also present a number of techniques which make it possible to come up with an *optimal* solution as well as optimizations to improve the performance of the search. Similarly to many other prior works we use the size of the synthesized code, which can be measured statically, to distinguish better programs. Our implementation[1] makes use of OCanren[2] — a typed implementation of miniKanren for OCaml [Kosarev and Boulytchev 2016], and noCanren[3] — a converter from the subset of plain OCaml into OCanren [Lozov et al. 2017]. On an initial evaluation, performed for a set of benchmarks taken from other papers, showed our synthesizer performing well. However, being aware of some pitfalls of our approach, we came up with a set of counterexamples on which it did not provide any results in observable time, so we do not consider the problem completely solved. We also started to work on mechanized formalization[4], written in Coq [Bertot and Castéran 2004], to make the justification of our approach more solid and easier to verify, but this formalization is not yet complete.

## 2 THE PATTERN MATCHING SYNTHESIS PROBLEM

We start from a simplified view on pattern matching which does not incorporate some practically important aspects of the construct such as name bindings in patterns, guards or even semantic actions in branches. In a purified form, however, it represents the essence of pattern matching as an "inspect-and-branch" procedure. Once we come up with the solution for the essential part of the problem we embellish it with other features until it reaches a complete form.

First, we introduce a finite set of *constructors* $\mathcal{C}$, equipped with arities, a set of values $\mathcal{V}$ and a set of patterns $\mathcal{P}$:

$$\begin{array}{rcl} \mathcal{C} & = & \{C_1^{k_1}, \ldots, C_n^{k_n}\} \\ \mathcal{V} & = & \mathcal{C}\,\mathcal{V}^* \\ \mathcal{P} & = & \_ \mid \mathcal{C}\,\mathcal{P}^* \end{array}$$

We define a matching of a value $v$ (*scrutinee*) against an ordered non-empty sequence of patterns $p_1, \ldots, p_k$ by means of the following relation

[1] https://github.com/kakadu/pat-match
[2] https://github.com/JetBrains-Research/OCanren
[3] https://github.com/Lozov-Petr/noCanren
[4] https://github.com/dboulytchev/Coq-matching-workout

$$\langle v;\ \_\rangle \qquad \text{[Wildcard]}$$

$$\frac{\forall i\ \ \langle v_i;\ p_i\rangle}{\left\langle C^k\, v_1 \ldots v_k;\ C^k\, p_1 \ldots p_k\right\rangle},\ k \geq 0 \qquad \text{[Constructor]}$$

Fig. 2. Matching against a single pattern

$$\frac{\langle v;\ p_1\rangle}{i \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* i} \qquad \text{[MatchHead]}$$

$$\frac{\neg\, \langle v;\ p_1\rangle \qquad i+1 \vdash \langle v;\ p_2, \ldots, p_k\rangle \longrightarrow_* j}{i \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* j} \qquad \text{[MatchTail]}$$

$$i \vdash \langle v;\ \varepsilon\rangle \longrightarrow_* i \qquad \text{[MatchOtherwise]}$$

$$\frac{1 \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* i}{\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i} \qquad \text{[Match]}$$

Fig. 3. Matching against an ordered sequence of patterns

$$\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i,\ 1 \leq i \leq k+1$$

which gives us the index of the leftmost matched pattern or $k+1$ if no such pattern exists. We use an auxiliary relation $\langle ; \rangle \subseteq \mathcal{V} \times \mathcal{P}$ to specify the notion of a value matched by an individual pattern (see Fig. 2). The rule [Wildcard] says that a wildcard pattern "_" matches any value, and [Constructor] specifies that a constructor pattern matches exactly those values which have the same constructor at the top level and all subvalues matched by corresponding subpatterns. The definition of "$\rightarrow$" is shown on Fig. 3. An auxiliary relation "$\rightarrow_*$" is introduced to specify the left-to-right matching strategy, and we use current index as an environment. An important rule, [MatchOtherwise] specifies that if we exhausted all the patterns with no matching we stop with the current index (which in this case is equal to the number of patterns plus one).

The relation "$\rightarrow$" gives us a *declarative* semantics of pattern matching. Since we are interested in synthesizing implementations, we need a *programmatical* view on the same problem. Thus, we introduce a language $\mathcal{S}$ (the "switch" language) of test-and branch constructs:

$$\begin{array}{rcl} \mathcal{M} & = & \bullet \\ & & \mathcal{M}\,[\mathbb{N}] \\ \mathcal{S} & = & \textbf{return}\,\mathbb{N} \\ & & \textbf{switch}\ \mathcal{M}\ \textbf{with}\ [C \rightarrow \mathcal{S}]^*\ \textbf{otherwise}\ \mathcal{S} \end{array}$$

Here $\mathcal{M}$ stands for a *matching expression*, which is either a reference to a scrutinee "$\bullet$" or an indexed subexpression of scrutinee. Programs in the switch language can discriminate based on the structure of matching

$$v \vdash \bullet \longrightarrow_{\mathcal{M}} v \qquad \text{[Scrutinee]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} C^k v_1 \ldots v_k}{v \vdash m[i] \longrightarrow_{\mathcal{M}} v_i} \qquad \text{[SubMatch]}$$

Fig. 4. Semantics of matching expression

$$v \vdash \textbf{return } i \longrightarrow_{\mathcal{S}} i \qquad \text{[Return]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} C^k\, v_1 \ldots v_k \qquad v \vdash s \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } [C^k \to s]s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchMatched]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} D^n\, v_1 \ldots v_n \qquad C^k \neq D^n \qquad v \vdash \textbf{switch } m \textbf{ with } s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } [C^k \to s]s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchNotMatched]}$$

$$\frac{v \vdash s \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } \varepsilon \textbf{ otherwise } s \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchOtherwise]}$$

Fig. 5. Semantics of switch programs

expressions, testing their top-level constructors and eventually returning natural numbers as results. The switch language is similar to the intermediate representations for pattern matching code used in previous works on pattern matching implementation [Le Fessant and Maranget 2001; Maranget 2008].

The semantics of the switch language is given by mean of relations "$\rightarrow_{\mathcal{M}}$" and "$\rightarrow_{\mathcal{S}}$" (see Fig. 4 and 5). The first one describes the semantics of matching expression, while the second describes the semantics of the switch language itself. In both cases the scrutinee $v$ is used as an environment ($v \vdash$).

The following observations can be easily proven by structural induction.

Observation 1. *For arbitrary pattern the set of matching values is non-empty:*

$$\forall p \in \mathcal{P} : \{v \in \mathcal{V} \mid \langle v;\, p\rangle\} \neq \varnothing$$

Observation 2. *Relations "$\rightarrow$" and "$\rightarrow_{\mathcal{S}}$" are functional and deterministic respectively:*

$$\forall p_1, \ldots, p_k \in \mathcal{P},\ \forall v \in \mathcal{V},\ \forall \pi \in \mathcal{S} \quad : \quad \begin{aligned} &|\{i \in \mathbb{N} \mid \langle v;\, p_1, \ldots, p_k\rangle \longrightarrow i\}| = 1 \\ &|\{i \in \mathbb{N} \mid v \vdash \pi \longrightarrow_{\mathcal{S}} i\}| \leq 1 \end{aligned}$$

With these definitions, we can formulate the *pattern matching synthesis problem* as follows: for a given ordered sequence of patterns $p_1, \ldots, p_k$ find a switch program $\pi$, such that

$$\forall v \in \mathcal{V},\ \forall 1 \leq i \leq n+1 : \langle v;\, p_1, \ldots, p_n\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i \qquad (\star)$$

In other words, program $\pi$ delivers a correct and complete implementation for pattern matching.

## 3 PATTERN MATCHING SYNTHESIS, RELATIONALLY

In this section we describe a relational formulation for the pattern matching synthesis problem. Practically, this amounts to constructing a goal with a free variable corresponding to the switch program to synthesize for (arbitrary) list of patterns. In order to come up with a tractable goal certain steps have to be performed. We first describe the general idea, and then consider these steps is details.

Our idea of using relational programming for pattern matching synthesis is based on the following observations:

- For the switch language we can implement a relational interpreter $eval^o_{\mathcal{S}}$ with the following property: for arbitrary $v \in \mathcal{V}$, $\pi \in \mathcal{S}$ and $i \in \mathbb{N}$

$$eval^o_{\mathcal{S}}\ v\ \pi\ i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i$$

  In other words, $eval^o_{\mathcal{S}}$ interprets a program $\pi$ for a scrutinee $v$ and returns exactly the same branch (if any) which is prescribed by the semantics of the switch language.
- On the other hand, we can directly encode the declarative semantics of pattern matching as a relational program $match^o$ such that for arbitrary $v \in \mathcal{V}$, $p_i \in \mathcal{P}$ and $i \in \mathbb{N}$

$$match^o\ v\ p_1, \ldots, p_k\ i \iff \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$$

  Again, $match^o$ succeeds with $1 \le i \le k$ iff $p_i$ is the leftmost pattern, matching $v$; otherwise it succeeds with $i = k + 1$.

We address the construction of relational interpreters for both semantics in Section 3.1.

Being relational, both $eval^o_{\mathcal{S}}$ and $match^o$ do not just succeed or fail for ground arguments, but also can be *queried* for arguments with free logical variables, thus performing a search for all substitutions for these variables which make the relation hold. This observation leads us to the idea of utilizing the definition of the pattern matching synthesis problem, replacing "$\rightarrow$" with $match^o$, "$\rightarrow_{\mathcal{S}}$" with $eval^o$, and $\pi$ with a free logical variable $\textcircled{?}$, which gives us the goal

$$\forall v \in \mathcal{V},\ \forall 1 \le i \le n+1 : match^o\ v\ p_1, \ldots, p_n\ i \iff eval^o\ v\ \textcircled{?}\ i$$

This goal, however, is problematic from relational point of view due to a number of reasons.

First, miniKanren provides rather a limited support for universal quantification. Apart from being inefficient from a performance standpoint, existing implementations either do not coexist with disequality constraints [Byrd [n. d.]] or do not support quantified goals with infinite number of answers [Moiseenko 2019]. As we will see below, both restrictions are violated in our case. Second, there is no direct support for the equivalence of goals ("$\Leftrightarrow$"). Thus, reducing the original synthesis problem to a viable relational goal involves some "massaging".

We eliminate the universal quantification over the infinite set of scrutinees, replacing it by a *finite* conjunction over a *complete set of samples*. For a sequence of patterns $p_1, \ldots, p_k$ a complete set of samples is a finite set of values $\mathcal{E}(p_1, \ldots, p_k) \subseteq \mathcal{V}$ with the following property:

$$\begin{aligned}\forall \pi \in \mathcal{S}\ \ :\ \ & [\forall v \in \mathcal{E}(p_1, \ldots, p_k), \forall i \in \mathbb{N} : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i] \Longrightarrow \\ & [\forall v \in \mathcal{V}, \forall i \in \mathbb{N} : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i]\end{aligned}$$

In other words, if a program implements a correct and complete pattern matching for all values in a complete set of samples, then this program implements a correct and complete pattern matching for all values. The idea of using a complete set of samples originates from the following observation: each pattern describes a (potentially infinite) set of values, and pattern matching splits the set of all values into equivalence classes, each corresponding to a certain matching pattern. Moreover, the values of different classes can be distinguished only by looking

down to a *finite* depth (as different patterns can be distinguished in this way). The generation of complete sample set will be addressed below (see Section 3.2).

To eliminate the universal quantification over the set of answers we rely on the functionality of declarative pattern matching semantics. Indeed, given a fixed sequence $p_1, \ldots, p_k$ of patterns, for every value $v$ there is exactly one answer value $i$, such that $\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$. We can reformulate this property as

$$\exists i :\ \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \Longrightarrow \big(\forall j : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow j \Longrightarrow j = i\big)$$

Thus, we can replace universal quantification over the sets of answers by existential one, for which we have an efficient relational counterpart — the "**fresh**" construct.

Following the same argument, we may replace the equivalence with conjunction: indeed, if

$$\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$$

for some $i$, then (by functionality), for any other $j \neq i$

$$\neg\ (\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow j)$$

A correct pattern matching implementation $\pi$ should satisfy the condition

$$v \vdash \pi \longrightarrow_{\mathcal{S}} i$$

But, by the determinism of the switch language semantics, it immediately follows, that for arbitrary $j \neq i$

$$\neg\ (v \vdash \pi \longrightarrow_{\mathcal{S}} j)$$

Thus, the goal we eventually came up with is

$$\bigwedge_{v \in \mathcal{E}\ (p_1,\ldots,p_k)} \textbf{fresh}\ (i)\ \{match^o\ v\ p_1, \ldots, p_k\ i \wedge eval^o_{\mathcal{S}}\ v\ \textcircled{?}\ i\} \qquad (\star\star)$$

From relational point of view this is a pretty conventional goal which can be solved by virtually any decent miniKanren implementation in which the relations $eval^o_{\mathcal{S}}$ and $match^o$ can be encoded.

Finally, we can make the following important observation. Obviously, any pattern matching synthesis problem has at least one trivial solution. This, due to the completeness of relational interleaving search [Kiselyov et al. 2005; Rozplokhas and Boulytchev 2019], means that the goal above *can not diverge* with no results. Actually it is rather easy to see that any pattern matching synthesis problem has *infinitely many* solutions: indeed, having just one it is always possible to "pump" it with superfluous "**otherwise**" clauses; thus, the goal above is *refutationally complete* [Byrd 2009; Rozplokhas and Boulytchev 2018]. These observations justify the totality of our synthesis approach. In Section 4 we show how we can make it provide optimal solution.

### 3.1 Constructing Relational Interpreters

In this section we address the implementation of relations $eval^o_{\mathcal{S}}$ and $match^o$. In principle, it amounts to accurate encoding of relations "$\Rightarrow$" and "$\Rightarrow_{\mathcal{S}}$" in miniKanren (in our case, OCanren). We, however, make use of a relational conversion [Lozov et al. 2017] tool, called noCanren, which automatically converts a subset of OCaml into OCanren. Thus, both interpreters are in fact implemented in OCaml and repeat corresponding inference rules almost literally in a familiar functional style. For example, functional implementation of a declarative semantics looks like follows:

```
let rec ⟨v; p⟩ =
  match (v, p) with
  | (_, Wildcard)  → true
  | (C^k v*, C^k p*) → list_all ⟨;⟩ (list_combine v* p*)
  | _              → false

let match^o v p* =
  let rec inner i p* =
    match p* with
    | []       → i
    | p :: p*  → if ⟨v; p⟩ then i else inner S(i) p*
  in inner 0 p*
```

We mixed here the concrete syntax of OCaml and mathematical notation, used in the definition of the relation in question; the actual implementation only a few lines of code longer. Note, we use here natural numbers in Peano form and custom list processing functions in order to apply relational conversion later.

Using relational conversion saves a lot of efforts as OCanren specifications tend to be much more verbose; in addition relational conversion implements some "best practices" in relational programming (for example, moves unifications forward in conjunctions and puts recursive calls last). Finally, it has to be taken into account that relational conversion of pattern matching introduces disequality constraints.

### 3.2 Dealing with a Complete Set of Samples

As we mentioned above, a complete set of samples plays an important role in our approach: it allows us to eliminate universal quantification over the set of all values. As we replace the universal quantifier with a finite conjunction with one conjunct per sample value reducing the size of the set is an important task. At the present time, however, we build an excessively large (worst case exponential of depth) number of samples. We discuss the issues with this choice in Section 5 and consider developing a more advanced approach as the main direction for improvement.

Our construction of a complete set of samples is based upon the following simple observations. We simultaneously define the *depth* measure for patterns and sequences of patterns as follows:

$$\begin{aligned} d\,(p_1, \ldots, p_k) &= max\,\{d\,(p_i)\} \\ d\,(\_) &= 0 \\ d\,(C^k\,p_1, \ldots p_k) &= 1 + d\,(p_1, \ldots, p_k) \end{aligned}$$

As a sequence of patterns is the single input in our synthesis approach we will call its depth *synthesis depth.*

Similarly, we define the depth of matching expressions

$$\begin{aligned} d_{\mathcal{M}}\,(\bullet) &= 1 \\ d_{\mathcal{M}}\,(m\,[i]) &= 1 + d_{\mathcal{M}}\,(m) \end{aligned}$$

and switch programs:

$$\begin{aligned} d_{\mathcal{S}}\,(\textbf{return}\ i) &= 0 \\ d_{\mathcal{S}}\,(\textbf{switch}\ m\ \textbf{of}\ C_1 \to s_1, \ldots, C_k \to s_k\ \textbf{otherwise}\ s) &= max\,\{d_{\mathcal{M}}\,(m),\, d_{\mathcal{S}}\,(s_i),\, d_{\mathcal{S}}\,(s)\} \end{aligned}$$

Informally, the depth of a switch program tells us how deep the program can look into a value.

From the definition of $\langle ; \rangle$ it immediately follows that a pattern $p$ can only discriminate values up to its depth $d\,(p)$: changing a value at the depth greater or equal than $d\,(p)$ cannot affect the fact of matching/non matching. This means that we need only consider switch programs of depth no greater than the synthesis depth. But for these programs the set of all values with height no greater than the synthesis depth forms a complete set of samples. Indeed, if the height of a value less or equal to the synthesis depth, then this value is a member of complete set of samples and by definition the behavior of the synthesized program on this value is correct. Otherwise there exists some value $s$ from the complete set of samples, such that given value can be obtained as an "extension" of $s$ at the depth greater than the synthesis depth. Since neither declarative semantics nor switch programs can discriminate values at this depth, they behavior for a given value will coincide with the correct-by-definition behavior for $s$.

The implementation of complete set generation, again, is done using relational conversion. The enumeration of all terms up to a certain depth can be acquired from a function which calculates the depth of a term: indeed, converting it into a relation and then running with *fixed* depth and *free* term arguments delivers what we need. Thus, we add an extra conjunct which performs the enumeration of all values to the relational goal ($\star\star$), arriving at

$$depth^o\ v\ n \wedge \textbf{fresh}\ \ (i)\ \{match^o\ v\ p_1, \ldots, p_k\ i \wedge eval^o_{\mathcal{S}}\ v\ \textcircled{?}\ i\} \qquad (\star\star\star)$$

Here $n$ is a precomputed synthesis depth in Peano form.

## 4 IMPLEMENTATION AND OPTIMIZATIONS

In this section we address two aspects of our solution: a number of optimizations which make the search more efficient, and the way it ends up with the optimal solution.

The relational goal in its final form, presented in the previous section, does not demonstrate good performance. Thus, we apply a number of techniques, some of which require extending the implementation of the search. Namely, we apply the following optimizations:

- We make use of type information to restrict the subset of constructors which may appear in a certain branch of program being synthesized.
- We implement *structural constraints* which allow us to restrict the shape of terms during the search, and utilize them to implement pruning.

In our formalization we do not make any use of types since as a rule type information does not affect matching. In addition, utilizing the properties of a concrete type system would make our approach too coupled with this particular type system, hampering its reusability for other languages. Nevertheless we may use a certain abstraction of type system which would deliver only that part of information which is essential for our approach to function. Currently, we calculate the type of any matching expression in the program being synthesized and from this type extract the subset of constructors which can appear when branching on this expression is performed. The number of these constructors restricts the number of branches which a corresponding **switch** expression can have. In our implementation we assume the constructor set ordered, and we consider only ordered branches, which restricts branching even more.

Our approach to finding an optimal solution in fact implements branch-and-bound strategy. The birds-eye view of our plan is as follows:

| Patterns | Size constraint | Answers requested | Number of samples | 1st answer time (ms) | Answers found | Total search time (ms) |
|---|---|---|---|---|---|---|
| `A`<br>`B`<br>`C` | 100 | all | 3 | 1 | 1 | 1 |
| **`true`**<br>**`false`** | 100 | all | 2 | <1 | 1 | <1 |
| (**`true`**, _)<br>(_, **`true`**)<br>(**`false`**, **`false`**) | 100 | all | 4 | 6 | 2 | 10 |
| (_, **`false`**, **`true`**)<br>(**`false`**, **`true`**, _)<br>(_, _, **`false`**)<br>(_, _, **`true`**) | 100 | all | 8 | 323 | 3 | 729 |
| ([], _)<br>(_, [])<br>(_ :: _, _ :: _) | 100 | 10 | 4 | 5 | 1 | 6 |
| (Succ _, Succ _)<br>(Zero, _)<br>(_, Zero) | 1 | all | 4 | 53 | 2 | 108 |
| (Nil, _)<br>(_, Nil)<br>(Nil2, _)<br>(_, Nil2)<br>(_ :: _, _ :: _) | 1 | 10 | 9 | 643 | 2 | 3776 |
| | 10 | 10 | 9 | 95 | 2 | 540 |
| | 100 | 10 | 9 | 45 | 2 | 239 |

Fig. 6. The results of synthesis evaluation

- We construct a trivial solution, which gives us the first estimation.
- During the search we prune all partial solutions whose size exceeds current estimation. We can do this due to the top-down nature of partial solution construction.
- When we come up with a better solution we remember it and update current a estimate.

