

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 style. 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 email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXX.XXXXXXX>

1 Introduction

Here is going to be an introduction.

2 Examples of Translation

A relation $\text{add}^0 x y z$ succeeds whenever z is a sum of x and y . An implementation of this relation which uses Peano numbers is shown in Listing 1.

One can run a relation in some *direction* by passing it *input* arguments. For example, executing $\text{add}^0 (S\ 0)\ 0\ z$ finds the sum of the first two arguments and maps z to the sum $S\ 0$. We can also provide only the last argument: $\text{add}^0 x y (S\ 0)$. This computes all pairs of Peano numbers (x, y) which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/XXXXXX.XXXXXXX>

```
let rec add0 x y z = conde [
  (x ≡ 0 ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧
     z ≡ S z' ∧
     add0 x' y z')) ]
```

Listing 1. Addition relation

sum up to the given value $z = S\ 0$ which are $(0, S\ 0)$ and $(S\ 0, 0)$. Moreover, we can pass as input arguments not only *ground terms* but terms which contain fresh variables, such as $\text{add}^0 x (S\ y)\ z$. Executing this relation finds all triples (x, y, z) such that $x + (y + 1) = z$. Running in some directions can fail. For example $\text{add}^0 (S\ x)\ y\ 0$ may never succeed, since $(1 + x) + y$ can never be equal to **zero**.

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 $\text{add}^0 x y z$:

- three directions with one input: add^0 **in** out out, add^0 out **in** out, and add^0 out out **in**;
- three directions with two inputs: add^0 **in in** out, add^0 **in** out **in**, add^0 out **in in**;
- one direction with no input arguments: add^0 out out out;
- one direction when all arguments are input: add^0 **in in in**;

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

In the rest of this section we describe a translation scheme which allows to implement principal directions of a relation as functions. Each direction we consider illustrates some aspect of the translation. Functional implementations of the principal directions of the $\text{add}^0 x y z$ relation which does not make into this section, may be found in Appendix.

2.1 Basic Translation

Consider add^0 **in in** out. This direction can be expressed as a function presented in Listing 2. The relation $\text{add}^0 x y z$ has two branches in a **conde**: one unifies x with **zero** and the

```

111 addXY :: Nat → Nat → Nat
112 addXY x y =
113   case x of
114     0 → y
115     S x' → S (addXY x' y)

```

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

```

119 addXY :: Nat → Nat → Stream Nat
120 addXY x y =
121   case x of
122     0 → return y
123     S x' → S <$> addXY x' y

```

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

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 **zero**, the rest of the **conde** branch is the unification $y \equiv z$. 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 **zero**.

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 $\text{add}^0\ x'\ y\ z'$ is done in the direction $\text{add}^0\ \text{in in}\ \text{out}$ 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 $z \equiv S\ z'$.

2.2 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 $\text{add}^0\ \text{out out}\ \text{in}$. 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.

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 **zero**, and y is equal to z . This single result is then wrapped into a singleton stream.

```

166 addZ :: Nat → Stream (Nat, Nat)
167 addZ z =
168   return (0, z) `mplus`
169   case z of
170     0 → Empty
171     S z' → do
172       (x', y) ← addZ z'
173       return (S x', y)

```

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

```

177 addX :: Nat → Stream (Nat, Nat)
178 addX x =
179   case x of
180     0 → do
181       z ← genNat
182       return (z, z)
183     S x' → do
184       (y, z') ← addX x'
185       return (y, S z')

```

```

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

```

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

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

2.3 Free Variables in Answers

In some directions, there are infinitely many answers, such as in $\text{add}^0\ \text{in out}\ \text{out}$. 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 $(0, S\ 0), (S\ 0, S\ (S\ 0)), \dots$. To do it, we need to systematically generate a stream of

```

221 addXYZ :: Nat → Nat → Nat → Stream ()
222 addXYZ x y z =
223   case x of
224     0 | y == z → return ()
225     | otherwise → Empty
226   S x' →
227     case z of
228       0 → Empty
229       S z' → addXYZ x' y z'

```

Listing 6. Function for addo in in in direction

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 `addo in out out`. This direction is very similar to the `addo 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 **zero**, 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.

2.4 Predicates

When all arguments of a relation are input, the direction serves as a predicate. Consider `addo 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 another pattern matching, this time on z, which ensures that z is not **zero**.

