# Think of a Title

Ekaterina Verbitskaia
kajigor@gmail.com
JetBrains
Belgrade, Serbia

Daniil Berezun
JetBrains
Amsterdam, Netherlands
example@example.com

Dmitry Boulytchev
SPbSU, Huawei
Saint Petersburg, Russia

## Abstract

Languages in the Kanren family strive to bridge the gap between logic and general-purpose mainstream programming. Logic programming comes with an overhead such as keeping track of substitutions of logic variables and unifying terms. However, in many practical applications there is no need to bear all that overhead, and thus we should not. Ideally, we should be able to automatically rewrite a relation into a function which computes the outputs but omits most unnecessary overhead. In this paper we present a method to translate miniKanren relations into pure functions in continuation passing stule. The project is at an early stage, but it is promising: the functions run much faster than the original miniKanren code.

***Keywords:*** relational programming, functional programming, cps

**ACM Reference Format:**
Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2018. Think of a Title. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Implementing a program is often significantly easier than its inversion. For example, integer multiplication is much simpler than factoring, while program evaluation is easier than program generation. Although inversion is undecidable, there are approaches capable of inversing a computation in some cases, notably, universal resolving algorithm cite Gluck, logic and relational programming. Inversion comes with a lot of overhead which may be reduced in some circumstances.

One source of overhead in relational programming comes from *unification* — the basic operation which is at the core of miniKanren. Unification involves traversing terms being unified along with a list of substitutions and doing occurs-check all of which may be redundant when there is a specific execution *direction* in mind. Directions fix at compile-time which arguments of a relation are always going to be known and ground at runtime. Having this information, it is possible to specialize a relation for the direction cite Verbitskaia and get rid of some of the overhead. In this case, unifications may prove to be redundant and be replaced with much simpler pattern-matching and equality checks.

In this paper we present a scheme of translation of miniKanren programs into a host functional programming language as a sequence of examples. Examples start from the simplest translations and evolve to introduce different features of miniKanren which influence translation. Currently translation is not automated: everything is done manually. We believe the translation can be semi-automated, leaving some decisions up to a programmer. Although this project is at the early state, evaluation demonstrates its usefulness by significantly speeding up such programs as computing a topological sorting of a graph and generating logic formulas which evaluate to the given value.

## 2 Preliminaries

In this section we remind the reader some basics of miniKanren. Usually, miniKanren is implemented as an embedded language and consists of a small set of basic combinators: disjunction and conjunction of goals, unification of terms and a helper to introduce fresh variables. Relations can be defined and called in the same manner as functions of the host language. Each miniKanren goal maps a variable substitution into a stream of substitutions. Computation may fail, producing an empty stream, or succeed and produce a non-empty stream of substitutions. In order to assure completeness of search, miniKanren usually implements conjunctions as monadic `bind` on streams and disjunctions as `mplus` which interleaves streams cite Kiselyov.

We use the following syntactic conventions. We denote conjunctions as a right-associative binary relation $\wedge$. In place of disjunctions we use **conde** with a list of miniKanren goals which is just a syntactic sugar. Unifications between two terms are denoted by a not associative binary relation $\equiv$. Several fresh variables may be introduced to the scope by a construction **fresh**. We use superscript $^o$ to differentiate miniKanren relations from functions written in a host language.



*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

```
let rec addᵒ x y z = conde [
  (x ≡ 0 ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧
     z ≡ S z' ∧
     addᵒ x' y z') ) ]
```

**Listing 1.** Addition relation

Consider an addition relation $\text{add}^o$ x y z which specifies that z equals to x + y (Listing 1). This relation has three arguments: x, y and z, and is comprised of a single **conde** with two branches. The first **conde** branch is a conjunction of two unifications: x with a term 0 and y with z. The second **conde** branch introduces fresh variables x' and z' and follows with a conjunction of two unifications and a recursive relation call.