This strategy inevitably delivers us the optimal solution since there are only finitely many switch programs, shorter than trivial solution.

In order to implement this strategy we extended OCanren with a new primitive called *structural constraint*, which may fail on some terms depending on some criterion specified by an end-user. Structural constraints can be seen as a generalization of some known constraints like `absent`$^o$ or `symbol`$^o$ [Byrd et al. 2012] in existing miniKanren implementations, so they can be widely used in solving other problems as well. Note, we could

```
let rec run a s c =
  match a,s,c with
  | (_,_,Ldi i::_)  → 1
  | (_,_,Push::_)   → 2
  | (Int _,Val (Int _)::_,IOp _::_)  → 3
  | (Int _,_,Test (_,_)::c)  → 4
  | (Int _,_,Test (_,_)::c)  → 5
  | (_,_,Extend::_)  → 6
  | (_,_,Search _::_)  → 7
  | (_,_,Pushenv::_)  → 8
  | (_,Env e::s,Popenv::_)  → 9
  | (_,_,Mkclos cc::_)  → 10
  | (_,_,Mkclosrec _::_)  → 11
  | (Clo (_,_), Val _::_, Apply::_)  → 12
  | (_,(Code _::Env _::_),[])  → 13
  | (_,[],[])  → 14
```

Fig. 7. An example from a bytecode machine for PCF

implement other constraints we considered (on the depth of switch programs, on the type of scrutinee) as structural. However, our experience has shown that this leads to a less efficient implementation. Since these constraints are inherent to the problem, we kept them "hard coded".

## 5 EVALUATION

We performed an evaluation of the pattern matching synthesizer on a number of benchmarks. The majority of benchmarks were prepared manually; we didn't use any specific benchmark sets mentioned in literature [Scott and Ramsey 2000] yet. The 4th benchmark was taken from [Maranget 2008], we used it in Section 1 as the first example. The evaluation was performed on a desktop computer with Intel Core i7-4790K CPU @ 4.00GHz processor and 8GB of memory, OCanren was compiled with ocaml-4.07.1+fp+flambda. All benchmarks were executed in the native mode ten times, then average monotonic clock time was taken. The results of the evaluation are shown on Figure 6.

In the table the double bar separates input data from output. Inputs are: the patterns used for synthesis, the requested number of answers and the pruning factor. For example, in the first benchmark pruning factor equals 100 which means that structural constraint is checked every 100 unifications. Outputs are: the size of generated complete samples set, the running time before receiving the first answer, the total number of programs synthesized, the total search time.

Our approach currently does not work fast on a large benchmark. On Fig. 7 we cite an example extracted from a bytecode machine for PCF [Maranget 2008; Plotkin 1977]. For such complex examples (in terms of type definition complexity and number and size of patterns) both the size of the search space and the number of samples is too large for our approach to work so far.

## 6 CONCLUSION AND FUTURE WORK

We presented an approach for pattern matching implementation synthesis using relational programming. Currently, it demonstrates a good performance only on a very small problems. The performance can be improved by searching for new ways to prune the search space and by speeding up the implementation of relations and structural constraints. Also it could be interesting to integrate structural constraints more closely into OCanren's core. Discovering an optimal order of samples and reducing the complete set of samples is another direction for research.

The language of intermediate representation can be altered, too. It is interesting to add to an intermediate language so-called *exit nodes* described in [Le Fessant and Maranget 2001]. The straightforward implementation of them might require nominal unification, but we are not aware of any miniKanren implementation in which both disequality constraints and nominal unification [Byrd and Friedman 2007] coexist nicely.

At the moment we support only simple pattern matching without any extensions. It looks technically easy to extend our approach with non-linear and disjunctive patterns. It will, however, increase the search space and might require more optimizations.

## REFERENCES


Lennart Augustsson. 1985. Compiling Pattern Matching. In *FPCA*.

Marianne Baudinet and David MacQueen. 1985. Tree pattern matching for ML. (1985).

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

William Byrd. [n. d.]. Relational Synthesis of Programs. ([n. d.]). http://webyrd.net/cl/cl.pdf

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

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *PACMPL* 1, ICFP (2017), 8:1–8:26. https://doi.org/10.1145/3110252

William E. Byrd and Daniel P. Friedman. 2007. $\alpha$kanren: A Fresh Name in Nominal Logic Programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*. 79–90.

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 2012, Copenhagen, Denmark, September 9-15, 2012*. 8–29. https://doi.org/10.1145/2661103.2661105

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

Jacques Garrigue. 1998. Programming with polymorphic variants. In *In ACM Workshop on ML*.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). (2005), 192–203. https://doi.org/10.1145/1086365.1086390

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. (2016), 1–22. https://doi.org/10.4204/EPTCS.285.1

Alain Laville. 1991. Comparison of Priority Rules in Pattern Matching and Term Rewriting. *J. Symb. Comput.* 11 (1991), 321–347.

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. *SIGPLAN Not.* 36, 10 (Oct. 2001), 26–37. https://doi.org/10.1145/507669.507641

Xavier Leroy, Damien Doligez, Jacques Garrigue Alain Frisch, Didier Rémy, and Jérôme Vouillon. 2020.

Petr Lozov, Andrei Vyatkin, and Dmitri Boulytchev. 2017. Typed Relational Conversion. In *TFP*.

Luc Maranget. 1992. Compiling lazy pattern matching. In *LFP '92*.

Luc Maranget. 1994. Two Techniques for Compiling Lazy Pattern Matching.

Luc Maranget. 2008. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML (ML '08)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/1411304.1411311

Simon Marlow and Simon Peyton Jones. 2012. *The Glasgow Haskell Compiler* (the architecture of open source applications, volume 2 ed.). Lulu. https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/

FV McBride, DJT Morrison, and RM Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969), 337–347.

Eugene Moiseenko. 2019. Constructive Negation for miniKanren. In *miniKanren and Relational Programming Workshop*.

Simon Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall. https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/

G. D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5 (1977), 223–255.

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/3236950.3236958

Dmitry Rozplokhas and Dmitry Boulytchev. 2019. Certified Semantics for miniKanren. In *miniKanren and Relational Programming Workshop*.

Kevin D Scott and Norman Ramsey. 2000. When Do Match-compilation Heuristics Matter?

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 446–464.

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible Pattern Matching via a Lightweight Language Extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/1291151.1291159

D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

Philip Wadler. 1987. Compilation of pattern matching.

# Some Novel miniKanren Synthesis Tasks

JASON HEMANN, Northeastern University, USA
DANIEL P FRIEDMAN, Indiana University, USA

Synthesizing quines with a relational interpreter in miniKanren is a relatively simple but not infrequently arresting example. We exhibit several related, novel synthesis tasks—either pre-existing programming challenges we newly solve with miniKanren synthesis, or newly formulated challenges of our own. In doing so we exhibit a relational interpreter for a "mirrored" language. These examples demonstrate miniKanren's potential and versatility as a platform or substrate for experimenting on executable program specifications. As with quines, these examples are themselves interesting from a theoretical perspective, may also suggest other, more practical future applications.

CCS Concepts: • **Theory of computation** → *Constraint and logic programming*; • **Software and its engineering** → **Constraint and logic languages**; **Automatic programming**;

Additional Key Words and Phrases: relational, program synthesis, quine, miniKanren, logic programming

> Difficult mathematical type exercise: Find a list e such that *value e* = *e*.
>
> John McCarthy, *A micro-manual for LISP-not the whole truth*

## 1 INTRODUCTION

Constraint-logic and relational programming languages such as miniKanren have found application in program synthesis tasks [7]. Although Kanren languages are also applicable to many of the standard logic programming tasks, such synthesis applications have arguably become Kanren languages' foremost specialty.

Generating *quines* is an early, not-infrequently arresting example of relational programming and of relational program-synthesis in miniKanren [8]. In his book *Gödel, Escher, Bach*, Hofstadter coined "quining" to mean "the operation of preceding some phrase by its quotation". This self-reference guarantees a quine—a program that evaluates to its own source code—in any sufficient programming language (e.g., the language of Listing 1). The existence of quines follows from Kleene's second recursion theorem (see e.g., Moschovakis [22] for details). Lee [16] and Thatcher [29] exhibit early examples of Turing machines that print their own descriptions. Bilar and Filiol [3] credit Hamisch Dewar with producing the earliest known self-replicating program, written in Atlas Autocode at the University of Edinburgh in the 1960s. The traditional, textbook proofs of quines' existence (e.g., Yasuhara [32, p. 102]) uses Gödel encodings of Turing machines, and is a bit afield from the programmer who wants to experiment.

This same demonstration is more straightforwardly expressed in miniKanren with the query in Listing 1. We remind the reader that when miniKanren constrains the synthesized programs' variables, these constraints are most often to exclude distinguished tags on defunctionalized closures, to prevent shadowing special forms in

Authors' addresses: Jason Hemann, jhemann@northeastern.edu, Northeastern University, USA; Daniel P Friedman, dfried@indiana.edu, Indiana University, USA.


miniKanren.org/workshop/2021/8-ART9

```
> (run 1 (q) (evalo q q))
'((((λ (_0) (list _0 (list 'quote _0)))
    '(λ (_0) (list _0 (list 'quote _0))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Listing 1. Querying against a relational interpreter for a quine

```
> (run 1 (v t u q p)
    (all-diffo '(v t u q p))
    (evalo q p) (evalo p q) (evalo u q) (evalo t u) (evalo v t))
'(((''''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
         '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   '''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
       '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   ''((λ (_0) (list 'quote (list _0 (list 'quote _0))))
      '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   '((λ (_0) (list 'quote (list _0 (list 'quote _0))))
     '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
   ((λ (_0) (list 'quote (list _0 (list 'quote _0))))
    '(λ (_0) (list 'quote (list _0 (list 'quote _0))))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0)))
```

Listing 2. Successful query for a binary iterating quine with a three-step transient.

the environment, or to restrict a logic variable to a sub-sort of the domain (e.g., the symbols). The language of Listing 1's interpreter, `evalo`, can produce only *proper quines*. Briefly, a proper quine has distinguishable and separable data and code components (disqualifying blank files or self-evaluating literals) and that does not reflect on its source code, e.g., through the file-system.[1] No interpreter in this paper can produce an improper quine, and so we use "quine" as a shorthand without ambiguity.

miniKanren can produce still more complicated examples with similar techniques. Byrd et al. [8] show how to implement "twines" (twin-quines) and "thrines" as well. The relational interpreter makes producing examples of such programs trivial with similar, slightly more sophisticated queries. Pagiamtzis [25] defines an *oscillating quine with initial transients*, by analogy to electrical engineering, as a sequence of distinct programs $p_0, \ldots, p_{k-1}, p_k, \ldots, p_n$, where each $p_i$ evaluates to $p_{i+1}$, and where $p_n$ evaluates to $p_k$. More precisely, this sequence of programs forms an $n-k$-quine cycle with a $k-1$ initial transient. Listing 2 demonstrates the query for such a program. The auxiliary `all-diffo` relation imposes disequality constraints between every two programs $p_i$ and $p_j$, $i \neq j$ in that list. From left to right, the first two evaluations in the query describe the quine cycle, and the remaining three evaluations describe the antecedent transient programs, in reverse order. miniKanren uses an interleaving depth-first search (see Rozplokhas et al. [27]) biased toward earlier successful branches of the search [31, p. 18]. In this order, miniKanren spends more of its search time looking for a twine, and with a twine in hand miniKanren can quickly synthesize the antecedent programs.

[1] See Madore [18] or the Quine contest rules at codegolf.meta.stackexchange.com/questions/4877 for more.

```
(define-relation (tsil-reporpo pxe vars env lav)
  (conde
    [(== `(tsil) pxe)
     (== `() lav)]
    [(fresh (a d t-a t-d)
       (== `(,a . ,d) pxe)
       (== `(,t-a . ,t-d) lav)
       (fo-lavo a vars env t-a)
       (tsil-reporpo d vars env t-d))]))
```

Listing 3. The `tsil-reporpo` help relation for the `fo-lavo` of Listing 4

Self-replicating programs do make occasional appearances in applied research, especially in security [30]. Writing quines, however, is usually a hacker's pastime (or even art [5]). Generating quines is not in and of itself an especially practical example of program synthesis. Byrd et al. [7] demonstrate that generating quines can be a first step toward more complex, useful synthesis tasks. We find this paper's novel miniKanren synthesis tasks interesting for similar reasons. To more practically-oriented readers we suggest that—like the earlier quine generation results—even if the programs we synthesize here are not themselves practically interesting, they may suggest related programs with practical applications and future uses of similar techniques to construct such related practical programs. Some of these tasks (e.g., Section 6) are novel in that the authors have found no earlier references that address the questions we raise and have not seen their specifications beyond our own work. We find intrinsically motivating how relational synthesis techniques suggest simple, novel, and (subjectively) interesting program specifications. Our other synthesis examples' specifications are not in and of themselves novel, but usually presentations offer these tasks as coding challenges for programmers; the authors are unaware of previous miniKanren syntheses of such programs. These examples also explore the space of programs and demonstrate how to experiment with a relational interpreter.

In Section 2 we introduce a version of Byrd et al.'s [8] relational interpreter with a reversed syntax and explain its implementation's unique bits. Then in Section 3 we remind the reader of polyglots and demonstrate a successful query for valid structurally-palindromic programs. In Section 4, we describe a technique for generating quine-relays. In Section 5, we synthesize programs that bootstrap quines. In Section 6, we specify a kind of quasi-quine cycles, exhibit both trivial and non-trivial programs that meet the specification and generalize this result to broader classes of such programs. In Section 7 we describe related synthesis work, and in Section 8 we conclude. The examples of Sections 2, 3 and 6 were previously published in Hemann [13, chap. 4]. At github.com/jasonhemann/novel-miniKanren-synthesis-tasks we provide the source code for our examples as well as instructions to execute it. In places we have slightly reformatted the output for spacing and clarity.

## 2 MIRRORED LANGUAGE INTERPRETER

Many of the following synthesis problems involve executing programs across multiple languages. To that end, we implement a relational interpreter like that of Byrd et al., but with the language's syntax reversed. That is, a program `((λ (arg) (list arg)) (quote cat))` would be `((cat etouq) ((arg tsil) (arg) adbmal))` in this reversed language. The two languages have the same semantics, only the concrete syntax has changed.

Listing 4 contains the mirrored language interpreter. While the syntax of the language is mirrored, the internal representations of closures and environments are not. Naturally enough, we named this relation `laveo`, which calls to `fo-lavo`. The one interesting difference is that testing for `tsil` expressions now requires a help relation.

```
(define-relation (laveo pxe lav)
  (fo-lavo pxe '() '() lav))

(define-relation (fo-lavo pxe vars env lav)
  (conde
    [(fresh (v)
       (== `(,v etouq) pxe)
       (absento 'etouq vars)
       (absento 'closure v)
       (== v lav))]
    [(absento 'tsil vars)
     (absento 'closure pxe)
     (tsil-reporpo pxe vars env lav)]
    [(symbolo pxe) (lookupo pxe vars env lav)]
    [(fresh (rotar dnar x ydob vars^ env^ a)
       (== `(,dnar ,rotar) pxe)
       (fo-lavo rotar vars env `(closure ,x ,ydob ,vars^ ,env^))
       (fo-lavo dnar vars env a)
       (fo-lavo ydob `(,x . ,vars^) `(,a . ,env^) lav))]
    [(fresh (x ydob)
       (== `(,ydob (,x) adbmal) pxe)
       (== `(closure ,x ,ydob ,vars ,env) lav)
       (symbolo x)
       (absento 'adbmal env))]))
```

Listing 4. The `fo-lavo` evaluation relation with a split environment and the `laveo` relation front end

Unification can immediately separate the first element of a list from the rest, however, matching against the last element of a list requires arbitrary recursion. To this end we implemented the help relation `tsil-reporpo` in Listing 3, a mirrored "proper list" relation that describes reversed proper lists.

That the two languages have the same semantics (they are isomorphic) is inconsequential in the following. It is important, however, that we have relational interpreters for two languages with overlapping syntaxes. More precisely still, for these results we need programs in the language of one interpreter to produce, as data, certain classes of programs in the language of the second interpreter.

## 3 PALINDROME PROGRAMS

*Polyglots* are programs valid in more than one language. In Listing 5, we query for programs that are well-formed in both the language of the standard relational interpreter and also Listing 4's interpreter's mirrored language. We query for a single program `q` that evaluates in both interpreters. We do not require that the two values be equal; the problem statement only demands that the programs have a value under each interpretation.

For the special case of these two relational interpreters, whose languages are mirror images of each other, these polyglot programs are also palindromes. The programs are not literally, character for character palindromes. Instead, they are palindromic in that their syntax trees are mirror images. Racket's default printer slightly

```
> (run 3 (q) (fresh (a b) (evalo q a) (laveo q b)))
'('etouq
  ((λ (_0) adbmal) (sym _0))
  ((λ (adbmal) adbmal) (λ (λ) adbmal)))
```

Listing 5. "Mirrored programs" that evaluate in both languages

```
> (run 1 (q) (fresh (p) (laveo p q) (evalo q p)))
'((((λ (_0) (list (list _0 (list 'quote _0)) 'etouq))
    '(λ (_0) (list (list _0 (list 'quote _0)) 'etouq)))
   (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
   (sym _0)))
```

Listing 6. A relational query for the first entry of a 2-cycle quine relay

obscures this, since it uses abbreviations for **`quote`** and **`lambda`**. We could, however, overcome this imbalance by installing a printer that either removes these abbreviations or provides analogous abbreviations for their mirrored counterparts.

## 4 QUINE RELAYS

Quine relays (or "ouroboros programs") are $n$-cycles of programs where each program $p_i$ in the cycle evaluates to the program $p_{i+1 \pmod n}$. To preclude trivial quine relays, the problem statement usually requires a cycle of *distinct* programs, and further that each program be written in a different programming language.

In Listing 6 we query miniKanren using the same two relational interpreters to synthesize a program `p` that in the language of the first interpreter evaluates to a program `q` in the language of the second interpreter, which itself then evaluates to `p`. The calls to the two different interpreters guarantee that the resulting pairs of programs will be in different languages; we do not have to overtly stipulate that the languages be different or write predicates to recognize each language's programs. If we had instead merged the two evaluation relations into one, we would then need to recursively specify the programs of each language to properly state our query. Further, because the only overlap in the two interpreters' languages are quoted (`etouq`'d) expressions, we do not need to explicitly stipulate that the programs be different with a disequality constraint.

miniKanren's first solution is not especially revealing. Rather than the self-quoting behavior often seen in miniKanren twines (e.g., Listing 2), miniKanren has generated a self-`etouq`ing expression. The quine cycles example does demonstrate an interesting facet of miniKanren's behavior. In queries for twines, the order of the calls to the interpreter make little to no difference, since both calls are to the same interpreter. For quine cycles though, reordering the calls can make a significant difference. miniKanren takes significantly longer to answer Listing 7's version of this query. Further, the answer largely reverses the roles of the two interpreters. The query's result is a self-quoting (that is, with the real **`quote`** of the standard interpreter) program, largely written in the language of the mirrored interpreter of Section 2. The result program is similar to, but not precisely, a mirrored version of the more traditional-looking result program from Listing 6. Reversing the order of the two evaluations biases miniKanren's search toward the already-partially successful branches of quoted programs when evaluating in `laveo`. `tsil` expressions cost more to evaluate than `list` expressions, and we believe this difference explains some of the overall performance disparity.