2.5 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 `multo x y z` relates variables such that $x * y = z$. The base cases in this relation are when x or y are **zero** and S 0. When x unifies with a successor of another value, then we can use equalities

```

276 let rec multo x y z = conde [
277   (x ≡ 0 ∧ z ≡ 0);
278   (y ≡ 0 ∧ z ≡ 0);
279   (x ≡ S 0 ∧ z ≡ y);
280   (y ≡ S 0 ∧ z ≡ x);
281   (fresh (x' r')
282     (x ≡ S x') ∧
283     (add y r' z) ∧
284     (mult x' y r')
285   ) ]

```

Listing 7. Multiplication relation

```

289 multXY' :: Nat → Nat → Stream Nat
290 multXY' 0 y = return 0
291 multXY' x 0 = return 0
292 multXY' (S 0) y = return y
293 multXY' x (S 0) = return x
294 multXY' (S x') y = do
295   (r', r) ← addX y
296   multXYZ x' y r'
297   return r

```

```

299 multXYZ :: Nat → Nat → Nat → Stream ()
300 multXYZ 0 y 0 = return ()
301 multXYZ x 0 0 = return ()
302 multXYZ (S 0) y z | y == z = return ()
303 multXYZ x (S 0) z | x == z = return ()
304 multXYZ (S x') y z = do
305   z' ← multXY' x' y
306   addXYZ y z' z
307   multXYZ _ _ _ = Empty

```

Listing 8. Inefficient implementation of multo in in out direciton

```

312 multXY :: Nat → Nat → Stream Nat
313 multXY 0 y = return 0
314 multXY x 0 = return 0
315 multXY (S 0) y = return y
316 multXY x (S 0) = return x
317 multXY (S x') y = do
318   r' ← multXY x' y
319   addXY y r'

```

Listing 9. Efficient implementation of multo in in out direciton

$(x' + 1) * y = x' * y + y$. This is done by adding y to the intermediate result of multiplying x' by y.

When translating it into a function for the given direction, we need to make sure to call functional counterparts of `addo` and `multo` 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** 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 be 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** 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 mult^o 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.

3 Evaluation

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

3.1 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.

```

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 interpreter for topologic sort of a graph

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 [red 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 Booleans. 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


```

441 let topsorto graph numbering r =
442   let rec topsorto graph numbering n r = conde [
443     (graph ≡ [] ∧ r ≡ true);
444     (fresh (b e graph')
445       (graph ≡ (b, e) : graph' ∧
446         (fresh (q47 nb ne)
447           (lookupo numbering b nb ∧
448             lookupo numbering e ne ∧
449             lesso nb ne q47 ∧
450             conde [
451               (q47 ≡ !! false ∧ r ≡ false);
452               (fresh (q43)
453                 (q47 ≡ true ∧
454                   lesso ne n q43 ∧
455                   conde [
456                     (q43 ≡ false ∧ r ≡ false);
457                     (q43 ≡ true ∧ topsorto graph' numbering n r)])))])))] in
458   (fresh (n n') (n' ≡ s n ∧ numberOfNodeso graph n ∧ topsorto graph numbering n' r))

```

Listing 11. Relational interpreter for topologic sort of a graph

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 `lookupo` relation is done in the direction `lookupo 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 `lesso x y r` relations are both done in direction `lesso 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.

3.2 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 `evalo 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 `elemo i st v`. The relation `evalo` uses relations `addo`, `oro`, and `noto` for boolean operations.

Translation of `evalo` relation in the direction `evalo out out in` is presented in Listing 17. As in the previous example, here relation `evalo` 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 `evalo out in in` is shown in Listing 18. The implementations of the directions `addo in in out`, `oro in in out`, `noto in out`, and `elemo in in out` are in Listing ??

3.3 Execution Time Comparison

In order to assess the usefulness of the proposed transformation scheme we compared execution times of `MINIKANREN` relations `topsorto` and `evalo` 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](#).

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 `topsorto` 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 `topsorto`. 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)],

```

551 let topsortoTrue graph numbering =
552   let rec topsorto graph numbering n = conde [
553     (graph ≡ []);
554     (fresh (b e graph')
555       (graph ≡ (b, e) : graph' ∧
556         (fresh (q47 q43 nb ne)
557           (lookupo numbering b nb ∧
558             lookupo numbering e ne ∧
559             lesso nb ne q47 ∧
560             q47 ≡ true ∧
561             lesso ne n q43 ∧
562             q43 ≡ true ∧
563             topsorto graph' numbering n)))))] in
564   (fresh (n n') (n' ≡ s n ∧ numberOfNodeso graph n ∧ topsortoTrue graph numbering n'))

```

Listing 12. Specialized relational interpreter for topologic sort of a graph

```

568 topsortGraph :: Graph → Stream [Nat]
569 topsortGraph graph = do
570   n ← numberOfNodesG graph
571   go graph (n + 1) n (n + 1)
572 where
573   go graph n maxInt maxListLength =
574     case graph of
575       [] → return []
576       ((b, e) : graph') → do
577         (nb, numbering) ← lookupKey b maxInt maxListLength
578         ne ← lookupXsKey numbering e
579         q47 ← lessXY nb ne
580         guard q47
581         q43 ← lessXY ne n
582         guard q43
583         topsortGraphNumbering graph' numbering n

```

Listing 13. Functional implementation for a topsorttoTrue **in** out direction

Table 1. Execution times of the OCanren and functional implementations of eval_o, search for 10000 formulas which evaluate 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

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 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.