One can *run* a relation in some direction by passing it *input* arguments. For example, executing $\text{add}^o$ (S 0) 0 z finds the sum of the first two arguments and maps z to the sum S 0. We can also provide only the last argument: $\text{add}^o$ x y (S 0), which can be considered as an inversion of addition. This computes all pairs of Peano numbers (x, y) which sum up to the given value z = S 0, namely (0, S 0) and (S 0, 0). Moreover, we can pass as input arguments not only *ground terms* but terms which contain fresh variables, such as $\text{add}^o$ x (S y) z. Executing this relation finds all triples (x, y, z) such that x + (y + 1) = z. Running in some directions can fail. For example $\text{add}^o$ (S x) y 0 may never succeed, since (1 + x) + y can never be equal to 0.

There exists a multitude of different directions for each relation. In this paper we only consider directions in which input arguments are ground, i.e. do not contain any fresh variables, we will call them *principal directions*. We denote a principal direction by the name of a relation followed by specification of its arguments: in place of each argument we write either **in** when the argument is input or out if it is output. There are 8 principal directions for $\text{add}^o$ x y z:

- three directions with one input: $\text{add}^o$ **in** out out, $\text{add}^o$ out **in** out, and $\text{add}^o$ out out **in**;
- three directions with two inputs: $\text{add}^o$ **in** **in** out, $\text{add}^o$ **in** out **in**, $\text{add}^o$ out **in** **in**;
- one direction which does not have any input arguments: $\text{add}^o$ out out out;
- and one direction in which all arguments are input: $\text{add}^o$ **in** **in** **in**;

When all arguments of a relation are input arguments, it serves as a predicate, while passing no arguments corresponds to the generation of all valid values for all arguments of a relation.

```
addXY :: Nat → Nat → Nat
addXY x y =
  case x of
    0 → y
    S x' → S (addXY x' y)
```

**Listing 2.** Function for addo **in** **in** out direction

```
addXY :: Nat → Nat → Stream Nat
addXY x y =
  case x of
    0 → return y
    S x' → S <$> addXY x' y
```

**Listing 3.** Using streams in a function for addo **in** **in** out direction

## 3 Conversion by Examples

In this section we gradually introduce our conversion by means of a set of examples. Each direction we consider illustrates some aspect of the conversion.

### 3.1 Basic Conversion

Consider $\text{add}^o$ **in** **in** out. This direction can be expressed as a function presented in Listing 2. The relation $\text{add}^o$ x y z has two branches in a **conde**: one unifies x with 0 and the other — with S x'. Since we know that x is always ground in this direction, we can replace unifications with a pattern-matching.

When x unifies with 0, the rest of the **conde** branch is the unification y ≡ z. This unification means that the output value of the direction is equal to y. Thus we can just return y as the result when x is pattern-matched with 0.

Now consider the **conde** branch in which x unifies with S x' where x' is a fresh variable. The variable x in this direction is always ground, thus x' is also ground after unification. This means, that the recursive call $\text{add}^o$ x' y z' is done in the direction $\text{add}^o$ **in** **in** out and can be translated into a recursive call to the function addXY. This recursive call computes the value of z', making it ground. The only thing that is left is to apply the constructor S to the result of the recursive call, since z ≡ S z'.

### 3.2 Nondeterministic Directions

Running a relation in a given direction may succeed with one *or more* possible answers or it may fail, i.e. it may run nondeterministically. It is natural to implement nondeterminism by using streams which are at the core of miniKanren. Any deterministic directions can be trivially transformed to using streams as shown in Listing 3. One example in which there are multiple answers is $\text{add}^o$ out out **in**. This direction corresponds to finding all pairs of numbers which sum up to the given z and can be implemented as shown in Listing 4.

```
addZ :: Nat → Stream (Nat, Nat)
addZ z =
  return (0, z) `mplus`
  case z of
    0 → Empty
    S z' → do
      (x', y) ← addZ z'
      return (S x', y)
```

**Listing 4.** Function for addo out out **in** direction

```
addX :: Nat → Stream (Nat, Nat)
addX x =
  case x of
    0 → do
      z ← genNat
      return (z, z)
    S x' → do
      (y, z') ← addX x'
      return (y, S z')

genNat :: Stream Nat
genNat = Mature 0 (S <$> genNat)
```

**Listing 5.** Function for addo **in** out out direction

In this case, the input variable z does not discriminate two branches of **conde**. Although the second branch of **conde** unifies z with a term S z', the first branch unifies z with a free variable y. In this case we need to consider the two branches independently and then combine the results into a new stream.