```
> (run 1 (q) (fresh (p) (evalo q p) (laveo p q)))
'((('(((('etouq
       (_0
        (((_0 (etouq etouq) tsil) (_1 etouq) tsil)
         ((_1) etouq)
         (adbmal etouq)
         tsil)
        tsil)
       tsil)
      (_0)
      adbmal)
     (((((('etouq
          (_0
           (((_0 (etouq etouq) tsil) (_1 etouq) tsil)
            ((_1) etouq)
            (adbmal etouq)
            tsil)
           tsil)
          tsil)
         (_0)
         adbmal)
        etouq)
       _1)
      (_1)
      adbmal))
   (=/= ((_0 closure)) ((_0 etouq)) ((_0 tsil)) ((_1 closure)) ((_1 etouq)))
   (sym _0 _1)))
```

Listing 7. A second relational query for a 2-cycle quine relay, reordering the evaluations

## 5 BOOTSTRAPPING, RECOVERING QUINES

David Madore, author of the unlambda language, on his *Quines* page [18] inadvertently describes yet another application of synthesis by relational interpreter. Madore describes how, by executing a near-quine program whose "source code" portion is damaged, mangled, or otherwise modified, we might recover the original quine. Some quines are then, to a degree, self-healing programs: for certain damage, executing the damaged program *bootstraps* the quine, and recovers the original, undamaged source. In Listings 8 to 11 we query for three such near-quine "damaged-code" programs that, when invoked, produce a quine. The query stipulates via `(absento p s)`, an additional constraint, that we will exclude somewhat trivial cases like nullary damage (we can view quines as trivially, vacuously bootstrapping themselves) or programs simply containing the quine itself as data. Without these restrictions, in fact, the bootstrap programs are precisely the initial transients of oscillating 1-cycle quines with a single initial transient.

We break up the list of results over Listings 9 to 11. The first program miniKanren returns is $\alpha$-equivalent to a quine. The bootstrap program differs from the quine it produces only in the names of code portion's lambda expressions' variables. miniKanren constrains the variables `_0` and `_1` in the answer with a disequality, maintaining the query's requirement that `p` is absent from `s`. These subsequent results are also interesting.

```
> (run 3 (q)
    (fresh (p r s)
      (== q `(,s ,r))
      (absento p s)
      (evalo `(,p ,r) `(,p ,r))
      (evalo `(,s ,r) `(,p ,r))))
```

Listing 8. Query bootstrapping quines from "damaged" source code.

```
'((((λ (_0) (list _0 (list 'quote _0)))
    '(λ (_1) (list _1 (list 'quote _1))))
   (=/= ((_0 _1)) ((_0 closure)) ((_0 list)) ((_0 quote))
        ((_1 closure)) ((_1 list)) ((_1 quote)))
   (sym _0 _1))
```

Listing 9. First result for the query in Listing 8

```
((((λ (_0) (λ (_1) (list _1 (list 'quote _1)))) '_2)
  '(λ (_3) (list _3 (list 'quote _3))))
 (=/= ((_0 list)) ((_0 quote)) ((_0 λ))
      ((_1 _3)) ((_1 closure)) ((_1 list)) ((_1 quote))
      ((_3 closure)) ((_3 list)) ((_3 quote)))
 (sym _0 _1 _3)
 (absento ((λ (_3) (list _3 (list 'quote _3))) _2) (closure _2)))
```

Listing 10. Second result for the query in Listing 8

For this second result, two constraints on the answer bear particular mention. First, `_1` must not equal `_3`, since by the query's `absento` constraint the resulting quine's operator, `(λ (_3) (list _3 (list 'quote _3)))`, must not be present in the operator of the bootstrap program. That same `absento` induces the restriction on `_2` in the presented answer.

We choose to include Listing 11 and the third result because it showcases a good example of miniKanren generating a non-trivial bootstrap program. This bootstrap program, when executed, constructs the subsequent quine's operator **`lambda`** expression from quoted pieces. Taken together, these results may appear to the reader as "strategically" damaged source code, reverse-engineered. Bootstrapping from damaged quine code may feel more realistic in a lower-level language closer to the machine's representation. The specification we describe via our queries, however, would not change, and from miniKanren's complete search we know every answer occurs at some point in the sequence of results.

## 6 QUASI-QUINES AND QUASI-QUINE CYCLES

The query in Listing 12 asks not for quines, but instead a sub-expression `q` that, when applied to itself, returns itself as the result. This question arose while we were experimenting with a relational interpreter. We invite the reader to stop a moment and consider—does something like that exist? What would one look like? Absent the relational interpreter approach, how would you construct one?

```
(((λ (_0)
    (list (list 'λ '(_1) '(list _1 (list 'quote _1))) (list 'quote _0)))
  '(λ (_1) (list _1 (list 'quote _1))))
 (=/= ((_0 closure)) ((_0 list)) ((_0 quote))
      ((_1 closure)) ((_1 list)) ((_1 quote)))
 (sym _0 _1)))
```

Listing 11. Third result for the query in Listing 8

```
> (run 2 (q) (evalo `(,q ,q) q))
```

Listing 12. Query for trivial and non-trivial quasi-quines `q`

```
'(quote
  (((λ (_0) (λ (_1) (list _0 (list 'quote _0))))
    '(λ (_0) (λ (_1) (list _0 (list 'quote _0)))))
   (=/= ((_0 _1)) ((_0 closure)) ((_0 list)) ((_0 quote)) ((_0 λ))
        ((_1 closure)) ((_1 list)) ((_1 quote)))
   (sym _0 _1)))
```

Listing 13. Query for non-trivial quasi-quines `q`

```
(run 1 (a b c)
  (fresh (t)
    (== t `((,a ,b) (,b ,c) (,c ,a)))
    (=/= a b) (=/= b c) (=/= c a)
    (evalo `(,a ,b) c)
    (evalo `(,b ,c) a)
    (evalo `(,c ,a) b)))
```

Listing 14. Query for the triplet of expressions `a`, `b`, `c` that form a stuttered quine 3-cycle

miniKanren quickly finds a first answer: **`quote`**. In the language of the interpreter, as in Scheme, `(quote quote)` evaluates to **`quote`**. The first result of this query is trivially satisfiable. We present the first and second answers in Listing 13, where for readability we expand the first **`quote`** result from Racket's printer. The second answer is non-trivial, and looks closer to the usual quine answers, but parameterized additionally in both code and data by a "don't care" variable. The disequality constraint between `_0` and `_1` differs from either of the two superficially similar examples we discussed in Listings 9 and 10. Here, the disequality constraint is necessary to prevent shadowing, lest the "don't care" variable actually matter to the result.

Now suppose we wish to construct a triplet of distinct expressions `a`, `b`, and `c` so that the expressions `(a b)`, `(b c)`, and `(c a)` evaluate respectively to `c`, `a`, and `b`. Such programs are not the thrines discussed in Section 1. As far as we know, there is no canonical name for the program cycles with this property; we've nicknamed them "quasi-quine cycles", or "stuttered-quine cycles". We find even our clearest prose specifications for such programs wordier and more opaque than simply reading this program triplet's specification directly from the miniKanren query itself.

```
'((((λ (_0)
      '(λ (_1)
         (list
          'λ
          '(_2)
          (list
           'quote
           (list 'λ '(_0) (list 'quote (_1 '_3)))))))
    (λ (_2)
      '(λ (_0)
         '(λ (_1)
            (list
             'λ
             '(_2)
             (list
              'quote
              (list 'λ '(_0) (list 'quote (_1 '_3))))))))
    (λ (_1)
      (list
       'λ
       '(_2)
       (list 'quote (list 'λ '(_0) (list 'quote (_1 '_3)))))))
   (=/= ((_0 closure)) ((_0 quote)) ((_1 closure)) ((_1 list))
        ((_1 quote)) ((_2 closure)) ((_2 quote)))
   (sym _0 _1 _2)
   (absento (closure _3))))
```

Listing 15. Result of query in Listing 14

Listings 14 and 15 contain this augmented query and the first program triplet that miniKanren returns. The answers, again, are non-trivial. The authors were not certain what to expect when initially formulating these queries. We had not predicted the trivial answers. After first discovering the trivial results, we still were not sure that miniKanren could produce more interesting answers. One of the facets that makes miniKanren a worthwhile synthesis platform is the ease with which we can formulate specifications as executable miniKanren queries and run experiments.

## 7 RELATED QUINE-LIKE SPECIFICATIONS AND MINIKANREN SYNTHESIS TASKS

Self-reference is a broad, general phenomenon well-studied across many different contexts [28]. While of wide theoretical interest, the contortions required to express self-reference in many systems make it generally unusable as a practical technique. Self-reference is significantly easier to express, though, in systems with quotation. Little [17] and Polonsky [26] formalize the properties of quotation and syntactic self-reference and suggests directions for constructing languages with facilities for practically useful self-reference. Even just defining a class of non-trivial self-replicating systems has proved somewhat difficult, and recent efforts point to a non-binary sliding scale [1]. Quines are generally regarded among the more trivial kinds of self-replication; the examples we present here are of similar complexity.

Myhill [23] describes the general theory of self-reproduction, and mentions producing identical offspring as the simplest variety. His discussion includes varieties of automata that take no input, and also posits infinitely-descending chains of automata-generating automata. Myhill also suggests self-improving automata, and even automata that are the mirror-image of their parent. Case [9] studies machine lineages and the problem of deciding if a particular machine starts an infinite sequence, and if so, whether that sequence is periodic. "Periodic chains" is another description for those bootstrapped n-cycles. Kraus's [15] recently rediscovered MS thesis is something of a bridge between this earlier work and the application to computer virology. In it he also describes a wide variety of self-replicating adjacent programs, several of which we have synthesized here. Where quines do appear in security and computer virology, it's frequently in some deep and interesting places, such as in Thompson's [30] "Reflections on Trusting Trust." Gratzer and Naccache [12] offer a more recent example, showing how quines can help secure software systems, rather than undermine them.

Program synthesis is a broad research area with interdisciplinary connections to programming languages, machine learning, artificial intelligence and regular conferences in its own right. Unsurprisingly it also has deep connections to logic and specification. In its early form, synthesis research dates back to Church [10] in 1957. Manna and Waldinger [19] developed an influential model for synthesizing from a logical statement of the specification programs guaranteed-correct with respect to that specification. More recently, the *syntax-guided synthesis* [2] approach has gained traction. In syntax-guided synthesis, the user also provides partial template(s) as additional synthesis aids. Kanren-style relational interpreters take advantage of partial program templates. Their problem space, however, is much more general than the relatively straightforward synthesis problems we address here.

Byrd et al. [7] solve a problem nearly the inverse of polyglots. Instead, they evaluate the same concrete syntax under distinct semantics, in their case lexical and dynamic scope. These are orthogonal directions, and we could certainly mix and match distinct semantics and different (but not disjoint) syntaxes for even more interesting queries.

## 8 CONCLUSIONS

We have presented several instances of more general classes of well-known problems, sometimes presented as challenges or contests. Through surveying these myriad exemplary problems and their uniform solutions, we have demonstrated the power, simplicity, and convenience of miniKanren as a language for solving these quine and quine-like synthesis problems. We have used relational programming techniques to synthesize curious programs across multiple languages, including polyglot "palindrome programs", quine relays, and some stuttered quine-like programs. Our straightforward, declarative solutions to instances of so many diverse pre-existing problems should evidence miniKanren expressiveness. That these aren't artificially created problems, but largely known and pre-existing challenges further buttresses this claim.

These problems can play the role of "litmus tests" to assess, express, and checkpoint the power of relational language design and implementation. We do not claim the utmost immediate application. Applications, however, often present themselves when there is foundational research to apply. Research should not limited to that which can be put to immediate practical use. We think it also important that we consider here concrete synthesis problems in a broader context of computability theory. Distillation research [24] is itself a meritorious aim, and to the degree we fulfill this role, that too is a research contribution.

Although not as well known as quines themselves, producing polyglot programs and quine relays are also time-worn coding challenges. Challenges like these are usually solved by hand. Other stipulations are sometimes added in such contests for extra complexity, e.g., "the shortest such program that . . . " or "the longest cycle of such programs that . . . ." Margenstern and Rogozhin [20], for instance, describe significantly shrinking self-describing Turing machines.

Multi-quine synthesis would be an interesting challenge for the future. A "multi-quine" is a set of programs that print their own source code, except that on particular, given inputs, they instead produce another program in the set. Madore describes virology applications for these results, and even demonstrates an impressive encrypted multi-quine proof-of-concept for this behavior. These would require at least some portions of Byrd et al.'s more powerful interpreters, as by definition multi-quines requires variable-arity functions. Presumably this would also necessitate their techniques for taming the explosion of the search space. Along these same lines we suggest self-analyzing or self-describing programs as other synthesis targets. These programs are not *necessarily* self-referential, since, of course, there are many other programs that begin with the same character, have the same length, or what have you. Nor are they strictly speaking quines, since many varieties of these problems require input. They too might suggest other future applications.

The answers we synthesized are simple compared with record-holding solutions. The codegolf community is a repository for such challenges and impressive solutions. Endoh [11] constructed a 128-language quine relay. Also impressive are long-lasting group efforts producing 280+-language polyglot programs.[2] The authors do not anticipate dethroning these records with our techniques anytime soon. We remind the reader, however, that these records are not mechanically synthesized in seconds or minutes, but instead laboriously hand-crafted; in the latter case constructed communally by scores over years. Bratley and Millo [6] in 1972 anticipate the enduring fascination for such exercises, explaining

> there is something about this particular problem which excites keen programmers, so that research in self-reproducing program design is now under way for practically every compiler on our system.

We too could not resist the temptation to play.

[2] See the current record and discussion at codegolf.stackexchange.com/questions/102370.

```
(define-relation
  (plop acc whole res data)
  (conde
   ((fresh
     (s)
     (== data `(,s))
     (== acc `(,s))
     (conde
      ((fresh
        (a d)
        (== s `(,a unquote d))
        (=/= d '())
        (fresh (m) (== res `(,m)) (plop s whole m s))))
       ((== 'q s) (== res `(q ',whole))))))
    ((fresh
      (a d)
      (=/= d '())
      (== `(,a unquote d) data)
      (fresh
       (p)
       (== res `(,a unquote p))
       (fresh (q) (== acc `(,a unquote q)) (plop q whole p d)))))))
 (run*
  (q)
  (fresh
   (x)
   (plop
    x
    x
    q
    '((define-relation
       (plop acc whole res data)
       (conde
        ((fresh
          (s)
          (== data `(,s))
          (== acc `(,s))
          (conde
           ((fresh
             (a d)
             (== s `(,a unquote d))
             (=/= d '())
             (fresh (m) (== res `(,m)) (plop s whole m s))))
            ((== 'q s) (== res `(q ',whole))))))
         ((fresh
           (a d)
           (=/= d '())
           (== `(,a unquote d) data)
           (fresh
            (p)
            (== res `(,a unquote p))
            (fresh (q) (== acc `(,a unquote q)) (plop q whole p d)))))))
      (run* (q) (fresh (x) (plop x x q)))))))
```

## REFERENCES


[1] Bryant Adams and Hod Lipson. 2003. A universal framework for self-replication. In *Advances in Artificial Life, 7th European Conference, ECAL 2003, Dortmund, Germany, September 14-17, 2003, Proceedings* (Lecture Notes in Computer Science). Wolfgang Banzhaf et al., (Eds.) Vol. 2801. Springer, 1–9. DOI: 10.1007/978-3-540-39432-7_1. https://doi.org/10.1007/978-3-540-39432-7_1.

[2] Rajeev Alur et al. 2013. Syntax-guided synthesis. In *Proceedings of Formal Methods in Computer Aided Design, FMCAD 2013*. Barbara Jobstmann and Sandip Ray, (Eds.) IEEE, 1–17.

[3] Daniel Bilar and Eric Filiol. 2009. Editors/translators foreword. Daniel Bilar and Eric Filiol, (Eds.) (2009). DOI: 10.1007/s11416-008-0116-y. link.springer.com/journal/11416/5/1.

[4] Daniel Bilar and Eric Filiol, (Eds.) 2009. Journal in Computer Virology 5, 1 (2009): *On Self-Reproducing Computer Programs*. link.springer.com/journal/11416/5/1.

[5] Gregory W. Bond. 2005. Software as art. *Commun. ACM*, 48, 8, (08/2005), 118–124. DOI: 10.1145/1076211.1076215. https://doi.org/10.1145/1076211.1076215.

[6] Paul Bratley and Jean Millo. 1972. Computer recreations: self-reproducing programs. *Software: Practice and Experience*, 2, 4, 397–400. DOI: 10.1002/spe.4380020411. onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380020411.

[7] 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, (08/2017), 8:1–8:26. DOI: 10.1145/3110252. http://doi.acm.org/10.1145/3110252.

[8] 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, Copenhagen, Denmark, 8–29. DOI: 10.1145/2661103.2661105. http://doi.acm.org/10.1145/2661103.2661105.

[9] John Case. 1974. Periodicity in generations of automata. *Mathematical systems theory*, 8, 1, 15–32. DOI: 10.1007/BF01761704. https://doi.org/10.1007/BF01761704.

[10] Alonzo Church. 1960. Application of recursive arithmetic to the problem of circuit synthesis. In *Summaries of talks presented at the Summer Institute for Symbolic Logic—Cornell University, 1957*. Vol. 28. Communications Research Division, Institute for Defense Analyses, Princeton, New Jersey, 3–50.

[11] Yusuke Endoh. 2020. 128-language uroboros quine relay. https://github.com/mame/quine-relay.

[12] Vanessa Gratzer and David Naccache. 2006. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Advances in Cryptology - EUROCRYPT 2006*. Serge Vaudenay, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 48–58.

[13] Jason Hemann. 2020. *Constraint microKanren in the CLP Scheme*. Ph.D. Dissertation. Indiana University. http://hdl.handle.net/2022/25183.

[14] Douglas Hofstadter. 1979. *Gödel, Escher, Bach : an eternal golden braid*. Basic Books, New York.

[15] Jürgen Kraus. 2009. On self-reproducing computer programs. Trans. by Daniel Bilar and Eric Filiol. *Journal in Computer Virology*, 5, 1, 9–87. *On Self-Reproducing Computer Programs*. Daniel Bilar and Eric Filiol, (Eds.) DOI: 10.1007/s11416-008-0115-z. link.springer.com/journal/11416/5/1.

[16] C. Y. Lee. 1963. A Turing machine which prints its own code script. In *Proc. of the Symposium on Mathematical Theory of Automata*, 155–164.

[17] C. M. W. Little. 2002. Parametrized quotation and self-reference. *Electronic Notes in Theoretical Computer Science*, 74, 69–88. MFCSIT 2002, Second Irish Conf. on the Mathematical Foundations of Computer Science and Information Technology. DOI: 10.1016/S1571-0661(04)80766-7. doi.org/10.1016/S1571-0661(04)80766-7.

[18] David Madore. 2020. Quines (self-replicating programs). Last accessed 5 May 2020. madore.org/~david/computers/quine.html.

[19] Zohar Manna and Richard Waldinger. 1975. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6, 2, 175–208. DOI: https://doi.org/10.1016/0004-3702(75)90008-9. http://www.sciencedirect.com/science/article/pii/0004370275900089.

[20] Maurice Margenstern and Yurii Rogozhin. 2002. Self-describing Turing machines. *Fundamenta Informaticae*, 50, 3-4, 285–303.

[21] John McCarthy. 1978. A micro-manual for LISP -not the whole truth. *ACM Sigplan Notices*, 13, 8, 215–216.

[22] Yiannis N. Moschovakis. 2010. Kleene's amazing second recursion theorem. *The Bulletin of Symbolic Logic*, 16, 2, 189–239. DOI: 10.2178/bsl/1286889124.

[23] John Myhill. 1964. The abstract theory of self-reproduction. *Views on general systems theory*, 106–118.

[24] Chris Olah and Shan Carter. 2017. Research debt. *Distill*. https://distill.pub/2017/research-debt. DOI: 10.23915/distill.00005.

[25] Kostas Pagiamtzis. 2010. Oscillating (iterating) quine with initial transient. Last accessed 5 May 2020. pagiamtzis.com/articles/iterating-quine-with-transient/.

[26] Andrew Polonsky. 2011. Axiomatizing the Quote. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings* (LIPIcs). Marc Bezem, (Ed.) Vol. 12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 458–469. DOI: 10.4230/LIPIcs.CSL.2011.458. https://doi.org/10.4230/LIPIcs.CSL.2011.458.

[27] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2019. Certified semantics for minikanren. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop* number TR-02-19. Cambridge, Massachusetts, 80–98. dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.

[28] Raymond M Smullyan. 1994. *Diagonalization and Self-Reference*. Clarendon Press, Oxford New York.

[29] James W Thatcher. 1963. The construction of a self-describing Turing machine. In *Proc. of the Symposium on Mathematical Theory of Automata*, 165–171.

[30] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM*, 27, 8, (08/1984), 761–763. DOI: 10.1145/358198.358210. https://doi.org/10.1145/358198.358210.

[31] Edward Z Yang. 2010. Adventures in three monads. *The Monad. Reader*, 15, (01/2010), 11–37. Brent Yorgey, (Ed.) themonadreader.files.wordpress.com/2010/01/issue15.pdf.

[32] Ann Yasuhara. 1971. *Recursive Function Theory and Logic*. Academic Press, New York.

# A Relational Interpreter for Synthesizing JavaScript

ARTEM CHIRKOV, University of Toronto Mississauga, Canada
GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada

We introduce a miniKanren relational interpreter for a subset of JavaScript, capable of synthesizing imperative, S-expression JavaScript code to solve small problems that even human programmers might find tricky. We write a relational parser that parses S-expression JavaScript to an intermediate language called LambdaJS, and a relational interpreter for LambdaJS. We show that program synthesis is feasible through the composition of these two disjoint relations for parsing and evaluation. Finally, we discuss three somewhat surprising performance characteristics of composing these two relations.

CCS Concepts: • **Software and its engineering → Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: miniKanren, logic programming, relational programming, program synthesis, JavaScript

## 1 INTRODUCTION

Program synthesis is an important problem, where even a partial solution can change how programmers interact with machines [7]. The miniKanren community has been interested in program synthesis for many years, and has used relational interpreters to solve synthesis tasks [5, 6, 10, 13].

JavaScript is famously unpredictable [3]. However, JavaScript's ubiquity makes it an important language to target for program synthesis. To that end, this paper explores the feasibility of both writing a relational JavaScript interpreter, and using such an interpreter to solve small synthesis problems. To remind readers of the quirkiness of JavaScript, we invite you to try the puzzle in Figure 1, inspired by a similar example in Guha et al. [8].