```

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

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

4 Related Work

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

5 Future Work

- Research if there exists a way to automatically deduce better order of calls within a conjunction.
- Formalize translation scheme, prove its correctness
- Implement automatic translation
- Integrate the translator into a relational interpreters framework

6 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

```

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 ≡ 0 ∧ st ≡ (v : t));
    (i ≡ S i' ∧ st ≡ (h : t) ∧ elemo i' t v)]

```

Listing 16. Relational evaluator of logic formulas**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

```

881 evalR :: Bool → Int → Stream (Term, [Bool])
882 evalR result maxLength =
883   lit result `mplus`
884   var result `mplus`
885   neg result `mplus`
886   disj result `mplus`
887   conj result
888   where
889     conj result = do
890       (v, w) ← andR result
891       (y, st) ← evalR w maxLength
892       x ← evalStR st v
893       return (Conj x y, st)
894     disj result = do
895       (v, w) ← orR result
896       (y, st) ← evalR w maxLength
897       x ← evalStR st v
898       return (Disj x y, st)
899     neg result = do
900       v ← notR result
901       (x, st) ← evalR v maxLength
902       return (Neg x, st)
903     var result = do
904       (z, st) ← elemR result maxLength
905       return (Var z, st)
906     lit result =
907       if result
908       then return (Lit True, [])
909       else return (Lit False, [])

```

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

```

991 evalStR :: [Bool] → Bool → Stream Term
992 evalStR st result =
993     lit st result `mplus`
994     var st result `mplus`
995     neg st result `mplus`
996     disj st result `mplus`
997     conj st result
998 where
999     conj st result = do
1000         (v, w) ← andR result
1001         y ← evalStR st w
1002         x ← evalStR st v
1003         return (Conj x y)
1004     disj st result = do
1005         (v, w) ← orR result
1006         y ← evalStR st w
1007         x ← evalStR st v
1008         return (Disj x y)
1009     neg st result = do
1010         v ← notR result
1011         x ← evalStR st v
1012         return (Neg x)
1013     var st result = do
1014         z ← elemStR st result
1015         return (Var z)
1016     lit st result =
1017         if result
1018         then return (Lit True)
1019         else return (Lit False)

```

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

```

1101 andR :: Bool → Stream (Bool, Bool)
1102 andR result =
1103   if result
1104   then
1105     return (True, True)
1106   else
1107     return (True, False) `mplus`
1108     return (False, True) `mplus`
1109     return (False, False)
1110
1111 orR :: Bool → Stream (Bool, Bool)
1112 orR result =
1113   if result
1114   then
1115     return (True, False) `mplus`
1116     return (True, True) `mplus`
1117     return (False, True)
1118   else
1119     return (False, False)
1120
1121 notR :: Bool → Stream Bool
1122 notR result =
1123   if result
1124   then return False
1125   else return True
1126
1127 elemR :: Bool → Int → Stream (Int, [Bool])
1128 elemR _ maxLength | maxLength <= 0 = Empty
1129 elemR result maxLength =
1130   zero result `mplus` succ result
1131   where
1132     zero result =
1133       fromList [ (0, result : tl) | tl ← genList genBool (maxLength - 1) ]
1134     succ result = do
1135       (n', t) ← elemR result (maxLength - 1)
1136       fromList [(n' + 1, h : t) | h ← genBool ]

```

Listing 19. Functions used in logic formulas generation

1211	1266
1212	1267
1213	1268
1214	1269
1215	1270
1216	1271
1217	1272
1218	1273
1219	1274
1220	1275
1221	1276
1222	1277
1223	1278
1224	1279
1225	1280
1226	1281
1227	1282
1228	1283
1229	1284
1230	1285
1231	1286
1232	1287
1233	1288
1234	1289
1235	1290
1236	1291
1237	1292
1238	1293
1239	1294
1240	1295
1241	1296
1242	1297
1243	1298
1244	1299
1245	1300
1246	1301
1247	1302
1248	1303
1249	1304
1250	1305
1251	1306
1252	1307
1253	1308
1254	1309
1255	1310
1256	1311
1257	1312
1258	1313
1259	1314
1260	1315
1261	1316
1262	1317
1263	1318
1264	1319
1265	1320

1321	1376
1322	1377
1323	1378
1324	1379
1325	1380
1326	1381
1327	1382
1328	1383
1329	1384
1330	1385
1331	1386
1332	1387
1333	1388
1334	1389
1335	1390
1336	1391
1337	1392
1338	1393
1339	1394
1340	1395
1341	1396
1342	1397
1343	1398
1344	1399
1345	1400
1346	1401
1347	1402
1348	1403
1349	1404
1350	1405
1351	1406
1352	1407
1353	1408
1354	1409
1355	1410
1356	1411
1357	1412
1358	1413
1359	1414
1360	1415
1361	1416
1362	1417
1363	1418
1364	1419
1365	1420
1366	1421
1367	1422
1368	1423
1369	1424
1370	1425
1371	1426
1372	1427
1373	1428
1374	1429
1375	1430