The first **conde** branch produces a single answer in which x is 0, and y is equal to z. This single result is then wrapped into a singleton stream.

The second **conde** branch succeeds only if z is a successor of another value, thus when z is a 0 we should fail. We express this by pattern-matching on z and returning an Empty stream when z is 0. Otherwise z unifies with S z', which means that z' is ground, and the recursive call to the relation is done in the direction add$^o$ out out **in**. This recursive call returns a stream of pairs (x', y), and by applying the constuctor S to x' we get the value of x.

The two translated **conde** branches are then combined by using `mplus`: the same combinator which is used in miniKanren for disjunctions. We use do-notation when translating the second branch of **conde** which is just a syntactic sugar for the monadic bind operation >>=. Binds implement conjunctions in miniKanren and it is no surprise they fit well into the functional implementation.

```
addXYZ :: Nat → Nat → Nat → Stream ()
addXYZ x y z =
  case x of
    0 | y == z → return ()
      | otherwise → Empty
    S x' →
      case z of
        0 → Empty
        S z' → addXYZ x' y z'
```

**Listing 6.** Function for addo **in in in** direction

### 3.3 Free Variables in Answers

In some directions, there are infinitely many answers, such as in add$^o$ **in** out out. When only the second argument is known, the answer is all pairs of numbers (y, z) which satisfy x + y = z. In miniKanren, this is expressed with help of free variables. Say x is S 0, then the stream of answers is represented as (_ .0, S _ .0). This means that whatever the value of y is, z is just its successor. In our paper we only consider scenarios when the answers are ground, so we expect the stream of answers to be (0, S 0), (S 0, S (S 0)), ... . To do it, we need to systematically generate a stream of ground values for y and z. Currently, we leave the generation up to the user, but generators may be automatically created from their types.

Listing 5 shows the functional implementation of the direction add$^o$ **in** out out. This direction is very similar to the add$^o$ **in in** out: we can pattern match on x, call the same function recursively in the second **conde** branch and construct the resulting value for z by applying the constructor S. But in the case when x is 0, the only thing we know about the values of y and z is that they are equal. In this case can generate a stream of all Peano numbers for z (or y) and use them in the returned result.

The generation of all numbers is done as shown in Listing 5, function genNat. The only thing one should be careful about, is to ensure lazy generation of the values, especially in case of an eager host language, such as OCaml.

### 3.4 Predicates

When all arguments of a relation are input, the direction serves as a predicate. Consider add$^o$ **in in in** and its functional implementation in Listing 6. In this case there is no actual answers we should return: the only thing that matters is whether the computation succeeded or failed. Failure is expressed with an empty stream and success — as a singleton stream with a unit value.

All arguments of the relation in this direction are ground. This means, that all unification can be replaced with either pattern-matching or simple equality check. When translating the first **conde** branch we pattern match on x, and then check if y and z are equal. The second **conde** branch introduces

```
let rec mult^o x y z = conde [
  (x ≡ 0 ∧ z ≡ 0);
  (y ≡ 0 ∧ z ≡ 0);
  (x ≡ S 0 ∧ z ≡ y);
  (y ≡ S 0 ∧ z ≡ x);
  (fresh (x' r')
    (x ≡ S x') ∧
    (add y r' z) ∧
    (mult x' y r')
   ) ]
```