```
(function(x) {
  return (function() {
    if (false) {
      ____________________
    }
    return x;})();
})(42);
```

Fig. 1. A JavaScript puzzle: fill in the blank so that the entire expression evaluates to 'undefined' rather than '42'.

Authors' addresses: Artem Chirkov, University of Toronto Mississauga, Canada, artem.chirkov@mail.utoronto.ca; Gregory Rosenblatt, University of Alabama at Birmingham, USA, gregr@uab.edu; Matthew Might, University of Alabama at Birmingham, USA, might@uab.edu; Lisa Zhang, University of Toronto Mississauga, Canada, lczhang@cs.toronto.edu.


miniKanren.org/workshop/2021/8-ART2

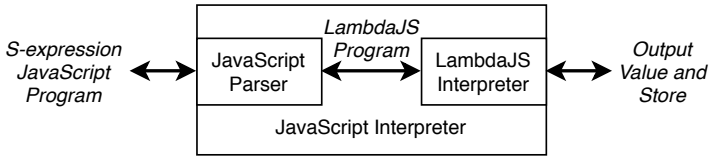


Fig. 2. Relational JavaScript Interpreter Design

Our main contribution is a proof-of-concept relational interpreter for a subset of JavaScript[1]. We build our JavaScript interpreter by combining two components:

(1) A relational interpreter for an intermediate language called LambdaJS [8]. LambdaJS describes the core semantics of JavaScript. We describe this interpreter in Section 2.
(2) A relational parser that desugars an S-expression representation of a JavaScript program into a LambdaJS program. We describe the S-expression representation and the desugaring process in Section 3.

The LambdaJS interpreter is capable of synthesizing LambdaJS code, but in order to synthesize actual S-expression JavaScript, we need to also use the relational parser from JavaScript to LambdaJS. When we combine the two relations, we have a relational JavaScript interpreter capable of synthesizing S-expression JavaScript code to solve synthesis problems like the one in Figure 1.

In addition to presenting synthesis experiments, we describe three performance characteristics of composing the parser and interpreter relations that are somewhat surprising. These observations may be helpful for miniKanren users even if they are not interested in JavaScript.

## 2 A RELATIONAL LAMBDAJS INTERPRETER

JavaScript syntax is complex, and its semantics is quirky and inconsistently defined [3]. Instead of building a relational interpreter directly, we first translate JavaScript syntax to a simpler, intermediate language called LambdaJS [8]. LambdaJS was initially developed in 2010 based on ECMAScript 3. Although the ECMAScript standards are regularly updated, we work with an older subset of JavaScript and we do not support the latest features.

### 2.1 LambdaJS

LambdaJS is a simple language that expresses the fundamental features of JavaScript. We can desugar any JavaScript program into a LambdaJS program. LambdaJS consists of these core language features:

- atomic values (number, string, boolean, undefined, null)
- immutable, statically-scoped variables, let expressions and functions
- mutable references (allocate, dereference, assign, delete)
- objects with prototype inheritance
- basic control flow constructs (begin, if, while)
- control effects (throw, break, try, catch, finally, label)

Other JavaScript syntax, such as `for` loops and `switch` statements, can be desugared into LambdaJS. Interested readers can find the details of the LambdaJS semantics in Guha et al. [8]. LambdaJS programs tend to be verbose, and are not meant to be written by hand. Instead, LambdaJS is intended to simplify tasks related to reasoning about the semantics of JavaScript, such as writing an interpreter.

[1] https://github.com/Artish357/RelateJS

## 2.2 LambdaJS Interpreter Implementation

Our implementation of a LambdaJS interpreter roughly follows Guha et al. [8], which defines a small-step operational semantics for LambdaJS. We implement a big-step interpreter consistent with these semantics. The rest of this section describes our implementation choices and details, and notes where our implementation differs from Guha et al. [8] .

*2.2.1 Atomic values.* Our implementation of LambdaJS includes these atomic types: `null`, `undefined`, `number`, `string`, and `boolean`. To differentiate these types, we use tagging: every value is represented by a list, with the first element being a symbol describing its type. For simplicity, we restrict numbers to only natural numbers, and represent them using the relational little-endian numerals from Kiselyov et al. [9]. We represent a string as a list of such numbers, where each number encodes a character.

*2.2.2 Immutable variables, let expressions, and functions.* LambdaJS is statically scoped, and variables are immutable. Once bound, a variable is never re-bound, but may be shadowed by a new binding. In our implementation, these variable bindings are stored in an immutable environment represented by an association list (list of key-value pairs). New variable bindings are created when functions are called and by `let` expressions.

*2.2.3 Mutable References.* Like JavaScript, LambdaJS is an imperative language that allows mutation. However, LambdaJS makes mutable references explicit, and assigns these references in a separate *store*. The store contains a list of values that are accessed by position, where a position is represented as a list-encoded Peano numeral. Our LambdaJS interpreter implements the following memory operations:

- `allocate`: put a value into the store at a new position, and increment the position counter.
- `dereference`: retrieve a value from the store.
- `assign`: put a value into the store at an existing position.

Memory operations are side-effects that are threaded through the evaluation. Subsequent evaluations can see the effects of previous evaluations. In particular, the interpreter has four additional parameters that do not appear in relational interpreters for functional languages: an initial value for the mutable store, a final value for the mutable store, an initial value for the mutable store index counter, a final value for the mutable store index counter.

*2.2.4 Functions and Objects.* LambdaJS functions are distinct from JavaScript functions. In JavaScript, a function is an object, so that statements like these are valid: `(function(x){ return x; })["why"] = 42`. In contrast, LambdaJS functions are simpler, and evaluate to a closure containing the environment at the time the function was defined.

LambdaJS objects are also distinct from JavaScript objects. In JavaScript, we manipulate objects indirectly through memory references (explained in detail in Section 3.2.2). LambdaJS objects contain accessible fields, which we implement using an association list. LambdaJS objects are immutable, and programs can manipulate objects functionally using the following operations:

- `get`: retrieve the value of an object field;
- `create/update`: create an object field and assign its value, or update the field if it already exists; return a new object with the new/updated fields;
- `delete`: remove an object field, and return a new object with the field removed.

The LambdaJS object semantics described in Guha et al. [8] include prototype inheritance. However, prototype inheritance can instead be implemented by the desugaring stage. For simplicity, our proof-of-concept interpreter does not support prototype inheritance.

*2.2.5 Basic Control Flow.* Our implementation of LambdaJS supports three basic control flow operations:

- `begin`: A `begin` expression sequences two subexpressions. Its value is the value of the second subexpression, although the first could have side-effects (alter the mutable store, throw an exception).
- `if`: An `if` expression consists of a condition, a `then` branch, and an `else` branch.
- `while`: A `while` loop expression consists of a condition and a body. A terminating `while` loop always eventually evaluates to `undefined`, although the body expression could have side-effects.

Other JavaScript control flow operations, such as `for` loops, can be expressed in terms of these operations.

*2.2.6 Handling control effects.* A control effect stops normal control flow, and jumps to an exception handler (if one exists). Control effects include `throw`, `break` and `return`. Our handling of control flow deviates slightly from Guha et al. [8]: we consolidate `throw` and `break` into a single `break` effect, and we combine `try`, `catch`, `finally`, and `labels` into a combined construct that catches a `break` with a particular label, binds the value carried by the `break` to a variable, and evaluates a handler expression.

In order to properly handle exceptions, we need a way to interrupt normal control flow. We check the result of an evaluation for a `break` value. If there is a `break` value, we stop the computation and return to the correct exception handler (if one exists). Otherwise, we continue normally. In our implementation, we model this behaviour using a macro called `effect-propagateo`.

## 3 A RELATIONAL JAVASCRIPT TO LAMBDAJS PARSER

Our parser desugars an S-expression JavaScript statement into a LambdaJS expression. The parser needs to be relational so that we can use the desugarer as a "sugarer" that generates S-expression JavaScript from the LambdaJS equivalent.

Desugaring or parsing is a non-trivial process since much of the complexity of JavaScript is encoded in the parser. For example, JavaScript is a statement-based language, and LambdaJS is an expression-based language. JavaScript functions are objects, whereas LambdaJS functions are not. JavaScript variables and objects have implicit mutable reference semantics, and LambdaJS makes mutable references and their operations explicit. JavaScript also has richer syntax for control flow operators, some of which we support in our work.

In Section 3.1, we describe the S-expression syntax that we use to represent a subset of JavaScript. Then, in Section 3.2, we describe the design choices we made to transform S-expression JavaScript into LambdaJS.

### 3.1 S-expression JavaScript syntax

The grammar for the S-expression JavaScript syntax is shown in Figure 3. In particular, the subset of JavaScript that we support includes:

- atomic values (same as in Section 2.2.1)
- variable allocation/assignment/access
- object creation, object field access/update
- function definition, function call
- if statements
- for/while loops
- try/catch/finally

Additionally, Figure 4 contains a side-by-side comparison of JavaScript syntax, and the corresponding S-expression notation.

Since relational arithmetic numbers and strings are difficult for humans to read, we add a layer to our parser that transforms human-readable literals into the relational number and string format described in Section 2.2.1.

```
Symbol         = ... infinite set of symbols ...
Number         = ... infinite set of numbers ...
String         = ... infinite set of strings ...
Operation      = + | - | * | / | < | string+ | string< | char->nat | nat->char
VarDeclaration = Symbol | (Symbol E)
L              = Number | String | #t | #f | (undefined) | (null)
E              = L | Symbol | (object (String E) ...) | (op Operation E ...)
               | (@ E E) | (:= Symbol E) | (:= (@ E E) E) | (call E E ...)
               | (function (Symbol ...) E ...)
S              = E | (var VarDeclaration ...) | (begin S ...) | (if E S S)
               | (for (S E E) S ...) | (return E) | (throw E) | (break)
               | (try S catch Symbol S) | (try S finally S)
               | (try S catch Symbol S finally S) | (while E S ...)
```

Fig. 3. S-expression JavaScript grammar

## 3.2 Relational Parser Implementation

The relational parser desugars an S-expression JavaScript statement into a LambdaJS expression. We make a distinction between JavaScript *statements* and *expressions*. At the top level, our parser expects a JavaScript program to be a single statement.

The rest of this section describes choices we made in converting a S-expression JavaScript statement into a LambdaJS expression.

*3.2.1 Variables.* Unlike in LambdaJS, JavaScript variables are mutable. Therefore, the parser desugars a JavaScript variable declaration by creating a LambdaJS immutable variable bound to an allocated mutable reference. By assigning the value of the underlying reference, we can express mutation through immutable variables. In order to access the values in these underlying references, the parser dereferences all variables before use.

The parser also implements JavaScript hoisting (or lifting) semantics, where variable declarations are moved to the top of their scope. This implies that variables defined using `var` are visible even before reaching the declaration location. Hoisting is performed in function definitions, and at the top level.

*3.2.2 Objects.* A JavaScript object does not translate directly into a single LambdaJS object. Instead, it is represented as a LambdaJS object containing two more objects stored in the fields "public" and "private". When the JavaScript program accesses or updates an object field, we use the corresponding fields in the "public" LambdaJS object. The "private" object contains hidden fields used to recognize and call functions. (See Section 3.2.3.)

For simplicity, we don't implement JavaScript object prototype inheritance. However, this feature can be implemented in a similar way using the "private" object.

*3.2.3 Function Definitions.* In JavaScript, a function is represented as an object with a special private "call" field. The standard object functionalities, such as field access and assignment, are available for functions.

For simplicity, we don't implement the implicit `this` variable, but we could do so by extending the environment when a function is called.

*3.2.4 Function Calls.* Since JavaScript functions are objects, during a function call we access the private "call" field of the object to obtain the corresponding LambdaJS function. We apply arguments in the usual way, but

```
// JavaScript Variables
var x;
var y = 3;
var a = y, b = "hi", f;
y;

// JavaScript Objects
{"a": 2, "b": false};
x["a"];
x["a"] = null;

// JavaScript Functions and Built-in
y = y + 1;
f = function(m, n) {
  return false;
};
f(x y);
f["t"] = 3;

// JavaScript if statements
if (y === undefined) {
    x["a"] = 3;
} else {
    x["a"] = 2;
}

// For Loops
for (var i = 0;
     i < 10;
     i = i + 1) {
    f(x, i);
    break;
}


// while loops
while (i < 10) {
    i = i + 1;
}
```

```
;; Corresponding S-expression Notation
(var x)                     ; allocation
(var (y 3))                 ; declaration
(var (a y) (b "hi") f)      ; many assignments
y                           ; use

(object ("a" 2) ("b" #f))   ; object literal
(@ x "a")                   ; field access
(:= (@ x "a") (null))       ; field update

(:= y (op + y 1))           ; built-in operations
(:= f (function (m n)       ; function definition
    (return #f)))

(call f x y)                ; function call
(:= (@ f "t") 3)            ; functions are objects

(if (op === y (undefined)) ; condition
    (:= (@ x "a") 3)        ; then-branch
    (:= (@ x "a") 2))       ; else-branch

(for ((var (i 0))           ; initializer
      (op < i 10)           ; condition
      (:= i (op + i 1)))    ; update
   (call f x i)             ; body
   (break))


(while (op < i 10)          ; condition
   (:= i (op + i 1)))       ; body
```

Fig. 4. A sample of our S-expression JavaScript syntax

before evaluating the body of the function, we re-bind the function parameters to mutable references containing the values of the incoming arguments.

*3.2.5 Built-in Operations.* For simplicity, the syntax for calling built-in operations differs from that of calling functions. The JavaScript built-in operations we support translate directly to LambdaJS built-in operations. We

support numeric operations (`+`, `-`, `*`, `/`, `<`) string operations (`string+`, `string<`, `char->nat`, `nat->char`), `typeof` and strict equality `===`.

*3.2.6 Assignments.* The semantics of the JavaScript assignment operator differs depending on whether we are assigning to a variable or to an object field. A JavaScript variable assignment corresponds to assigning its LambdaJS mutable reference to the value on the right hand side.

A JavaScript object field assignment (like `x["a"] = 2`) is a bit more complicated. We expect the object expression (in this case `x`) to evaluate to an object reference. We dereference it to obtain the "public" LambdaJS object, then use the `create/update` operation to update the object's field. The field update produces a new LambdaJS object, so we assign the reference to the new object.

*3.2.7 Control Flow.* We support JavaScript `if` statements, `while` and `for` loops, and a `begin` statement. Unlike in LambdaJS, these operations are *statements* rather than expressions.

An S-expression JavaScript `begin` statement contains a list of statements to be executed sequentially. It is equivalent to the semantics of the semicolon `;` in pure JavaScript.

Both `if` and `while` statements map almost directly to their LambdaJS counterparts. However, their bodies are statements and not expressions. The JavaScript `for` loop syntax is converted into a LambdaJS `while` loop in the typical way.

*3.2.8 Control Effects.* Effects like JavaScript `return` and `break` are translated into a LambdaJS `throw` that are caught by a LambdaJS `catch` with a hard-coded label (with the label name "return" or "break"). When functions are defined, their bodies are wrapped in a `catch` for the "return" label. Loop bodies are wrapped in a `catch` for the "break" label. The catch handler for the "return" evaluates to the thrown value. The catch handler for the "break" ignores the thrown value.

JavaScript `throw` is also translated to a LambdaJS `throw`. It is caught by a LambdaJS `catch` with a hard-coded label (with the label name "error"). A LambdaJS `catch` is produced when translating JavaScript `try/catch`, `try/catch/finally` and `try/finally`. Additionally, for JavaScript `try/catch` and `try/catch/finally`, the LambdaJS `catch` handler binds a variable to the thrown value.

## 4 JAVASCRIPT SYNTHESIS EXPERIMENTS

We compose the parser and the interpreter so that we can run the combined relation "backwards" to synthesize S-expression JavaScript. All of our experiments are performed on a Macbook Pro with a 2.7 GHz Intel Core i5 processor and 16 GB RAM.

### 4.1 The Puzzle in Figure 1

We use our interpreter to solve the puzzle in Figure 1. We translate the JavaScript puzzle into S-expression syntax, and use a logic variable to represent the blank in the then-branch of the `if` statement. In our S-expression syntax an `if` statement always requires a then-branch and an else-branch, so we add a trivial else-branch.

```
(run 1 (BLANK)
  (fresh (code store)
    (parseo/readable
      `(call (function (x)
               (return (call (function ()
                               (if #f      ; condition
                                   ,BLANK  ; then-branch
                                   #f)     ; else-branch
                               (return x)))))
             42)
      code)
    (evalo code (jundef) store)))
```

Running the above query, we obtain the following answer:

```
(var (x _.0))  ; where _.0 is any symbol
```

This answer is equivalent to the JavaScript code:

```
var x = <symbol>; // where <symbol> is any variable name
```

This answer leverages *hoisting*, the mechanism where variables and function declarations are moved to the top of their scope (Section 3.2.1). Even though this declaration of the variable x is unreachable, it is still moved to the top of the inner function scope.

If we modify the query to return multiple answers, we get similar answers:

```
((var (x _.0)) (sym _.0))  ;; var x = <symbol>;
(var (x #t))               ;; var x = true;
(var (x #f))               ;; var x = false;
(var x)                    ;; var x;
...
```

These answers translate to the following answers to the puzzle in Figure 1:

```
(function(x) {
   return (function() {
     if (false) {
       var x; // OR var x = <value>, OR EVEN var x = x;
     }
     return x;})();
})(42);
```

## 4.2 Synthesize Input Data

In this section we use our interpreter to generate numeric values that satisfy some simple constraints written in JavaScript. We will use the imperative `modulo` function defined below to express modular arithmetic constraints.

```
var modulo = function(n, m) {
  while (< m n) {
     n = n - m;
  }
  return n;
};
```

We can run this program "backwards" to find, for example, inputs `n` where `modulo(n, 3) == 1`. Running such a test yields the answers `(1 4 7 10 13 19 31 37 22 43 ...)`. Notice that the answers are not sorted: an artifact of the relational arithmetic operation.

If we add an additional constraint, so that we look for values `n` where `modulo(n, 3) == 1` and `modulo(n, 5) == 2`, we get these 10 answers in 24 seconds:

```
(7
 37
 22
 (op char->nat "\a")      ;  7
 (op char->nat "%")       ; 37
 (op char->nat "\u0016")  ; 22
 67
 52
 (op char->nat "C")       ; 67
 127
 ...)
```

The results contain not only numeric literals, but also calls to character-to-numeric conversion operations! The `char->nat` calls evaluate to the same values as the integer literals, and appear in the same order. As is usual for miniKanren relational interpreters, it becomes progressively slower to generate more and more answers: generating 20 answers takes 146 seconds.

### 4.3 Synthesizing Imperative Code

We use the interpreter to synthesize the loop body of the definition of `sum_of_range` function below.

```
var sum_of_range = function(n) {
  var total = 0;
  for (var i = 0; i < n; i = i + 1) {
    ____________________
  }
  return total;
};
```

We provide the two examples: `sum_of_range(3) = 3` and `sum_of_range(4) = 6`. Using faster-miniKanren [2], our relational interpreter synthesizes the update rule `var total = total + i;` in about 5 minutes. It might be surprising that the synthesized code uses an unnecessary `var` to perform the assignment, rather than using a simple assignment operation. This behaviour is an artifact of the search order of the parser.

We also try to synthesize the underlined portions of the `sum_of_range` function. We use the same two examples, and provide the rest of the function while replacing one of the underlined code fragments with a logic variable.

```
var sum_of_range = function(n) {
  var total = 0(1);
  for (var i = 0(2); i < n(3); i = i + 1(4)) {
    total = total + i
  }
  return total;
};
```

Using faster-miniKanren [2], our relational interpreter synthesizes:

(1) the entire declaration `var total = 0` in 0.4 seconds;
(2) the entire declaration `var i = 0` in 0.2 seconds;
(3) the stopping condition `i < n` in 0.6 seconds.

The loop increment `i = i + 1` did not synthesize within 10 minutes. This result is not surprising, given how unconstrained the control flow is without the loop increment information.

### 4.4 Synthesizing Recursive Code

To see how well our interpreter can reason about recursive code, we attempt to synthesize each of the underlined portions of a recursive definition of the `fibonacci` function.

```
var fibonacci = function(n) {
  if (n < 2(1)) {
    return n(2);
  } else {
    return fibonacci(n - 1(3)) + fibonacci(n - 2(4));
  }
};
```

We provide the following two examples: `fibonacci(2) = 1` and `fibonacci(5) = 5`. Using faster-miniKanren [2], our relational interpreter synthesizes the underlined portions of the code:

(1) the condition `n < 2` in 32 seconds;
(2) the base case `return n;` in 0.3 seconds;
(3) the argument `n - 1` to the first recursive call in 68 seconds;
(4) the argument `n - 2` to the second recursive call in 10 seconds.

The difference in run time in the last two experiments is due to the order of evaluation. When solving for the first recursive call, the value of the second recursive call is not yet computed. In contrast, when solving for the second recursive call, we have already computed the first.

### 4.5 Synthesizing LambdaJS Code Directly

Surprisingly, direct synthesis of LambdaJS through the interpreter (without the JavaScript parser) is sometimes slower than synthesizing the equivalent JavaScript code.

To demonstrate, we use the relational LambdaJS interpreter to solve the same problems as in Section 4.3 and Section 4.4. We also use the same examples for each synthesis problem, but encode each problem as a LambdaJS program rather than a JavaScript program.

Table 1 shows the synthesis time comparison for each problem. Most of the LambdaJS synthesis tasks do not succeed within 10 minutes, and the two that do so only succeed because we simplify the tasks. These simplifications are necessary because a JavaScript `var` declaration desugars into two LambdaJS operations: a mutable reference allocation and an assignment at a different program location. Synthesizing the full variable initializations requires synthesizing both components, which takes more than 10 minutes. Instead, the table reports the time to synthesize just the assignment.

Table 1. Comparison of JavaScript vs LambdaJS Synthesis Time

| **Problem** | **Blank** | | **JavaScript** | **LambdaJS** |
|---|---|---|---|---|
| `sum_of_range` | Initialization | `var total = 0` | 0.4 seconds | *5 seconds |
| `sum_of_range` | Loop initialization | `var i = 0` | 0.2 seconds | *0.2 seconds |
| `sum_of_range` | Loop stop condition | `i < n` | 0.6 seconds | (>10 min) |
| `sum_of_range` | Loop increment | `i = i + 1` | (>10 min) | (>10 min) |
| `sum_of_range` | Loop body | `total = total + i` | 5 minutes | (>10 min) |
| `fibonacci` | Base case condition | `n < 2` | 32 seconds | (>10 min) |
| `fibonacci` | Base case body | `return n;` | 0.3 seconds | (>10 min) |
| `fibonacci` | First recursive call | `n - 1` | 68 seconds | (>10 min) |
| `fibonacci` | Second recursive call | `n - 2` | 10 seconds | (>10 min) |

## 5 DISCUSSION

The parser-interpreter design of our relational JavaScript interpreter has both advantages and disadvantages. Some of these are straightforward: separating the relations makes the code readable, and combining (or interleaving) the relations would likely make synthesis faster. Through this work, we found a few characteristics of the parser-interpreter design that are not immediately obvious.

(1) **The parser almost entirely constrains the program structure.** The results in Section 4.5 show that synthesizing LambdaJS without the parser is *slower* than synthesizing S-expression JavaScript. This behaviour sounds paradoxical, but really isn't. Not all LambdaJS programs have a JavaScript equivalent, so the parser greatly reduces the search space and improves synthesis performance.

(2) **The ordering of conde branches in the LambdaJS interpreter does not affect JavaScript synthesis performance.** Typically, rearranging the `conde` branches of a relation alters its performance characteristics. However, since the parser almost entirely constrains the program structure, there are few decisions to be made by the LambdaJS interpreter. Rearranging the `conde` branches in the parser *does* change the performance characteristics, as one would expect.

(3) **The parser-interpreter combination is not refutationally complete.** In other words, a search for a nonexistent solution will generally not terminate. The constraint on the output LambdaJS program is only seen after we begin interpreting that LambdaJS program, which only happens after leaving the call to the parser. The parser does not get feedback to constrain its space of possible JavaScript programs. Therefore, there is an infinite number of JavaScript programs to enumerate.

## 6 RELATED WORK

The idea of a relational interpreter written in miniKanren is not new. Byrd et al. [6] implements a small Scheme interpreter capable of synthesizing quines. Byrd et al. [5] extends and optimizes the interpreter. Though published relational interpreters written in miniKanren tend to be for functional languages, Lundquist et al. [10] implemented a relational interpreter that can synthesize x86 assembly.

Beyond the miniKanren community, many researchers are interested in synthesizing imperative programs. Solar-Lezama [12] allows a programmer to specify an imperative program containing holes where expressions should be synthesized, similar to how we can specify a program containing logic variables. Their solving technique focuses on imperative programs, but includes a limited ability to reason about recursive computations by inlining them a fixed number of times. Blanc et al. [4] similarly allows a programmer to specify a program containing holes, supporting synthesis of a subset of Scala including support for mutation and loops. Their solving technique is able to reason about arbitrary recursive computations. Both Solar-Lezama [12] and Blanc et al. [4] restrict holes to stand for proper subexpressions. This choice makes the synthesis problem more tractable than allowing holes to be placed anywhere, and these tools achieve competitive benchmark performance. In contrast, our use of miniKanren gives us the freedom to place logic variables anywhere in a program, allowing us to synthesize any syntactic fragment, but makes it more difficult to achieve competitive performance.

There has been little published work that focuses exclusively on synthesizing JavaScript, possibly due to JavaScript's complexity. Some works apply a general purpose technique to JavaScript. For example, Padhye et al. [11] applies their semantic fuzzing technique to generate syntactically correct JavaScript for the purpose of finding flaws in JavaScript implementations. However, their application does not require reasoning about the semantics of the generated code. Other work on JavaScript synthesis focuses on narrow subsets of the language relevant to web programming, such as Bajaj et al. [1], which synthesizes DOM selectors by example.

## 7 CONCLUSION

We described a proof-of-concept relational interpreter for a subset of the JavaScript programming language, formed by composing a JavaScript parser and LambdaJS interpreter implemented as two separate relations. We have demonstrated that this interpreter can be used to solve small, and sometimes tricky, synthesis problems. We also described some characteristics of the parser-interpreter design that were not obvious.

## ACKNOWLEDGMENTS

We thank William E. Byrd and the anonymous reviewers for their helpful comments. We also thank Ina Jacobson for helpful feedback on the writing. Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

## REFERENCES

[1] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2015. Synthesizing Web Element Locators. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 331–341.

[2] Michael Ballantyne and William E. Byrd. 2015. A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez. https://github.com/michaelballantyne/faster-miniKanren

[3] Gary Bernhardt. 2012. Wat. A lightning talk from CodeMash, 2012. https://www.destroyallsoftware.com/talks/wat

[4] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. An overview of the Leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*. 1–10.

[5] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

[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*. ACM, 8–29.

[7] Molly Q Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards answering "Am I on the right track?" automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 13–24.

[8] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *European conference on Object-oriented programming*. Springer, 126–150.

[9] Oleg Kiselyov, William E. Byrd, Daniel P Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *International Symposium on Functional and Logic Programming*. Springer, 64–80.

[10] Gilmore R. Lundquist, Utsav Bhatt, and Kevin W. Hamlen. 2019. Relational Processing for Fun and Diversity. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop. Harvard Computer Science Group Technical Report TR-02-19*. 100–113.

[11] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[12] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley.

[13] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*. 1737–1746.

## A RELATIONAL LAMBDAJS INTERPRETER

```
;; Entry point for the LambdaJS interpreter
(define (evalo expr val store~)
  (fresh (next-address~)
    (eval-envo expr '() val '() store~ '() next-address~)))

;; LambdaJS interpreter with store
(define (eval-envo expr env value
                   store store~                ; mutable store before/after
                   next-address next-address~) ; index counter for the store
  (conde
    ;; Atomic values (Section 2.2.1)
    ((== expr value)
     (== store store~)
     (== next-address next-address~)
     (conde ((fresh (payload) (== expr (jrawnum payload))))
            ((fresh (bindings) (== expr (jobj bindings))))
            ((fresh (b) (== expr (jbool b))))
            ((fresh (payload) (== expr (jrawstr payload))))
            ((== expr (jundef)))
            ((== expr (jnul)))))
    ;; Immutable variable lookup (Section 2.2.2)
    ((fresh (var) ;; Look up a variable
       (== expr (jvar var))
       (== store store~)
       (== next-address next-address~)
       (lookupo var env value)))
    ;; Builtin operations (Section 3.2.5)
    ((fresh (rator rands args value^)
       (== expr (jdelta rator rands))
       (eval-env-listo rands env value^ store store~ next-address next-address~)
       (effect-propagateo value^ value store~ store~ next-address~ next-address~
         (== value^ (value-list args))
         (conde
           ((fresh (v1 v2 digits1 digits2 result remainder) ; numeric operations
              (== `(,v1 ,v2) args)
              (typeofo v1 (jstr "number") store~)
              (typeofo v2 (jstr "number") store~)
              (== `(,(jrawnum digits1) ,(jrawnum digits2)) args)
              (conde ((== rator '+)
                      (== value (jrawnum result))
                      (pluso digits1 digits2 result))
                     ((== rator '-)
                      (== value (jrawnum result))
                      (minuso digits1 digits2 result))
```

```
                  ((== rator '*)
                   (== value (jrawnum result))
                   (*o digits1 digits2 result))
                  ((== rator '/)
                   (== value (jrawnum result))
                   (/o digits1 digits2 result remainder))
                  ((== rator '<)
                   (conde ((== value (jbool #t)) (<o digits1 digits2))
                          ((== value (jbool #f)) (<=o digits2 digits1)))))))
       ((fresh (v1 v2) ; ===
          (== `(,rator ,args) `(=== (,v1 ,v2)))
          (conde ((== value (jbool #t)) (== v1 v2))
                 ((== value (jbool #f)) (=/= v1 v2)))))
       ((fresh (v1) ; typeof
          (== `(,rator (,v1)) `(typeof ,args))
          (typeofo v1 value store~)))
       ((fresh (str char) ; char->nat
          (== `(,str) args)
          (typeofo str (jstr "string") store~)
          (== `(,(jrawstr `(,char))) args)
          (== rator 'char->nat)
          (== value (jrawnum char))))
       ((fresh (num digits) ; nat->char
          (== `(,num) args)
          (typeofo num (jstr "number") store~)
          (== `(,(jrawnum digits)) args)
          (== rator 'nat->char)
          (== value (jrawstr `(,digits)))))
       ((fresh (v1 v2 chars1 chars2 result) ; string operations
          (== `(,v1 ,v2) args)
          (typeofo v1 (jstr "string") store~)
          (typeofo v2 (jstr "string") store~)
          (== `(,(jrawstr chars1) ,(jrawstr chars2)) args)
          (conde ((== rator 'string+)
                  (== value (jrawstr result))
                  (appendo chars1 chars2 result))
                 ((== rator 'string<)
                  (string-lesso chars1 chars2 value)))))))))
;; Mutable references: dereferencing (Section 2.2.3)
((fresh (addr-expr addr-val value^)
   (== expr (jderef addr-expr))
   (eval-envo addr-expr env value^ store store~
              next-address next-address~)
   (effect-propagateo value^ value store~ store~
                      next-address~ next-address~
     (== value^ (jref addr-val))
```

```
     (indexo store~ addr-val value))))
;; Mutable references: assignment (Section 2.2.3)
((fresh (addr-expr addr-val stored-expr stored-val store^ value^)
   (== expr (jassign addr-expr stored-expr))
   (eval-env-listo `(,addr-expr ,stored-expr) env value^ store store^
                   next-address next-address~)
   (effect-propagateo value^ value store^ store~ next-address~ next-address~
     (== value^ (value-list `(,(jref addr-val) ,stored-val)))
     (== value stored-val)
     (set-indexo store^ addr-val stored-val store~))))
;; Object field retrieval (2.2.4)
((fresh (obj-expr obj-bindings key-expr key-val value^)
   (== expr (jget obj-expr key-expr))
   (eval-env-listo `(,key-expr ,obj-expr) env value^ store store~
                   next-address next-address~)
   (effect-propagateo value^ value store~ store~ next-address~ next-address~
     (== value^ (value-list `(,key-val ,(jobj obj-bindings))))
     (typeofo key-val (jstr "string") store~)
     (conde ((alist-refo key-val obj-bindings value))     ; found
            ((== value (jundef))                          ; not found
             (absent-keyso key-val obj-bindings))))))
;; Object field create/update (2.2.4)
((fresh (obj-expr obj-bindings obj-bindings^
         key-expr key-val rhs-expr rhs-val value^)
   (== expr (jset obj-expr key-expr rhs-expr))
   (eval-env-listo `(,obj-expr ,key-expr ,rhs-expr) env value^ store store~
                   next-address next-address~)
   (effect-propagateo value^ value store~ store~ next-address~ next-address~
     (== value^ (value-list `(,(jobj obj-bindings) ,key-val ,rhs-val)))
     (== value (jobj obj-bindings^))
     (typeofo key-val (jstr "string") store~)
     (updateo obj-bindings key-val rhs-val obj-bindings^))))
;; Function application (Section 2.2.4)
((fresh (func params rands args body
         param-arg-bindings
         cenv cenv^ ; closure environment before/after param-arg-bindings
         value^ store^ next-address^)
   (== expr (japp func rands))
   (eval-env-listo `(,func . ,rands) env value^ store store^
                   next-address next-address^)
   (effect-propagateo value^ value store^ store~ next-address^ next-address~
     (== value^ (value-list `(,(jclo params body cenv) . ,args)))
     (zipo params args param-arg-bindings)
     (appendo param-arg-bindings cenv cenv^)
     (eval-envo body cenv^ value store^ store~
                next-address^ next-address~))))
```

```
;; Throw expressions (Section 2.2.5)
((fresh (label thrown-expr thrown-val) ;; Throw
   (== expr (jthrow label thrown-expr))
   (eval-envo thrown-expr env thrown-val store store~
              next-address next-address~)
   (effect-propagateo thrown-val value store~ store~
                      next-address~ next-address~
     (== value (jbrk label thrown-val)))))
;; Let expressions (Section 2.2.2)
((fresh (lhs-var          ; variable being bound
         rhs-expr rhs-val ; right-hand side expression & value
         store^ next-address^ ; store produced by evaluating the rhs
         body
         let-env)         ; environment after the let binding
   (== expr (jlet lhs-var rhs-expr body))
   (eval-envo rhs-expr env rhs-val store store^ next-address next-address^)
   (effect-propagateo rhs-val value store^ store~
                      next-address^ next-address~
     (== let-env `((,lhs-var . ,rhs-val) . ,env))
     (eval-envo body let-env value store^ store~
                next-address^ next-address~))))
;; Function definition (Section 2.2.4)
((fresh (body params)
   (== expr (jfun params body))
   (== value (jclo params body env))
   (== store store~)
   (== next-address next-address~)))
;; Mutable references: memory allocation (Section 2.2.3)
((fresh (stored-expr stored-val next-address^ store^)
   (== expr (jall stored-expr))
   (eval-envo stored-expr env stored-val store store^
              next-address next-address^)
   (effect-propagateo stored-val value store^ store~
                      next-address^ next-address~
     (== value (jref next-address^))
     (appendo store^ `(,stored-val) store~)
     (incremento next-address^ next-address~))))
;; Object field delete (Section 2.2.4)
((fresh (obj-expr obj-bindings obj-bindings^
         key-expr key-val value^)
   (== expr (jdel obj-expr key-expr))
   (eval-env-listo `(,obj-expr ,key-expr) env value^ store store~
                   next-address next-address~)
   (effect-propagateo value^ value store~ store~ next-address~ next-address~
     (== value^ (value-list `(,(jobj obj-bindings) ,key-val)))
     (== value (jobj obj-bindings^))
```

```
     (typeofo key-val (jstr "string") store~)
     (deleteo obj-bindings key-val obj-bindings^))))
;; Begin expression (Section 2.2.5)
((fresh (first-expr second-expr first-val value^)
   (== expr (jbeg first-expr second-expr))
   (eval-env-listo `(,first-expr ,second-expr) env value^ store store~
                   next-address next-address~)
   (effect-propagateo value^ value store~ store~ next-address~ next-address~
     (== value^ (value-list `(,first-val ,value))))))
;; If expression (Section 2.2.5)
((fresh (cond-expr cond-val then-expr else-expr store^ next-address^)
   (== expr (jif cond-expr then-expr else-expr))
   (eval-envo cond-expr env cond-val store store^ next-address next-address^)
   (effect-propagateo cond-val value store^ store~ next-address^ next-address~
     (conde ((== cond-val (jbool #f))
             (eval-envo else-expr env value store^ store~
                        next-address^ next-address~))
            ((== cond-val (jbool #t))
             (eval-envo then-expr env value store^ store~
                        next-address^ next-address~))))))
;; While expression (Section 2.2.5)
((fresh (cond-expr cond-val body-expr store^ next-address^)
   (== expr (jwhile cond-expr body-expr))
   (eval-envo cond-expr env cond-val store store^
              next-address next-address^)
   (effect-propagateo cond-val value store^ store~
                      next-address^ next-address~
     (conde ((== cond-val (jbool #f))
             (== value (jundef))
             (== store^ store~)
             (== next-address^ next-address~))
            ((== cond-val (jbool #t))
             (eval-envo (jbeg body-expr (jwhile cond-expr body-expr))
                        env value store^ store~
                        next-address^ next-address~))))))
;; Try/finally expression (Section 2.2.5)
((fresh (try-expr try-val finally-expr finally-val store^ next-address^)
   (== expr (jfin try-expr finally-expr))
   (eval-envo try-expr env try-val store store^ next-address next-address^)
   (eval-envo finally-expr env finally-val store^ store~
              next-address^ next-address~)
   (effect-propagateo finally-val value store~ store~
                      next-address~ next-address~
     (== value try-val))))
;; Try/catch expression (Section 2.2.5)
((fresh (try-expr try-val try-val-tag try-val-payload
```

```
            catch-label catch-var catch-expr
            break-label break-val
            store^ next-address^ env^)
       (== expr (jcatch catch-label try-expr catch-var catch-expr))
       (eval-envo try-expr env try-val store store^ next-address next-address^)
       (conde ((== try-val (jbrk break-label break-val)) ; break doesn't match
               (== `(,value ,store~ ,next-address~)
                   `(,try-val ,store^ ,next-address^))
               (=/= break-label catch-label))
              ((== try-val `(,try-val-tag . ,try-val-payload)) ; no break caught
               (== `(,value ,store~ ,next-address~)
                   `(,try-val ,store^ ,next-address^))
               (=/= try-val-tag 'break))
              ((== try-val (jbrk catch-label break-val)) ; break caught
               (appendo `((,catch-var . ,break-val)) env env^)
               (eval-envo catch-expr env^ value store^ store~
                          next-address^ next-address~)))))))

;; Sequentially evaluate a list of LambdaJS expressions
(define (eval-env-listo exprs env vals store store~ next-address next-address~)
  (conde
    ; base case: empty list of expressions
    ((== exprs '())
     (== vals (value-list '()))
     (== store store~)
     (== next-address next-address~))
    ; non-empty list of expressions
    ((fresh (expr expr-rest val val-rest val-rest-payload store^ next-address^)
       (== exprs `(,expr . ,expr-rest))
       (eval-envo expr env val store store^ next-address next-address^)
       (effect-propagateo val vals store^ store~ next-address^ next-address~
         (eval-env-listo expr-rest env val-rest store^ store~
                         next-address^ next-address~)
         (effect-propagateo val-rest vals store~ store~
                            next-address~ next-address~
           (== val-rest (value-list val-rest-payload))
           (== vals (value-list `(,val . ,val-rest-payload)))))))))

; For propagating control effects (Section 2.2.6)
(define-syntax-rule (effect-propagateo value^        value~
                                       store^        store~
                                       next-address^ next-address~
                                       cont ...)
  (fresh (label bval tag rest)
    (conde ((== (jbrk label bval) value^)
            (== `(,value~ ,store~ ,next-address~)
```

```
                `(,value^ ,store^ ,next-address^)))
            ((== `(,tag . ,rest) value^)
             (=/= tag 'break)
             cont ...))))