**Listing 7.** Multiplication relation

```
multXY' :: Nat → Nat → Stream Nat
multXY' 0 y = return 0
multXY' x 0 = return 0
multXY' (S 0) y = return y
multXY' x (S 0) = return x
multXY' (S x') y = do
  (r', r) ← addX y
  multXYZ x' y r'
  return r

multXYZ :: Nat → Nat → Nat → Stream ()
multXYZ 0 y 0 = return ()
multXYZ x 0 0 = return ()
multXYZ (S 0) y z | y == z = return ()
multXYZ x (S 0) z | x == z = return ()
multXYZ (S x') y z = do
  z' ← multXY' x' y
  addXYZ y z' z
multXYZ _ _ _ = Empty
```

**Listing 8.** Inefficient implementation of `multo` **in in** `out` direciton

another pattern matching, this time on `z`, which ensures that `z` is not `0`.

Functional implementations of the principal directions of the add$^o$ `x y z` relation which does not make into this section, may be found in Appendix.

### 3.5 Order within Conjunctions

Up until now we only seen examples with only one recursive call which is done to the same relation. Many programs in miniKanren use several relations in the same bodies, see for example Listing 7. The relation mult$^o$ `x y z` relates variables such that `x * y = z`. The base cases in this relation are when `x` or `y` are `0` and `S 0`. When `x` unifies with a successor of another value, then we can use equalities `(x' + 1) * y = x' * y + y`. This is done by adding `y` to the intermediate result of multiplying `x'` by `y`.

```
multXY :: Nat → Nat → Stream Nat
multXY 0 y = return 0
multXY x 0 = return 0
multXY (S 0) y = return y
multXY x (S 0) = return x
multXY (S x') y = do
  r' ← multXY x' y
  addXY y r'
```

**Listing 9.** Efficient implementation of `multo` **in in** `out` direciton

When translating it into a function for the given direction, we need to make sure to call functional counterparts of add$^o$ and mult$^o$ in the right order which depends on the direction. Consider the direction mult$^o$ **in in** `out`. The translation of base cases is done with the same principals as the previous examples. The last **conde** branch contains two call to two different relations: add$^o$ and mult$^o$. Variables `x'` and `y` in this direction are ground, which impose possible directions on the relation calls. There are two ways we can do these calls.

One of them is to first call add$^o$ in the direction add$^o$ **in** `out out` since `y` is ground, while `r` and `r'` are to be computed. After this, all arguments in the call to mult$^o$ are known, and it can be used as a predicate mult$^o$ **in in in**. Finally, we return `r` if the predicate succeeds: see Listing 8. Unfortunately, this order proves to bee too slow: it takes about half of a second to multiply 4 by 4, and more than 300 seconds to multiply 5 by 5. This can be explained by the fact that add$^o$ **in** `out out` generates an infinite streams of answers, only one which succeeds in multiplication, but considering them all even to find the first (and only) answer to `multXY'` takes too much time.

Better and more efficient implementation of mult$^o$ **in in** `out` is shown in Listing 9. Here, we first execute the recursive call of the direction mult$^o$ **in in** `out`, and then use add$^o$ **in in** `out` to compute the final result. None of these relations produce an infinite stream, and the function runs in a fraction of a second. You may note also that in this case there is no need to generate any additional functions for directions which are different from the one being translated.

In general, it is not clear how to choose the best order in which to translate calls within a conjunction. One heuristic is to favor calls which do not produce infinite streams, namely do not use generators for free variables.

## 4 Evaluation

To evaluate our proposed translation scheme, we manually rewritten severals problems in different directions and compared their execution times with their relational counterparts. Here we showcase two relational programs and their translations.

```
topsort graph numbering =
    let n = S (numberOfNodes graph) in
    go graph numbering n
  where
    go graph numbering n =
      case graph of
        []  →  True
        (b, e) : graph'  →
          let nb = lookup numbering b in
          let ne = lookup numbering e in
          less nb ne &&
          less ne n &&
          topsort graph' numbering
```

**Listing 10.** Functional intepreter for topologic sort of a graph

### 4.1 Topologic sort

This program topologically sorts a directed graph. A graph is represented as a list of edges, where each edge is a pair of vertices. First vertex in a pair is the beginning of the edge, and the second vertex is the end of the edge. A vertex is a distinct Peano number in the range [0.. n−1] where n is the number of edges. The vertices are sorted as a result of executing the program. The sort is represented as a list of length n in which the order of vertex i is the i-th element of the list. We call this list *numbering*. For example, numbering [2, 1, 0] means that the zeroth variable is the second, the first variable is the first, and the last variable is the zeroth in the ordering.

The relational program is generated from a functional interpreter cite stuff. The functional interpreter takes a graph and a numbering and checks if the variables are indeed topologically sorted as shown in Listing 10. To do it, it checks all edges of the graph in order, finds the numbers which correspond to the vertices in the numbering, and ensures that the beginning comes before the end of the edge, and that the edge is not greater than the number of vertices in graph.

This simple predicate along with the other functions it uses is translated into a relational program shown in Listing 11. The relation is then specialized so that it searches for a correct topologic sort by fixing its last argument to **true**. The result of specialization is in Listing 12. Specialization removes any **conde** branches which are failing, i.e. unify the result r with **false** .

The specialized version is manually translated in a direction topsort$^o$ **in** out. This creates a function which constructs a numbering which topologically sorts vertices in a given graph. Most of the translation follows the principles outlined in ref section, but there are several notable details about this translation.

First of all, we translated all Peano numbers into Ints and all miniKanren boolean values into Bools. This can be done because of the groundness of variables in this direction. Write something a little more convincing.

Second of all, the relational interpreter contains two consecutive calls to lookup$^o$ relation, both of which has the same numbering passed to them. When translating them, the first call is done in the lookup$^o$ out **in** out direction, since only the value of its second argument b is known to be ground. Calling this direction computes the numbering which is a list with only its b-th element fixed — nb. We generate values of nb with a generator, since nb is a free variable. The same goes for all other elements of the numbering. We restrict the amount of the generating lists by capping their length with maxListLength and capping maximum value of an element with maxInt, both of which correspond to the number of vertices in the input graph.

Having now numbering ground, the second call to lookup$^o$ relation is done in the direction lookup$^o$ **in in** out. The second direction is much simpler as it does not involve generation of any new values for free variables. Translations of the both directions are in Listing 14.

Calls to less$^o$ x y r relations are both done in direction less$^o$ in in out, and their outputs must be **true**. To express this check we use guard which fails computation (i.e. returns an Empty stream) if its argument is false.

### 4.2 Logic Formulas Generation

In this example we translate an evaluator of logic formulas in a direction which generates formulas which evaluate to a given result. Logic formulas are values of type Term presented in Listing 15. A formula is either a boolean literal, a variable indexed by an integer number, a negation of another formula, a conjunction or disjunction of two formulas.

The relational interpreter is shown in Listing 16. The relation eval$^o$ fm st r computes the value r of a formula fm with a given variable mapping st. The boolean value v of a variable Var i is the i-th element of st which can be retrieved by means of the relation elem$^o$ i st v The relation eval$^o$ uses relations add$^o$, or$^o$, and not$^o$ for boolean operations.

Translation of eval$^o$ relation in the direction eval$^o$ out out **in** is presented in Listing 17. As in the previous example, here relation eval$^o$ is called twice when formula is either a conjunction or a disjunction. The direction of the second call is different from the direction of the first call, as first call generates possible variable mappings. The implementation of the direction eval$^o$ out **in in** is shown in Listing 18. The implementations of the directions add$^o$ **in in** out, or$^o$ **in in** out, not$^o$ **in** out, and elem$^o$ **in in** out are in Listing **??**