(define (typeofo value type store)
  (fresh (temp)
    (conde ((== value (jundef))
            (== type (jstr "undefined")))
           ((== value (jnul))
            (== type (jstr "object")))
           ((== value `(string . ,temp))
            (== type (jstr "string")))
           ((== value  `(number . ,temp))
            (== type (jstr "number")))
           ((== value  `(boolean . ,temp))
            (== type (jstr "boolean")))
           ((fresh (fields priv call)
              (== value (jref temp))
              (conde ((== type (jstr "object"))
                      (indexo store temp `(object ,fields))
                      (lookupo (jstr "private") fields (jobj priv))
                      (absent-keyso (jstr "call") priv))
                     ((== type (jstr "function"))
                      (indexo store temp `(object ,fields))
                      (lookupo (jstr "private") fields (jobj priv))
                      (lookupo (jstr "call") priv call))))))))

(define (string-lesso s1 s2 value)
  (fresh (x x^ y y^ rest)
    (conde ((== s1 '())
            (== s2 `(,x . ,rest))
            (== value (jbool #t)))
           ((== s2 '())
            (== value (jbool #f)))
           ((== s1 `(,x . ,x^))
            (== s2 `(,x . ,y^))
            (string-lesso x^ y^ value))
           ((== s1 `(,x . ,x^))
            (== s2 `(,y . ,y^))
            (== value (jbool #t))
            (=/= x y)
            (<o x y))
           ((== s1 `(,x . ,x^))
            (== s2 `(,y . ,y^))
            (== value (jbool #f))
```

```
            (=/= x y)
            (<=o y x)))))

(define (value-list values)
  (cons 'value-list values))
```

## B RELATIONAL S-EXPRESSION JAVASCRIPT TO LAMBDAJS PARSER

```
; Parse a JavaScript statement with human-readable literals
(define (parseo/readable stmt jexpr)
  (parse-topo (dehumanize stmt) jexpr))

; Parse a JavaScript statement with relational number and string literals
(define (parse-topo stmt jexpr)
  (fresh (vars exp^ allocations body)
    (hoist-varo stmt vars)
    (allocateo vars body jexpr)
    (parseo stmt body)))

; Parse a JavaScript statement to LambdaJS expression
(define (parseo stmt jexpr)
  (conde
    ; variable declaration (Section 3.2.1)
    ((fresh (vars)
       (== stmt `(var . ,vars))
       (parse-var-assignmentso vars jexpr)))
    ; expressions have a helper for their own
    ((parse-expro stmt jexpr))
    ; begin statement (Section 3.2.7)
    ((fresh (stmts jexprs begin-jexpr)
       (== stmt `(begin . ,stmts))
       (== jexpr (jbeg begin-jexpr (jundef)))
       (parse-listo stmts jexprs)
       (begino jexprs begin-jexpr)))
    ; if statement  (Section 3.2.7)
    ((fresh (cond-expr then-stmt else-stmt cond-jexpr then-jexpr else-jexpr)
       (== stmt `(if ,cond-expr ,then-stmt ,else-stmt))
       (== jexpr (jbeg (jif cond-jexpr then-jexpr else-jexpr) (jundef)))
       (parse-expro cond-expr cond-jexpr)
       (parse-listo `(,then-stmt ,else-stmt) `(,then-jexpr ,else-jexpr))))
    ; for loops (Section 3.2.7)
    ((fresh (init-stmt init-jexpr
             cond-expr cond-jexpr
             inc-expr inc-jexpr
             body-stmts body-jexprs body-jexpr)
       (== stmt `(for (,init-stmt ,cond-expr ,inc-expr) . ,body-stmts))
```

```
   (== jexpr (jbeg init-jexpr
                  (jcatch 'break
                          (jwhile cond-jexpr (jbeg body-jexpr inc-jexpr))
                          'e (jundef))))
   (parse-expr-listo `(,cond-expr ,inc-expr) `(,cond-jexpr ,inc-jexpr))
   (parseo init-stmt init-jexpr)
   (parse-listo body-stmts body-jexprs)
   (begino body-jexprs body-jexpr)))
; return control effect (Section 3.2.8)
((fresh (val-expr val-jexpr)
   (== stmt `(return ,val-expr))
   (== jexpr (jthrow 'return val-jexpr))
   (parse-expro val-expr val-jexpr)))
; throw control effect (Section 3.2.8)
((fresh (val-expr val-jexpr)  ;; throw
   (== stmt `(throw ,val-expr))
   (== jexpr (jthrow 'error val-jexpr))
   (parse-expro val-expr val-jexpr)))
; break control effect (Section 3.2.8)
((== stmt `(break)) (== jexpr (jthrow 'break (jundef))))
; try/catch control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr catch-stmt catch-jexpr catch-var)
   (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt))
   (== jexpr
       (jbeg (jcatch 'error try-jexpr catch-var
                     (jlet catch-var (jall (jvar catch-var)) catch-jexpr))
             (jundef)))
   (parseo try-stmt try-jexpr)
   (parseo catch-stmt catch-jexpr)))
; try/finally control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr finally-stmt finally-jexpr)
   (== stmt `(try ,try-stmt finally ,finally-stmt))
   (== jexpr (jfin try-jexpr (jbeg finally-jexpr (jundef))))
   (parseo try-stmt try-jexpr)
   (parseo finally-stmt finally-jexpr)))
; try/catch/finally control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr
         catch-stmt catch-jexpr catch-var
         finally-stmt finally-jexpr)
   (== stmt `(try ,try-stmt
              catch ,catch-var ,catch-stmt
              finally ,finally-stmt))
   (== jexpr (jbeg (jcatch 'error try-jexpr catch-var
                           (jlet catch-var (jall (jvar catch-var))
                                 catch-jexpr))
                   (jbeg finally-jexpr (jundef))))
```

```
       (parseo try-stmt try-jexpr)
       (parseo catch-stmt catch-jexpr)
       (parseo finally-stmt finally-jexpr)))
    ; while statements (Section 3.2.7)
    ((fresh (cond-expr cond-jexpr body-stmts body-jexprs body-jexpr)
       (== stmt `(while ,cond-expr . ,body-stmts))
       (== jexpr (jcatch 'break (jwhile cond-jexpr body-jexpr) 'e (jundef)))
       (parse-expro cond-expr cond-jexpr)
       (parse-listo body-stmts body-jexprs)
       (begino body-jexprs body-jexpr)))))

; Parse a JavaScript expression to LambdaJS expression
(define (parse-expro expr jexpr)
  (conde
    ;; variables (Section 3.2.1)
    ((symbolo expr) (== jexpr (jderef (jvar expr))))
    ;; simple literals
    ((conde ((== expr jexpr) (conde ((fresh (x) (== expr (jrawnum x))))
                                    ((fresh (x) (== expr (jrawstr x))))))
            ((== expr #t) (== jexpr (jbool #t)))
            ((== expr #f) (== jexpr (jbool #f)))
            ((== expr (jnul)) (== jexpr (jnul)))
            ((== expr (jundef)) (== jexpr (jundef)))))
    ;; builtin operations (Section 3.2.5)
    ((fresh (rator rands rand-jexprs)
       (== expr `(op ,rator . ,rands))
       (== jexpr (jdelta rator rand-jexprs))
       (parse-expr-listo rands rand-jexprs)))
    ;; assignments (Section 3.2.6)
    ((fresh (lhs-expr rhs-expr rhs-jexpr)
       (== expr `(:= ,lhs-expr ,rhs-expr))
       (conde ((symbolo lhs-expr)
               (== jexpr (jassign (jvar lhs-expr) rhs-jexpr))
               (parse-expro rhs-expr rhs-jexpr))
              ((fresh (obj-expr obj-jexpr key-expr key-jexpr)
                 (== lhs-expr `(@ ,obj-expr ,key-expr))
                 (== jexpr
                     (japp (jfun '(obj key rhs)
                                 (jbeg (jassign
                                         (jvar 'obj)
                                         (jset (jderef (jvar 'obj))
                                               (jstr "public")
                                               (jset (jget (jderef (jvar 'obj))
                                                           (jstr "public"))
                                                     (jvar 'key) (jvar 'rhs))))
                                       (jvar 'rhs)))
```

```
                        (list obj-jexpr key-jexpr rhs-jexpr)))
             (parse-expro obj-expr obj-jexpr)
             (parse-expro key-expr key-jexpr)
             (parse-expro rhs-expr rhs-jexpr))))))
;; object field access (Section 3.2.2)
((fresh (obj-expr obj-jexpr field-expr field-jexpr)
   (== expr `(@ ,obj-expr ,field-expr))
   (== jexpr (jget (jget (jderef obj-jexpr) (jstr "public")) field-jexpr))
   (parse-expro obj-expr obj-jexpr)
   (parse-expro field-expr field-jexpr)))
;; Function call (Section 3.2.4)
((fresh (func-expr func-jexpr arg-exprs arg-jexprs)
   (== expr `(call ,func-expr . ,arg-exprs))
   (== jexpr (japp (jget (jget (jderef func-jexpr) (jstr "private"))
                         (jstr "call"))
                   arg-jexprs))
   (parse-expro func-expr func-jexpr)
   (parse-expr-listo arg-exprs arg-jexprs)))
;; object creation (Section 3.2.2)
((fresh (binding-exprs public-jexpr)
   (== expr `(object . ,binding-exprs))
   (== jexpr (jall (jset (jobj `((,(jstr "private") . ,(jobj '()))))
                         (jstr "public") public-jexpr)))
   (parse-obj-bindingso binding-exprs public-jexpr)))
;; function definition (Section 3.2.3)
((fresh (params
          body-stmts ; javascript statements
          body-jexprs ; a list of LambdaJS expressions
          body-jexpr ; a single LambdaJS begin expression
          body-jexpr/vars
          body-jexpr/vars+params
          vars
          hoisted-vars
          return-var)
   (== expr `(function ,params . ,body-stmts))
   (== jexpr (jall (jset (jobj `((,(jstr "public") . ,(jobj '()))))
                         (jstr "private")
                         (jset (jobj '()) (jstr "call")
                               (jfun params
                                     (jcatch 'return
                                             (jbeg body-jexpr/vars+params
                                                   (jundef))
                                             'result
                                             (jvar 'result)))))))
   (hoist-var-listo body-stmts vars)
   (differenceo vars params hoisted-vars)
```

```
       (parse-listo body-stmts body-jexprs)
       (begino body-jexprs body-jexpr)
       (allocateo hoisted-vars body-jexpr body-jexpr/vars)
       (assigno params body-jexpr/vars body-jexpr/vars+params)))
    ;; Comma expression sequencing (not in paper since we can't decide on a name)
    ((fresh (exps exps^)
       (== expr `(comma . ,exps))
       (parse-expr-listo exps exps^)
       (begino exps^ jexpr)))))

; Parse a list of statements
(define (parse-listo stmts jexprs)
  (conde ((== stmts '()) (== jexprs '()))
         ((fresh (stmt jexpr stmts-rest jexprs-rest)
            (== stmts `(,stmt . ,stmts-rest))
            (== jexprs `(,jexpr . ,jexprs-rest))
            (parseo stmt jexpr)
            (parse-listo stmts-rest jexprs-rest)))))

; Parse a list of expressions
(define (parse-expr-listo exprs jexprs)
  (conde ((== exprs '()) (== jexprs '()))
         ((fresh (expr jexpr exprs-rest jexprs-rest)
            (== exprs `(,expr . ,exprs-rest))
            (== jexprs `(,jexpr . ,jexprs-rest))
            (parse-expro expr jexpr)
            (parse-expr-listo exprs-rest jexprs-rest)))))

(define (parse-var-assignmentso vars assignments-jexpr)
  (conde ((== vars '()) (== assignments-jexpr (jundef)))
         ((fresh (var-name var-expr var-jexpr vars-rest assignments-rest-jexpr)
            (== vars `((,var-name ,var-expr) . ,vars-rest))
            (== assignments-jexpr (jbeg (jassign (jvar var-name) var-jexpr)
                                        assignments-rest-jexpr))
            (parse-expro var-expr var-jexpr)
            (parse-var-assignmentso vars-rest assignments-rest-jexpr)))
         ((fresh (var-name vars-rest)
            (== vars `(,var-name . ,vars-rest))
            (symbolo var-name)
            (parse-var-assignmentso vars-rest assignments-jexpr)))))

; object {...}
(define (parse-obj-bindingso binding-exprs obj-jexpr)
  (conde ((== binding-exprs '()) (== obj-jexpr (jobj '())))
         ((fresh (field-expr field-jexpr val-expr val-jexpr
                  binding-exprs-rest obj-jexpr-rest)
```

```
            (== binding-exprs  `((,field-expr ,val-expr) . ,binding-exprs-rest))
            (== obj-jexpr (jset obj-jexpr-rest field-jexpr val-jexpr))
            (parse-expro field-expr field-jexpr)
            (parse-expro val-expr val-jexpr)
            (parse-obj-bindingso binding-exprs-rest obj-jexpr-rest)))))

; Build nested LambdaJS begin out of a list of LambdaJS exprs
(define (begino jexprs jexpr)
  (conde ((== jexprs '()) (== jexpr (jundef)))
         ((== jexprs `(,jexpr)))
         ((fresh (first rest rest-jexpr)
            (== jexprs`(,first . ,rest))
            (=/= rest '())
            (== jexpr (jbeg first rest-jexpr))
            (begino rest rest-jexpr)))))

; Hoist declared variables out of a statement
(define (hoist-varo stmt vars)
  (conde ((fresh (x) (== stmt `(var . ,x)) (var-nameso x vars)))
         ((== vars '())  ;; These never embed var declarations
          (conde ((fresh (x) (== stmt `(return ,x))))
                 ((fresh (x) (== stmt `(throw ,x))))
                 ((fresh (x) (== stmt `(number ,x))))
                 ((fresh (x) (== stmt `(string ,x))))
                 ((fresh (x) (== stmt `(object . ,x))))
                 ((fresh (x) (== stmt `(comma . ,x))))
                 ((== stmt #t))
                 ((== stmt #f))
                 ((== stmt (jundef)))
                 ((== stmt (jnul)))
                 ((symbolo stmt))
                 ((fresh (func args) (== stmt `(call ,func . ,args))))
                 ((fresh (erest) (== stmt `(function . ,erest))))
                 ((== stmt `(break)))
                 ((fresh (x) (== stmt `(op . ,x))))
                 ((fresh (x) (== stmt `(@ . ,x))))
                 ((fresh (x) (== stmt `(:= . ,x))))))
         ((fresh (try-stmt catch-stmt catch-var)
            (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt))
            (hoist-var-listo `(,try-stmt ,catch-stmt) vars)))
         ((fresh (try-stmt catch-stmt finally-stmt catch-var)
            (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt
                           finally ,finally-stmt))
            (hoist-var-listo `(,try-stmt ,catch-stmt ,finally-stmt) vars)))
         ((fresh (cond then else)
            (== stmt `(if ,cond ,then ,else))
```

```
           (hoist-var-listo `(,then ,else) vars)))
         ((fresh (cond body)
           (== stmt `(while ,cond . ,body))
           (hoist-var-listo body vars)))
         ((fresh (exps)
           (== stmt `(begin . ,exps))
           (hoist-var-listo exps vars)))
         ((fresh (init cond inc body)
           (== stmt `(for (,init ,cond ,inc) . ,body))
           (hoist-var-listo `(,init . ,body) vars)))))

(define (hoist-var-listo stmts vars)
  (conde ((== stmts '()) (== vars '()))
         ((fresh (stmt stmts-rest vars-first vars-rest)
           (== stmts `(,stmt . ,stmts-rest))
           (hoist-varo stmt vars-first)
           (hoist-var-listo stmts-rest vars-rest)
           (appendo vars-first vars-rest vars)))))

(define (var-nameso var-decls names)
  (conde ((== var-decls '()) (== names '()))
         ((fresh (var-decl var-decls-rest name expr names-rest)
           (== var-decls `(,var-decl . ,var-decls-rest))
           (== names `(,name . ,names-rest))
           (conde ((== var-decl `(,name ,expr)))
                  ((symbolo var-decl) (== name var-decl)))
           (var-nameso var-decls-rest names-rest)))))

(define (allocateo var-names body-jexpr full-jexpr)
  (conde ((== var-names '()) (== full-jexpr body-jexpr))
         ((fresh (var-name vars-rest rest-jexpr)
           (== var-names `(,var-name . ,vars-rest))
           (== full-jexpr (jlet var-name (jall (jundef)) rest-jexpr))
           (allocateo vars-rest body-jexpr rest-jexpr)))))

(define (assigno var-names body-jexpr full-jexpr)
  (conde ((== var-names '()) (== full-jexpr body-jexpr))
         ((fresh (var-name vars-rest rest-jexpr)
           (== var-names `(,var-name . ,vars-rest))
           (== full-jexpr (jlet var-name (jall (jvar var-name)) rest-jexpr))
           (assigno vars-rest body-jexpr rest-jexpr)))))

; list (set) difference operation
(define (differenceo items toremove remaining)
  (conde ((== items '()) (== remaining items))
         ((fresh (el rest remaining-rest)
```

```
            (== items `(,el . ,rest))
            (conde ((== remaining `(,el . ,remaining-rest))
                    (not-in-listo el toremove)
                    (differenceo rest toremove remaining-rest))
                   ((membero el toremove)
                    (differenceo rest toremove remaining)))))))

(define (humanize-string x)
  (define (humanize-char x)
    (define n (mknum->num x))
    (if (integer? n) (integer->char n) n))
  (if (list? x)
    (let ((cs (map humanize-char x)))
      (if (andmap char? cs) (list->string cs) `(string ,cs)))
    `(string ,x)))

(define/match (humanize stmt)
  [((list 'string x)) (humanize-string x)]
  [((list 'number x))
   (define result (mknum->num x))
   (if (integer? result) result `(number ,result))]
  [((list)) '()]
  [((? list?)) (cons (humanize (car stmt)) (humanize (cdr stmt)))]
  [(_) stmt])

(define/match (dehumanize stmt)
  [((? string?)) (jstr stmt)]
  [((? integer?)) (jnum stmt)]
  [((list)) '()]
  [((? list?)) (cons (dehumanize (car stmt)) (dehumanize (cdr stmt)))]
  [(_) stmt])

(define (mknum->num xs)
  (if (and (list? xs) (andmap integer? xs))
    (foldr (lambda (d n) (+ d (* 2 n))) 0 xs)
    xs))
```

## C LAMBDAJS STRUCTURE DEFINITIONS

```
(define (jlet key value exp)
  `(let ,key ,value ,exp))

(define (jfun params body)
  `(fun ,params ,body))

(define (jclo params body env)
```

```
  `(jclosure ,params ,body ,env))

(define (japp closure args)
  `(app ,closure ,args))

(define (jget obj key)
  `(get ,obj ,key))

(define (jset obj key value)
  `(set ,obj ,key ,value))

(define (jdel obj key)
  `(delete ,obj ,key))

(define (jvar var)
  `(var ,var))

(define (jrawnum n)
  `(number ,n))

(define (jnum n)
  `(number ,(build-num n)))

(define (jobj bindings)
  `(object ,bindings))

(define (jall value)
  `(allocate ,value))

(define (jref value)
  `(ref ,value))

(define (jderef address)
  `(deref ,address))

(define (jassign var val)
  `(assign ,var ,val))

(define (jbeg first second)
  `(begin ,first ,second))

(define (jbool bool)
  `(boolean ,bool))

(define (jif cond then else)
  `(if ,cond ,then ,else))
```

```
(define (jundef)
  '(undefined))

(define (jnul)
  '(null))

(define (jwhile cond body)
  `(while ,cond ,body))

(define (jthrow label value)
  `(throw ,label ,value))

(define (jbrk label value)
  `(break ,label ,value))

(define (jfin try-exp fin-exp)
  `(finally ,try-exp ,fin-exp))

(define (jcatch label try-exp catch-var catch-exp)
  `(catch ,label ,try-exp ,catch-var ,catch-exp))

(define (jrawstr str)
  `(string ,str))

(define (jstr str)
  `(string ,(map (compose1 build-num char->integer) (string->list str))))