### 4.3 Execution Time Comparison

In order to assess the usefulness of the proposed transformation scheme we compared execution times of miniKanren

```
let topsortᵒ graph numbering r =
  let rec topsortᵒ graph numbering n r = conde [
    (graph ≡ [] ∧ r ≡ true);
    (fresh (b e graph')
      (graph ≡ (b, e) : graph' ∧
      (fresh (q47 nb ne)
        (lookupᵒ numbering b nb ∧
         lookupᵒ numbering e ne ∧
         lessᵒ nb ne q47 ∧
         conde [
          (q47 ≡ false ∧ r ≡ false);
          (fresh (q43)
            (q47 ≡ true ∧
             lessᵒ ne n q43 ∧
             conde [
              (q43 ≡ false ∧ r ≡ false);
              (q43 ≡ true ∧ topsortᵒ graph' numbering n r)]))])))))] in
   (fresh (n n') (n' ≡ s n ∧ numberOfNodesᵒ graph n ∧ topsortᵒ graph numbering n' r))
```

**Listing 11.** Relational intepreter for topologic sort of a graph

```
let topsortᵒTrue graph numbering =
  let rec topsortᵒ graph numbering n = conde [
    (graph ≡ []);
    (fresh (b e graph')
      (graph ≡ (b, e) : graph' ∧
      (fresh (q47 q43 nb ne)
        (lookupᵒ numbering b nb ∧
         lookupᵒ numbering e ne ∧
         lessᵒ nb ne q47 ∧
         q47 ≡ true ∧
         lessᵒ ne n q43 ∧
         q43 ≡ true ∧
         topsortᵒ graph' numbering n))))] in
   (fresh (n n') (n' ≡ s n ∧ numberOfNodesᵒ graph n ∧ topsortᵒTrue graph numbering n'))
```

**Listing 12.** Specialized relational intepreter for topologic sort of a graph

relations `topsort`$^o$ and `eval`$^o$ with their functional translations. All functional translations are done by hand, having a specific direction in mind. All implementations are written in OCaml language and can be found in this repository. Note that throughout this paper we presented all examples written in Haskell for brevity, but we used OCaml in evaluation to make the comparison with OCanren more fair. Technically, to implement our translations in OCaml, we had to desugar Haskell do-notation into binds and make some calls return lazy streams.

For the evaluator of logic formulas, we run both implementations to search for 10000 formulas which evaluate to `True`. The functional implementation restricts the length of the variable mapping list, thus we also restricted the size of it in its relational counterpart. We averaged the execution time over 10 runs. The result are presented in table 1. "OCanren" contains execution time of relational implementation, and "Function" column contains execution time of the functional implementation. In our experiments, functional implementation outperforms the relational interpretation by 1.3-2.5 times.

We run `topsort`$^o$ on directed graphs with exactly one edge between each pair of edges. For example, graph with 4 vertices has the following edges: [(0, 1), (0, 2), (0, 3), (1, 2), which we sort lexicographically. We generated graphs for a given number of vertices and then executed both relational and functional implementations of `topsort`$^o$. The correct numbering in this condition should map each vertex into itself. We also run the same functions on the same graph, but with its list of edges reversed, i.e. [(2, 3), (1, 3), (1, 2), (0, 3),

```
topsortGraph :: Graph → Stream [Nat]
topsortGraph graph = do
    n ← numberOfNodesG graph
    go graph (n + 1) n (n + 1)
  where
    go graph n maxInt maxListLength =
      case graph of
        [] → return []
        ((b, e) : graph') → do
          (nb, numbering) ← lookupKey b maxInt maxListLength
          ne ← lookupXsKey numbering e
          q47 ← lessXY nb ne
          guard q47
          q43 ← lessXY ne n
          guard q43
          topsortGraphNumbering graph' numbering n
```

**Listing 13.** Functional implementation for a `topsortoTrue` **in** `out` direction

```
lookupKey :: Int → Int → Int → Stream (Int, [Int])
lookupKey key maxInt maxListLength =
  case key of
    0 → fromList [(x, x:xs) | xs ← genList (genInt maxInt) (maxListLength - 1),
                              x ← genInt maxInt]
    _ | key > 0 → do
      (value, tl) ← lookupKey (key - 1) maxInt (maxListLength - 1)
      fromList [(value, y : tl) | y ← genInt maxInt]
    _ → Empty
lookupXsKey :: [Int] → Int → Stream Int
lookupXsKey xs key =
  case xs of
    [] → Empty
    (h : tl) → case key of
                 0 → return h
                 S key' → lookupXsKey tl key'
```

**Listing 14.** Functional implementations for a `lookupo out` **in** `out` and `lookupo` **in in** `out` directions

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

**Listing 15.** Term data type

In this case, the correct numbering maps a vertex `i` into `n - i`, where `n` is the number of vertices in the graph.

Execution times averaged over 10 runs are presented in table 2. Columns "Functional" and "Functional (r)" contain execution times of functional implementations when run on a graph and reversed graph correspondingly. Columns "OCanren" and "OCanren (r)" contain execution times of functional implementations when run on a graph and reversed graph correspondingly. Relational implementation took more than 300 seconds for a sorted graph with 7 vertices, thus we only consider graphs with up to 6 vertices. On all graphs, functional implementation much less time that the miniKanren program. Topologically sorting a reversed

```
evalᵒ st fm u =
  fresh (x y v w z) (conde [
    (fm ≡ Conj x y ∧ andᵒ v w u ∧ evalᵒ st x v ∧ evalᵒ st y w);
    (fm ≡ Disj x y ∧ orᵒ v w u ∧ evalᵒ st x v ∧ evalᵒ st y w);
    (fm ≡ Neg x ∧ notᵒ v u ∧ evalᵒ st x v);
    (fm ≡ Var z ∧ elemᵒ z st u);
    (fm ≡ Lit u)])

andᵒ x y b = conde [
  (x ≡ True ∧ y ≡ True ∧ b ≡ True);
  (x ≡ False ∧ y ≡ True ∧ b ≡ False);
  (x ≡ True ∧ y ≡ False ∧ b ≡ False);
  (x ≡ False ∧ y ≡ False ∧ b ≡ False)]

orᵒ x y b = conde [
  (x ≡ True ∧ y ≡ True ∧ b ≡ True);
  (x ≡ False ∧ y ≡ True ∧ b ≡ True);
  (x ≡ True ∧ y ≡ False ∧ b ≡ True);
  (x ≡ False ∧ y ≡ False ∧ b ≡ False)]

notᵒ x b = [
  (x ≡ True ∧ b ≡ False);
  (x ≡ False ∧ b ≡ True)]

elemᵒ i st v =
  fresh (h t i') conde [
    (i ≡ O ∧ st ≡ (v : t));
    (i ≡ S i' ∧ st ≡ (h : t) ∧ elemᵒ i' t v)]
```

**Listing 16.** Relational evaluator of logic formulas

**Table 1.** Execution times of the OCanren and functional implementations of evalo, search for 10000 formulas which evalute to True

| Var. mapping length | Function (sec.) | OCanren (sec.) |
|---|---|---|
| 0 | 0.283 | 0.998 |
| 1 | 0.306 | 0.668 |
| 2 | 0.227 | 0.543 |
| 3 | 0.224 | 0.500 |
| 4 | 0.206 | 0.482 |
| 5 | 0.211 | 0.482 |
| 6 | 0.254 | 0.483 |
| 7 | 0.370 | 0.491 |
| 8 | 0.357 | 0.492 |
| 9 | 0.377 | 0.491 |

graph takes much less time. This is caused by earlier rejection of candidate solutions, since vertex numbers are higher in the beginning of the list.

As a result of our evaluation, we can conclude that the translation of miniKanren program with a given direction into a function speeds up execution a lot and thus it is reasonable to continue working in this direction.