(define (jdelta fun vals)
  `(delta ,fun ,vals))

(define (jpass) 'pass)
```

## D OTHER RELATIONAL UTILITIES

```
(define (incremento in out)
  (== out `(,in)))

(define (decremento in out)
  (incremento out in))

(define (absent-keyso key obj)
  (conde ((== obj '()))
         ((fresh (k v rest)
            (== obj `((,k . ,v) . ,rest))
            (=/= k key)
```

```
            (absent-keyso key rest)))))

(define (updateo obj key value result)
  (conde ((== obj '())
          (== result `((,key . ,value))))
         ((fresh (orest rrest k v v2)
            (== obj    `((,k . ,v ) . ,orest))
            (== result `((,k . ,v2) . ,rrest))
            (conde ((== k key)
                    (== v2 value)
                    (== orest rrest))
                   ((=/= k key)
                    (== v2 v)
                    (updateo orest key value rrest)))))))

(define (deleteo obj key result)
  (conde ((== obj '()) (== result '()))
         ((fresh (orest rrest k v)
            (== obj `((,k . ,v) . ,orest))
            (conde ((== k key)
                    (== orest result))
                   ((== result `((,k . ,v) . ,rrest))
                    (=/= k key)
                    (deleteo orest key rrest)))))))

(define (appendo s t r)
  (conde ((== s '()) (== t r))
         ((fresh (r^ sel srest)
            (== r `(,sel . ,r^))
            (== s `(,sel . ,srest))
            (appendo srest t r^)))))

(define (zipo as ds pairs)
  (conde ((== as '()) (== ds '()) (== pairs '()))
         ((fresh (a as-rest d ds-rest pairs-rest)
            (== as `(,a . ,as-rest))
            (== ds `(,d . ,ds-rest))
            (== pairs `((,a . ,d) . ,pairs-rest))
            (zipo as-rest ds-rest pairs-rest)))))

(define (not-in-listo el list)
  (conde
    ((== list '()))
    ((fresh (x rest)
       (== `(,x . ,rest) list)
       (=/= el x)
```

```
       (not-in-listo el rest)))))

(define (lookupo x env t)
  (fresh (rest y v)
    (== `((,y . ,v) . ,rest) env)
    (conde
      ((== y x) (== v t))
      ((=/= y x) (lookupo x rest t)))))

(define (indexo lst index result)
  (fresh (l lrest index^)
    (== lst `(,l . ,lrest))
    (conde ((== index '())
            (== result l))
           ((=/= index '())
            (decremento index index^)
            (indexo lrest index^ result)))))

(define (set-indexo lst index value result)
  (fresh (l lrest rrest index^)
    (== lst `(,l . ,lrest))
    (conde ((== index '())
            (== result `(,value . ,lrest)))
           ((== result `(,l . ,rrest))
            (=/= index '())
            (decremento index index^)
            (set-indexo lrest index^ value rrest)))))

(define (membero item lst)
  (fresh (first rest)
    (== lst `(,first . ,rest))
    (conde ((== first item))
           ((=/= first item) (membero item rest)))))

(define (no-alist-refo key lst)
  (conde ((== '() lst))
         ((fresh (first-key first-value rest)
            (== lst `((,first-key . ,first-value) . ,rest))
            (=/= key first-key)
            (no-alist-refo key rest)))))

(define (alist-refo key lst value)
  (fresh (first-key first-value rest)
    (== lst `((,first-key . ,first-value) . ,rest))
    (conde ((== first-key key)
            (== value first-value)
```

```
 (no-alist-refo key rest))
((=/= first-key key)
 (alist-refo key rest value)))))
```

# dxo: A System for Relational Algebra and Differentiation

JULIE S. STEELE, Georgetown Day School, USA
WILLIAM E. BYRD, University of Alabama at Birmingham, USA

We present *dxo*, a relational system for algebra and differentiation, written in miniKanren. *dxo* operates over math expressions, represented as s-expressions. *dxo* supports addition, multiplication, exponentiation, variables (represented as tagged symbols), and natural numbers (represented as little-endian binary lists). We show the full code for *dxo*, and describe in detail the four main relations that compose *dxo*. We present example problems *dxo* can solve by combining the main relations. Our differentiation relation, `do`, can differentiate polynomials, and by running backwards, can also integrate. Similarly, our simplification relation, `simpo`, can simplify expressions that include addition, multiplication, exponentiation, variables, and natural numbers, and by running backwards, can complicate any expression in simplified form. Our evaluation relation, `evalo`, takes the same types of expressions as `simpo`, along with an environment associating variables with natural numbers. By evaluating the expression with respect to the environment, `evalo` can produce a natural number; by running backwards, `evalo` can generate expressions (or the associated environments) that evaluate to a given value. `reordero` also takes the same types of expressions as `simpo`, and relates reordered expressions.

CCS Concepts: • **Computing methodologies → Computer algebra systems**; • **Software and its engineering → Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: relational programming, differentiation, simplification, miniKanren, Racket, Scheme

## 1 INTRODUCTION

Consider this calculus problem:

> Find two different polynomials, $f(x)$ and $g(x)$, and two different natural numbers $a$ and $b$, such that $f'(a) = b$, and $g'(b) = a$.

Differentiating polynomials is an easy calculus problem, but the problem above is more complicated because of the relationships between the polynomials, their derivatives, and the two natural numbers. We invite the reader to pause, try to find solutions to this problem, and to think about how these types of problems might be solved more generally.

We have developed a relational algebra system, *dxo*, that uses relational programming to solve problems like the one above. We show the `run` expression for solving this problem in Section 2. *dxo* is a collection of four main relations: `simpo` for simplification, `do` for differentiation, `evalo` for evaluation, and `reordero` for permuting arguments. Implementing *dxo* relationally makes it flexible. For example, the relation `do` can differentiate polynomials with respect to some variable. Since `do` is a relation, it can also integrate polynomials. Also, the expression to be differentiated and its derivative can both contain fresh logic variables. The relations `simpo`, `evalo`, and `reordero` similarly benefit from this flexibility.

Authors' addresses: Julie S. Steele, Georgetown Day School, USA, jshermansteele@gmail.com; William E. Byrd, University of Alabama at Birmingham, USA, webyrd@uab.edu.


miniKanren.org/workshop/2021/8-ART11

We assume the reader is familiar with core miniKanren [Byrd 2009; Byrd and Friedman 2006; Friedman et al. 2018] (==, `fresh`, `conde`, `run`), extended with disequality (`=/=`) and `absento` constraints [Byrd et al. 2012]. Detailed explanations to the core miniKanren language can be found in Friedman et al. [2018], Byrd [2009], and Byrd and Friedman [2006]. Descriptions of disequality and `absento` constraints can be found in Byrd et al. [2012] and Byrd et al. [2017].

Section 2 gives a high-level explanation of *dxo*, its uses, and its four main relations. Section 3 explains in detail the main relations. Section 4 discusses some open problems and possible future work. Section 5 discusses related work. We conclude the paper in Section 6. Appendix A contains the full implementation of *dxo*.

## 2 HIGH-LEVEL OVERVIEW

*dxo* is composed of four main relations, `simpo`, `do`, `evalo`, and `reordero`, that when used in combination can solve interesting differentiation math problems. Here are the four relations and their uses:

- **`(simpo comp simp)`** relates `comp` and `simp`, where `comp` can be any arithmetic expression and `simp` is an equivalent, fully simplified one;
- **`(do x expr deriv)`** relates a polynomial expression `expr` with its derivative `deriv`, where the derivative is with respect to `x`;
- **`(evalo env expr value)`** relates an expression `expr` with its value `value`, where each variable in `expr` is associated with a natural number by the environment `env`;
- **`(reordero e1 e2)`** relates two equivalent expressions, `e1` and `e2`, by changing the order of subexpressions in an addition or multiplication in any level of the other expression.

Figure 1 contains the grammar for expressions accepted by `simpo`, `evalo`, and `reordero`, and Figure 2 contains the grammar for polynomial expressions accepted as the `expr` for `do`. `deriv` is a subset. The implementation of *dxo* uses the relational arithmetic system created by Oleg Kiselyov, which is presented in Friedman et al. [2018] and Kiselyov et al. [2008].

⟨*dxo-expression*⟩ ::=
  | ⟨*numeral-or-variable*⟩
  | ‘`(+ ` ⟨*dxo-expression*⟩ . . . ‘`)`’
  | ‘`(* ` ⟨*dxo-expression*⟩ . . . ‘`)`’
  | ‘`(^ ` ⟨*dxo-expression*⟩ ⟨*dxo-expression*⟩ ‘`)`’

⟨*numeral-or-variable*⟩ :: = ⟨*tagged-numeral*⟩ | ⟨*tagged-variable*⟩

⟨*tagged-variable*⟩ ::= ‘`(var ` ⟨*symbol*⟩ ‘`)`’

⟨*tagged-numeral*⟩ ::= ‘`(num ` ⟨*numeral*⟩ ‘`)`’

⟨*numeral*⟩ ::= ‘`()`’ | ‘`(0 . ` ⟨*positive-numeral*⟩ ‘`)`’ | ‘`(1 . ` ⟨*numeral*⟩ ‘`)`’

⟨*positive-numeral*⟩ ::= ‘`(0 . ` ⟨*positive-numeral*⟩ ‘`)`’ | ‘`(1 . ` ⟨*numeral*⟩ ‘`)`’

Fig. 1. Grammar for general *dxo* expressions accepted by `simpo`, `evalo`, and `reordero`.

⟨*polynomial-expression*⟩ ::=
  | ⟨*numeral-or-variable*⟩
  | '(+ ' ⟨*polynomial-expression*⟩ ... ')'
  | '(* ' ⟨*polynomial-expression*⟩ ... ')'
  | '(^ ' ⟨*numeral-or-variable*⟩ ⟨*tagged-numeral*⟩ ')'

Fig. 2. Restricted grammar for polynomial expressions accepted by do.

Using the *dxo* relations, we can solve the problem proposed in the introduction: find two different polynomials, $f(x)$ and $g(x)$, and two different natural numbers $a$ and $b$, such that $f'(a) = b$, and $g'(b) = a$. We relate `f` and `g` with their derivatives, `fd` and `gd`, using `do`. Then we use `evalo` to evaluate these derivatives at `a` and `b` respectively (we do this by making one environment where `x` is `a` and one where `x` is `b`), and set the evaluation to `b` and `a` respectively. Last, we make sure `f` and `g` are different but both simplified and `a` and `b` are different.

```
(run 20 (f g envb enva)
  (fresh (b a gd fd)
    (=/= f g)
    (=/= b a)
    (== `((x . ,b)) envb)
    (== `((x . ,a)) enva)
    (do 'x f fd)
    (simpo f f)
    (do 'x g gd)
    (simpo g g)
    (evalo enva fd b)
    (evalo envb gd a)))
⇒
'(
  ...
  ((num _.0) (var x))                          ; f = c (where c is any natural number), g = x
   ((x)) ((x 1)))                              ; b = 0, a= 1
  ...
  ((var x) (^ (var x) (num (0 0 1 1)))         ; f = x, g = x^12
   ((x 1)) ((x 0 0 1 1)))                      ; b = 1, a = 12
  ...
)
```

Of the 20 outputs produced by the run expression, many had $b = 0$ so we only showed two. The first shown answer shows

$$\frac{d}{dx}[c] = 0, \text{where } c \text{ is any natural number and } \frac{d}{dx}[x] = 1, \text{where } x = 5$$

The second shown answer shows

$$\frac{d}{dx}[x] = 1, \text{where } x = 12 \text{ and } \frac{d}{dx}[x^{12}] = 12, \text{where } x = 1$$

The concise `run` expression solving this problem shows how *dxo* benefits from the expressiveness of relational programming.

We showed a combination of the four main *dxo* relations in solving the problem in the introduction. We will shortly demonstrate another way to combine these core relations in the definition of anydo, below.

Let's use do to differentiate the polynomial $x^3 + x^0$. Mathematically,

$$\frac{d}{dx}[x^3 + x^0] = (x^2 * 3) + 0$$

The equivalent call to do succeeds:

```
(do 'x
    '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))        ; x^3 + x^0 = expr
    '(+ (* (^ (var x) (num (0 1))) (num (1 1))) (num ()))    ; (x^2 * 3) + 0 = deriv
```

The derivative $(x^2 * 3) + 0$ is equivalent to $1 * x^2 * 3$, so we might expect the call

```
(do 'x
    '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))       ; x^3 + x^0 = expr
    '(* (num (1)) (^ (var x) (num (0 1))) (num (1 1))))     ; 1 * x^2 * 3 = deriv
```

to succeed. Unfortunately, this call fails because do requires the derivative to be in canonical form, $(x^2 * 3) + 0$ in this case. This means some mathematically correct expression and derivative pairs fail as arguments to do.

We created anydo to fix this problem. (anydo expr deriv x), like do, relates an expression with its derivative with respect to x, except anydo generalizes this to simplified, complicated, or reordered forms of expr and deriv. This relaxes the restriction on deriv being in canonical form, making running "backward" more convenient. Calling anydo with the same arguments as above succeeds:

```
(anydo '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))      ; x^3 + x^0 = expr
       '(* (num (1)) (^ (var x) (num (0 1))) (num (1 1)))     ; 1 * x^2 * 3 = deriv
       'x)
```

anydo is centered around a call to do with arguments similar to expr and deriv, ecomp and dcomp. expr and ecomp are similar in that they simplify to the same value, esimp, making them equivalent. anydo does the same for deriv and dcorder, with the additional step of reordering dcorder to be dcomp.

```
(define anydo
  (lambda (expr deriv x)
    (fresh (esimp dsimp ecomp dcomp dcorder)
      (simpo expr esimp)
      (simpo ecomp esimp)
      (do x ecomp dcomp)
      (reordero dcomp dcorder)
      (simpo dcorder dsimp)
      (simpo deriv dsimp))))
```

## 3 *DXO* IMPLEMENTATION WALK-THROUGH

In this section we explain in detail the four main relations in *dxo*.[1]

[1]We have released the *dxo* code under an MIT licence at https://github.com/JShermanSteele/dxo .

### 3.1 simpo

`(simpo comp simp)` relates `comp` and `simp`, where `comp` can be any arithmetic expression and `simp` is an equivalent, fully simplified one. *Simplified* means making all following simplifications:

- $v + 0 = v$;
- $v * 0 = 0$;
- $v * 1 = v$;
- $v^0 = 1$ $(v \neq 0)$;
- $v^1 = v$;
- $0^v = 0$ $(v \neq 0)$;
- and $1^v = 1$;

where $v$ is any expression. For example, let's simplify $0^5 + (2 * 1)$:

```
(run* (simp) (simpo `(+ (^ (num ()) (num (1 0 1) ))          ; 0^5 + 2 * 1 = comp
                        (* (num (0 1)) (num (1))))
                    simp))
⇒
'((num (0 1)))                                               ; 2 = simp
```

`simpo` has base cases of `(== comp simp)` for `comp` and `simp` being the same number or variable. The three non-base cases, addition, multiplication, and exponentiation, deeply recursively simplify sub-expressions by checking for those simplifyable cases.

If `comp` is ground, `(simpo comp simp)` will either succeed exactly once or fail because there is at most one way to simplify any concrete expression, and `simpo` has no overlapping cases when running "forwards". If `comp` is fresh, then `simpo` could succeed, but if it is an impossible relation, simpo can try longer and longer `comp`s, never succeeding, and loop forever.

An example of these behaviors is that running `(simpo comp simp)` with `comp` as $1^1$ and `simp` as a logic variable succeeds because $1^1$ simplified is 1. Running with `comp` as a logic variable and `simp` as (the unsimplified) $1^1$ diverges, searching for a `comp` forever.

```
(run* (simp) (simpo '(^ (num (1)) (num (1))) simp))       ; 1^1 = comp
⇒
'((num (1)))

(run 1 (comp) (simpo comp '(^ (num (1)) (num (1)))))       ; 1^1 = simp
```

There is also a case with a ground `comp` and partially ground `simp` in which `simpo` will fail. If `comp` is $1^1$, which simplifies to 1, and `simp` is any addition expression, which may include fresh variables, `simpo` will fail finitely.

```
(run* (q) (simpo `(^ (num  (1)) (num (1)))            ; 1^1 = comp
                 `(+ . ,q)))                          ; simp is some addition expression
⇒
'()
```

### 3.2 do

`(do x expr deriv)` relates a polynomial expression `expr` with its derivative `deriv`, with respect to `x`. For example, running `do` with `expr` and `deriv` fresh finds integral/derivative pairs:

```
(run 24 (expr deriv) (do 'x expr deriv))
⇒
‘(...
```

```
((^ (var x) (num (0 1)))                          ; x^2 = expr
 (* (^ (var x) (num (1))) (num (0 1))))           ; x^1 * 2 = deriv
...
((^ (var x) (num (1 _.0 . _.1)))                    ; x^a where a is odd = expr
 (*                                                 ; x^(a-1) * a = deriv
   (^ (var x) (num (0 _.0 . _.1)))
   (num (1 _.0 . _.1))))
...
((* (^ (var x) (num ())) (^ (var x) (num ())))     ; x^0 * x^0 = expr
 (+                                                ; 0 * x^0 + x^0 * 0 = deriv
   (* (num ()) (^ (var x) (num ())))
   (* (^ (var x) (num ())) (num ()))))
```

The best way to understand do is as a case analysis on expr, which is either a variable, a number, an exponentiation, an addition, or a multiplication.

Since the derivative of a sum is the sum of the derivatives of its parts, when expr and deriv are sums, the sub-expressions of expr and deriv are pair-wise related using do. Since the sum can have any positive number of terms, a helper relation, map-do-o, relates each pair in the sums.

If expr is a multiplication, do must use the multiplication rule that

$$\frac{d}{dx}(ab) = \frac{da}{dx} * b + a * \frac{db}{dx},$$

and recur down the list of sub-expressions being multiplied. To improve the efficiency this process, we wrote a helper relation, multruleo, that relates the list of sub-expressions being multiplied and the multiplication's derivative. If the list has length greater than one, multruleo separates the first term $e1$ from the rest, $e2 * e3 * \ldots$. Applying the multiplication rule to $e1 * e2 * e3 * \ldots$ yields $\frac{d}{dx}[e1] * e2 * e3 * \ldots + e1 * \frac{d}{dx}[e2 * e3 * \ldots]$, which is recursive with multruleo because $\frac{d}{dx}[e2 * e3 * \ldots]$ is the related derivative argument to multruleo with $e2 * e3 * \ldots$ as the first argument. This is conde the clause in do for multiplication.

```
((fresh (l e)
   (== expr `(* . ,l))
   (letrec ((multruleo
             (lambda (l dd)
               (fresh (e e* d d* a b)
                 (conde
                   [(== l `(,e))(do x e dd)]
                   [(== l `(,e . ,e*))
                    (== e* `(,a . ,b))
                    (== dd `(+ (* ,d . ,e*) (* ,e ,d*)))
                    (do x e d)
                    (multruleo e* d*)])))))
     (multruleo l deriv))))
```

We could recur through the multiplication by recurring with every shorter multiplication as an argument to do, but our approach is simpler because it does not exit multruleo while reccuring through expr's multiplication.

If expr is an exponentiation, the second subexpression in the exponent must be a number by do's grammar, so $\frac{d}{dx}[x^n] = (n * (x^{n-1}))$ where $n$ is any number. There are three clauses for constants, $\frac{d}{dx}[x^0] = 0$, $\frac{d}{dx}[n^m] = 0$, and $\frac{d}{dx}[n] = 0$ where $n$ and $m$ are any numbers so long as they both are not 0. Finally, the derivative of just x is one.

, Vol. 1, No. 1, Article 11. Publication date: August 2021.

Since do orders deriv a certain way (for example, $x^2 * 6$ instead of $6 * x^2$), some integratable derivs will fail. This is why do should be used with reordero. For example,

```
(run 1 (expr)
  (do 'x expr '(* (^ (var x) (num (1))) (num (0 1)))) ; x^1 * 2 = deriv
⇒
'((^ (var x) (num (0 1))))                            ; x^2 = expr
```

produces an answer, but switching deriv's multiplication order diverges:

```
(run 1 (expr)
  (do 'x expr '(* (num (0 1)) (^ (var x) (num (1))))) ; 2 * x^1 = deriv
```

Similarly to simpo, do succeeds exactly once or fails running "forwards" when expr and x are ground. With expr fresh, do can succeed or can loop infinitely just like simpo.

### 3.3 evalo

evalo evaluates, and is useful for solving equations.(evalo env expr value) relates an expression expr with its value value, where each variable in expr is associated with a natural number by the environment env. For example we can look for expressions that evaluate to 8 in an environment that binds z to 2:

```
(run 200 (expr) (evalo `((z . (0 1))) expr '(0 0 0 1)))      ; z= 2, value= 8
⇒
‘(...
  (* (var z) (num (0 0 1)))                                   ; z * 4 = expr
  ...
  (^ (num (0 1)) (num (1 1)))                                 ; 2^3 = expr
  ...
  (+ (var z) (num ()) (num (0 1 1))))                          ; z + 0 + 6 = expr
```

evalo deeply recurs through expr, evaluating tagged little-endian binary lists into miniKanren numbers. For evalo's base cases when expr is a variable or number, value is the variable's value from env or the miniKanren number respectively.If expr is an addition, multiplication, or an exponentiation, then evalo relates the first term with its value evc, the rest with its value evrest, and adds, multiplies, or exponentiates evc and evrest to obtain value.The evalo code for addition does this:

```
((== `(+ ,c . ,rest) expr)
 (conde
   ((== '() rest) (evalo env c value))
   ((=/= '() rest)
   (evalo env c evc)
   (evalo env `(+ . ,rest) evrest)
   (pluso evc evrest value))))
```

This will recur through rest and sum all the parts to relate expr with value.

An interesting use of evalo is to solve algebra problems by making env fresh, for example looking for Pythagorean triples. So we set up $x^2 + y^2 = z^2$ and also a $z * z =$ *z-squared* relation to make sure that z is a natural number to find the classic Pythagorean triple!

```
(run 1 (env)
  (fresh (xv yv zv z2v)
    (== `((x . ,xv) (y . ,yv) (z . ,zv) (z-squared . ,z2v)) env)
    (evalo env `(+ (^ (var x) (num (0 1))) (^ (var y) (num (0 1)))) z2v)
```

```
    (*o zv zv z2v)))
⇒
'(((x) (y) (z) (z-squared)))                                          ; x= 0, y= 0, z= 0
```

Not what we wanted, alas. Setting non-zero constraints, though, produces:

```
(run 1 (env)
  (fresh (xv yv zv z2v)
    (poso xv)
    (poso yv)
    (poso zv)
    (== `((x . ,xv) (y . ,yv) (z . ,zv) (z-squared . ,z2v)) env)
    (evalo env `(+ (^ (var x) (num (0 1))) (^ (var y) (num (0 1)))) z2v)
    (*o zv zv z2v)))
⇒
'(((x 1 1)                                                            ; x= 3
   (y 0 0 1)                                                          ; y= 4
   (z 1 0 1)                                                          ; z= 5
   (z-squared 1 0 0 1 1)))                                            ; z^2 = 25
```

which is a 3-4-5 right triangle.

If either `env` or `expr` is fresh, `evalo` can loop forever, trying more and more complicated `env`s or `expr`s. If both `env` or `expr` are ground, then `evalo` will terminate since `evalo` runs simply forwards in this case.

### 3.4 reordero

`(reordero e1 e2)` relates two equivalent expressions, `e1` and `e2`, by changing the order of subexpressions in an addition or multiplication in any level of the other expression. It is useful for taking integrals with `do`. We can use `reordero` to find all reorderings of an expression:

```
(run* (e2) (reordero `(+ (num (1)) (* (num (0 1)) (num (1 1)))) e2))   ; 1 + 2 * 3 = e1
⇒
'((+ (num (1)) (* (num (0 1)) (num (1 1))))                           ; 1 + 2 * 3 = e2
  (+ (* (num (0 1)) (num (1 1))) (num (1)))                           ; 2 * 3 + 1 = e2
  (+ (num (1)) (* (num (1 1)) (num (0 1))))                           ; 1 + 3 * 2 = e2
  (+ (* (num (1 1)) (num (0 1))) (num (1))))                          ; 3 * 2 + 1 = e2
```

`(reordero e1 e2)` relates `e1` and `e2` by having the same outer operation (addition, multiplication, or exponentiation). For addition and multiplication the code is,

```
((fresh (o e1* e2*)
   (== `(,o . ,e1*) e1)
   (== `(,o . ,e2*) e2)
   (typeo o '+or*)
   (reorderitemso e1* e2*)))
```

where `e1*` and `e2*` are permutations of each other. `reorderitemso` checks that `e1*` and `e2*` have the same length, and then calls `reorderinnero` on them. We created `reorderitemso` to improve speed and divergence behavior of `reordero` by requiring `e1*` and `e2*` have the same length before considering the relations in `reorderinnero`. `reorderinnero` relates permuted lists at any depth. To deeply reorder, `reorderinnero` calls `reordero` on the corresponding sub-expressions for `e1*` and `e2*`.

```
(define reorderinnero
  (lambda (e1* e2*)
    (fresh (c1 rc1 rest1 rest2)
      (conde
        ((== '() e1*) (== '() e2*))
        ((== `(,c1 . ,rest1) e1*)
         (removeo rc1 e2* rest2)
         (reorderinnero rest1 rest2)
         (reordero c1 rc1))))))
```

`reordero` greatly reduces infinite loops because `reorderitemso` checks that its arguments are the same length. This check keeps `e1` and `e2` the same structure and length at every depth, keeping the search finite. Checks like this would be useful to add other places in *dxo* to reduce divergence.

## 4 OPEN PROBLEMS

`dxo` could be improved by expanding the grammars, improving the speed and termination, and using automatic differentiation. We would like to add expressions like $2^x$, $sin(x)$, and multiple variables. Currently, *dxo* searches inefficiently, especially combinations of relations like `anydo`, so we would like to speed these up. We would also like to make more calls terminate. We are interested in improving `simpo`, possibly implementing Knuth-Bendix Completion[Dick 1991] relationally. We have done some preliminary work making a relation to replace `do` that automatic differentiates forwards and backwards.

## 5 RELATED WORK

Expresso [Schünemann 2017] is a computer algebra system written in Clojure using the miniKanren-inspired core.logic library. expresso's original intent was to be relational, but the author made it non-relational to include more advanced features.[Schünemann 2020] Like *dxo*, it includes algebraic simplification, differentiation, and evaluation. Beyond *dxo*, it includes rewriting in normal form and expressions like *sin*.

The Reduce-Algebraic-Expressions system in Prolog [Jasoria 2019] is similar to `simpo`, using certain simplification identities. simplifies expressions like $((x + x)/x) * (y + y - y) \Rightarrow 2 * y$. It can make simplifications like $x + x \Rightarrow 2 * x$ which `simpo` cannot since `simpo` currently only includes simplification rules involving 0 and 1. The Reduce-Algebraic-Expressions system is not relational.

## 6 CONCLUSION

*dxo* applies relational programming to algebra and differentiation. It can differentiate, integrate, simplify, complicate, evaluate, create, and reorder. *dxo* can concisely represent non-trivial math problems and find solutions. *dxo* is a foundation for future exploration of relational programming in algebra.

## ACKNOWLEDGMENTS

We are grateful for the work of all the relational programmers whose work we have built upon. Our work would not have existed without the efforts of Dan Friedman and Oleg Kiselyov. In addition, we would like to thank Maik Schünemann for explaining his work. We would like to thank Brandon T. Willard and Alan T. Sherman for comments on a draft of this paper and sharing ideas with us. Finally, we thank the anonymous reviewers for their many helpful comments.

## REFERENCES

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

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

William E. Byrd and Daniel P. Friedman. 2006. From Variadic Functions to Variadic Relations: A miniKanren Perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop (University of Chicago Technical Report TR-2006-06)*, Robby Findler (Ed.). 105–117.

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* (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

A. J. J. Dick. 1991. An Introduction to Knuth–Bendix Completion. *Comput. J.* 34, 1 (01 1991), 2–15. https://doi.org/10.1093/comjnl/34.1.2 arXiv:https://academic.oup.com/comjnl/article-pdf/34/1/2/1181698/340002.pdf

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (2nd ed.). The MIT Press, Cambridge, MA, USA.

Shruti Jasoria. 2019. Reduce-Algebraic-Expressions. https://github.com/SJasoria/Reduce-Algebraic-Expressions. Accessed: 2020-07-14.

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 Symposium on Functional and Logic Programming (LNCS, Vol. 4989)*, Jacques Garrigue and Manuel Hermenegildo (Eds.). Springer, 64–80.

Maik Schünemann. 2017. expresso. https://github.com/clojure-numerics/expresso. Accessed: 2020-07-14.

Maik Schünemann. 2020. private correspondence.

## A FULL IMPLEMENTATION OF *DXO*

```
(require "faster-miniKanren/mk.rkt")
(require "faster-miniKanren/numbers.rkt")

;defines ^ as expt
(define ^ (lambda (a b) (expt a b)))

;defines ZERO and ONE
(define ZERO `(num ,(build-num 0)))
(define ONE `(num ,(build-num 1)))

;exponent: (^ a b)=c
(define expo
  (lambda (a b c)
    (fresh (bm1 rec)
      (conde
        ((== (build-num 0) b) (== (build-num 1) c))
        ((=/= (build-num 0) b)
         (pluso bm1 (build-num 1) b)
         (expo a bm1 rec)
         (*o a rec c))))))

;atom?
(define atom?
  (lambda (expr)
    (cond
      ((list? expr) #f)
```

```
      ((null? expr) #f)
      (else #t))))

;atom, null, or list
(define typeo
  (lambda (expr answer)
    (fresh (a b)
      (conde
        ((== `(,a . ,b) expr) (== 'list answer) (=/= 'num a) (=/= 'var a))
        ((== `() expr) (== 'null answer))
        ((== `(num ,a) expr) (== 'atom answer))
        ((== `(var ,a) expr) (== 'atom answer))
        ((== '+ expr) (== '+or* answer))
        ((== '* expr) (== '+or* answer))))))

;miniKanren number
(define numo
  (lambda (n)
    (fresh (b rest)
      (conde
        ((== '() n))
        ((== `(,b . ,rest) n)
         (conde
           ((== 1 b))
           ((== 0 b)))
         (numo rest))))))

;empty env
(define empty-env `())

;ext-env
(define ext-env
  (lambda (x v env)
    (cons `(,x . ,v) env)))

;lookupo
(define lookupo
  (lambda (x env v)
    (fresh (env* y w)
      (conde
        ((== `((,x . ,v) . ,env*) env))
        ((== `((,y . ,w) . ,env*) env) (=/= y x) (lookupo x env* v))))))

;unbuild-numinner to every element and if list, then to list
(define unbuild-numhelper
  (lambda (expr)
```

```
    (cond
      ((null? expr) '())
     ((list? (car expr)) (cons (unbuild-numinner (car expr)) (unbuild-numhelper (cdr expr))))
      (else (cons (car expr) (unbuild-numhelper (cdr expr)))))))

;calls unbuld-numinner for every answer in miniKanren
(define unbuild-num
  (lambda (expr)
    (cond
      ((null? expr) '())
      (else (cons (unbuild-numinner (car expr)) (unbuild-num (cdr expr)))))))
```

```
;undoes build-num by calling unbinary
(define unbuild-numinner
  (lambda (expr)
    (match expr
      [`() `()]
      [`(num ,b) `(num ,(unbinary b 1))]
      [`(var ,b) `(var ,b)]
      [`(+ ,e . ,e*) (unbuild-numhelper `(,e . ,e*))]
      [`(* ,e . ,e*) (unbuild-numhelper `(,e . ,e*))]
      [`(^ ,e . ,e*) (unbuild-numhelper `(,e . ,e*))])))

;helper for unbuild-numinner that goes from binary to base 10
(define unbinary
  (lambda (expr n)
    (cond
      ((null? expr) 0)
      ((atom? expr) expr)
      ((equal? (car expr) 1) (+ n (unbinary (cdr expr) (* 2 n))))
      ((equal? (car expr) 0) (unbinary (cdr expr) (* 2 n))))))


;l contains item
(define membero
  (lambda (item l)
    (fresh (a rest)
      (conde
        ((== `(,item . ,rest) l))
        ((== `(,a . ,rest) l) (=/= item a) (membero item rest))))))

;l does not contain item
(define notmembero
  (lambda (item l)
    (fresh (a rest)
      (conde
        ((== '() l))
        ((== `(,a . ,rest) l)
         (=/= a item)
         (notmembero item rest))))))

;removes item from l
(define removeo
  (lambda (item contain removed)
    (fresh (rest rest2 c)
      (conde
        ((== `(,item . ,rest) contain)
```

```
          (== rest removed))

         ((== `(,c . ,rest) contain)
          (=/= item c)
          (== `(,c . ,rest2) removed)
          (removeo item rest rest2))))))


"SIMPLIFY";-----------------------------------------------------------

(define simpo
  (lambda (comp simp)
    (fresh ()
      (conde
        ((fresh (n)
           (== `(num ,n) comp)
           (== comp simp)))
        ((fresh (v)
           (== `(var ,v) comp)
           (== comp simp)))

        ((fresh (e1 e2 s1 s2)
           (== `(^ ,e1 ,e2) comp)
           (conde
             ((== ONE s1) (== ONE simp))
             ((== ZERO s1) (=/= ZERO s2) (== ZERO simp))
             ((=/= ZERO s1) (== ZERO s2) (=/= ONE s1) (== ONE simp))
             ((=/= ZERO s1) (=/= ONE s1) (== ONE s2) (== s1 simp))
             ((== `(^ ,s1 ,s2) simp)
              (=/= ONE s1)
              (=/= ONE s2)
              (=/= ZERO s1)
              (=/= ZERO s2)))
           (simpo e1 s1)
           (simpo e2 s2)))

        ((fresh (e e* s temp t* n v)
           (== `(* ,e . ,e*) comp)
           (conde
             ((== '() e*) (simpo e simp))
             ((== ZERO s)(=/= '() e*)(== ZERO simp))
             ((== ONE s)(=/= '() e*) (simpo `(* . ,e*) simp))
             ((=/= ONE s)
              (=/= ZERO s)
              (=/= '() e*)
```

```
            (conde
              ((== ZERO temp) (== ZERO simp))
              ((== ONE temp) (== s simp))
              ((== `(^ . ,n) temp) (== `(* ,s ,temp) simp))
              ((== `(+ . ,n) temp) (== `(* ,s ,temp) simp))
              ((== `(num ,n) temp) (=/= ZERO temp)
               (=/= ONE temp) (== `(* ,s ,temp) simp))
              ((== `(var ,v) temp) (== `(* ,s ,temp) simp))
              ((==`(* . ,t*) temp) (== `(* ,s . ,t*) simp)))
            (simpo `(* . ,e*) temp)))
         (simpo e s)))

      ((fresh (e e* s temp t* n v)
         (== `(+ ,e . ,e*) comp)
         (conde
           ((== '() e*) (simpo e simp))
           ((== ZERO s)(=/= '() e*)(simpo `(+ . ,e*) simp))
           ((=/= ZERO s)
            (=/= '() e*)
            (conde
              ((== ZERO temp) (== s simp))
              ((== `(^ . ,n) temp) (== `(+ ,s ,temp) simp))
              ((== `(* . ,n) temp) (== `(+ ,s ,temp) simp))
              ((== `(num ,n) temp) (=/= ZERO temp) (== `(+ ,s ,temp) simp))
              ((== `(var ,v) temp) (== `(+ ,s ,temp) simp))
              ((== `(+ . ,t*) temp) (== `(+ ,s . ,t*) simp)))
            (simpo `(+ . ,e*) temp)))
         (simpo e s)))))))


"DERIVATIVE";------------------------------------------------------------

;takes derivative
(define do
  (lambda (x expr deriv)
    (fresh ()
      (symbolo x)
      (conde
        ((fresh (d* e* a b c d)
           (== expr `(+ . ,e*)) (== e* `(,a . ,b))
           (== deriv `(+ . ,d*)) (== d* `(,c . ,d))
           (samelengtho e* d*)
           (map-do-o x e* d*)))
```

```
((fresh ()
   (== expr `(^ (var ,x) (num ,(build-num 0))))
   (== deriv ZERO)))

((fresh (l e)
   (== expr `(* . ,l))
   (letrec ((multruleo
             (lambda (l dd)
               (fresh (e e* d d* a b)
                 (conde
                   [(== l `(,e))
                    (do x e dd)]
                   [(== l `(,e . ,e*))
                    (== e* `(,a . ,b))
                    (== dd `(+ (* ,d . ,e*) (* ,e ,d*)))
                    (do x e d)
                    (multruleo e* d*)])))))
     (multruleo l deriv))))

((fresh (int intm1)
   (== expr  `(^ (var ,x) (num ,int)))
   (== deriv `(* (^ (var ,x) (num ,intm1)) (num ,int)))
   (minuso int (build-num 1) intm1)))

((fresh (int1 int2)
   (== expr `(^ (num ,int1) (num ,int2)))
   (== deriv ZERO)
   (conde
     ((poso int1))
     ((== ZERO int1)(poso int2)))))

((fresh ()
   (== expr `(var ,x))
   (== deriv ONE)))

((fresh (int)
   (== expr `(num ,int))
   (== deriv ZERO))))))
```

```
;maps do relation
(define map-do-o
  (lambda (x expr* output)
    (fresh (e* e out out*)
      (conde
        [(== expr* '()) (== output '())]
        [(== expr* `(,e . ,e*))
         (== output `(,out . ,out*))
         (do x e out)
         (map-do-o x e* out*)]))))

"EVALUATE";-------------------------------------------------------------

;evaluater
(define evalo
  (lambda (env expr value)
    (fresh (m x c a b rest evc evrest eva evb)
      (conde
        ((== `(var ,x) expr) (lookupo x env value))
        ((== `(num ,m) expr) (numo m) (== m value))
        ((== `(+ ,c . ,rest) expr)
         (conde
           ((== '() rest) (evalo env c value))
           ((=/= '() rest)
            (evalo env c evc)
            (evalo env `(+ . ,rest) evrest)
            (pluso evc evrest value))))
        ((== `(* ,c . ,rest) expr)
         (conde
           ((== '() rest) (evalo env c value))
           ((=/= '() rest)
            (evalo env c evc)
            (evalo env `(* . ,rest) evrest)
            (*o evc evrest value))))
        ((== `(^ ,a ,b) expr)
         (evalo env a eva)
         (evalo env b evb)
         (expo eva evb value))))))

"REORDER";_______________________________________________________________

;another option instead of using reordero is to always enter expressions in the same right order
;reorders expression deeply, reordering any + and * expressions
```

```
(define reordero
  (lambda (e1 e2)
    (fresh ()
      (conde
        ((== e1 e2) (typeo e1 'atom))
        ((fresh (o e1* e2*)
           (== `(,o . ,e1*) e1)
           (== `(,o . ,e2*) e2)
           (typeo o '+or*)
           (reorderitemso e1* e2*)))
        ((fresh (a1 b1 a2 b2)
           (== `(^ ,a1 ,b1) e1)
           (== `(^ ,a2 ,b2) e2)
           (reordero a1 a2)
           (reordero b1 b2)))))))


;permutes a list by calling reorderinnero, and calls reordero on the items in the list deeply
(define reorderitemso
  (lambda (e1* e2*)
    (fresh ()
      (samelengtho e1* e2*)
      (reorderinnero e1* e2*))))



;permutes and calls reordero on the items, helper for reorderitemso
(define reorderinnero
  (lambda (e1* e2*)
    (fresh (c1 rc1 rest1 rest2)
      (conde
        ((== '() e1*)(== '() e2*))
        ((== `(,c1 . ,rest1) e1*)
         (removeo rc1 e2* rest2)
         (reorderinnero rest1 rest2)
         (reordero c1 rc1))))))
```

```
"ANYDO";_____________________________________________________________________________

(define anydo
  (lambda (expr deriv x)
    (fresh (ecomp dcomp esimp dsimp dcorder)
      (project (expr deriv)
        (if (var? expr)
            (fresh ()
              (simpo deriv dsimp)
              (simpo dcorder dsimp)
              (reordero dcomp dcorder)
              (do x ecomp dcomp)
              (simpo ecomp esimp)
              (simpo expr esimp))

            (fresh ()
              (simpo expr esimp)
              (simpo ecomp esimp)
              (do x ecomp dcomp)
              (reordero dcomp dcorder)
              (simpo dcorder dsimp)
              (simpo deriv dsimp)))))))


(define doitallevalo
  (lambda (ieval inte deriv deval x env)
    (fresh (icomp dcomp isimp dsimp dcorder)
      (evalo env inte ieval)
      (evalo env deriv deval)
      (do x icomp dcomp)
      (reordero dcomp dcorder)
      (simpo deriv dsimp)
      (simpo dcorder dsimp)
      (simpo inte isimp)
      (simpo icomp isimp))))
```