Topological Sort

## 5 Related Work

- Relational interpeters for context ([1])
- Mercury for mode analysis
- Curry as translation to Haskell

Automatic translation from a general purpose programming cite Lozov, unnesting makes it possible to create relational specifications which then may be run in a direction of choice and thus do more than original program. As an example, one may implement a simple functional verifier which checks that some candidate is indeed a solution for a search problem. When translated into miniKanren, this verifier may be used to actually solve search problems with no deep knowledge required from a programmer. cite rel.interpreters.

## 6 Future Work

- Research if there exists a way to automatically deduce better order of calls within a conjunction.

```
evalR :: Bool → Int → Stream (Term, [Bool])
evalR result maxListLength =
    lit  result `mplus`
    var  result `mplus`
    neg  result `mplus`
    disj result `mplus`
    conj result
  where
    conj result = do
      (v, w) ← andR result
      (y, st) ← evalR w maxListLength
      x ← evalStR st v
      return (Conj x y, st)
    disj result = do
      (v, w) ← orR result
      (y, st) ← evalR w maxListLength
      x ← evalStR st v
      return (Disj x y, st)
    neg result = do
      v ← notR result
      (x, st) ← evalR v  maxListLength
      return (Neg x, st)
    var result = do
      (z, st) ← elemR result maxListLength
      return (Var z, st)
    lit result =
      if result
      then return (Lit True, [])
      else return (Lit False, [])
```

**Listing 17.** Functional implementation of the direction `evalo out out` **in**

**Table 2.** Execution times of the OCanren and functional implementations of `topsorto`

| Number of vertices | Function (sec.) | OCanren (sec.) | Function (r) (sec.) | OCanren (r) (sec.) |
|---|---|---|---|---|
| 3 | 0.000 | 0.001 | 0.000 | 0.001 |
| 4 | 0.000 | 0.015 | 0.000 | 0.012 |
| 5 | 0.001 | 0.346 | 0.000 | 0.107 |
| 6 | 0.021 | 14.309 | 0.003 | 0.764 |

- Formalize translation scheme, prove its correctness
- Implement automatic translation
- Integrate the translator into a relational interpreters framework

## 7 Conclusion

## Acknowledgments

Here is where acknowledgments come

## References

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

## A Principal Directions of the Addition Relation

```
evalStR :: [Bool] → Bool → Stream Term
evalStR st result =
      lit  st result `mplus`
      var  st result `mplus`
      neg  st result `mplus`
      disj st result `mplus`
      conj st result
  where
    conj st result = do
      (v, w) ← andR result
      y ← evalStR st w
      x ← evalStR st v
      return (Conj x y)
    disj st result = do
      (v, w) ← orR result
      y ← evalStR st w
      x ← evalStR st v
      return (Disj x y)
    neg st result = do
      v ← notR result
      x ← evalStR st v
      return (Neg x)
    var st result = do
      z ← elemStR st result
      return (Var z)
    lit st result =
      if result
      then return (Lit True)
      else return (Lit False)
```

**Listing 18.** Functional implementation of the direction evalo out **in** **in**

```
andR :: Bool → Stream (Bool, Bool)
andR result =
  if result
  then
    return (True, True)
  else
    return (True, False) `mplus`
    return (False, True) `mplus`
    return (False, False)

orR :: Bool → Stream (Bool, Bool)
orR result =
  if result
  then
    return (True, False) `mplus`
    return (True, True) `mplus`
    return (False, True)
  else
    return (False, False)

notR :: Bool → Stream Bool
notR result =
  if result
  then return False
  else return True

elemR :: Bool → Int → Stream (Int, [Bool])
elemR _ maxListLength | maxListLength <= 0 = Empty
elemR result maxListLength =
     zero result `mplus` succ result
  where
    zero result  =
      fromList [ (0, result : tl) | tl ← genList genBool (maxListLength - 1) ]
    succ result = do
      (n', t) ← elemR result (maxListLength - 1)
      fromList [(n' + 1, h : t) | h ← genBool ]
```

**Listing 19.** Functions used in logic formulas generation