# On a Direction-Driven Functional Conversion

Ekaterina Verbitskaia
kajigor@gmail.com
JetBrains Research
Belgrade, Serbia

Daniil Berezun
daniil.berezun@jetbrains.com
JetBrains Research
Amsterdam, Netherlands

Dmitry Boulytchev
dboulytchev@math.spbu.ru
SPbSU
Saint Petersburg, Russia

## Abstract

Relational programming is known for its capability to provide a short and concise executable specifications for a wide range of interesting problems. Specifically, the nature of relational programming makes it possible to consider a single specification as a whole family of concrete programs. Individual programs of this family can be taken and run by placing free variables inside a top-level goal arguments. In particular, relational programming provides a very generic way to implement *program inversion*, which opens a way for program synthesis via converting *verifiers* into *solvers*. However, acquired in such a way solvers often come with an overhead, originating from the very nature of relational computations with substitutions, unifications, interleaving, etc. In this paper we study a conversion of relational programs into functional form taking into account a concrete *direction* of evaluation. The project is at an early stage, but the results so far are promising: converted functions run much faster than the original relations.

***Keywords:*** relational programming, functional programming, conversion

**ACM Reference Format:**
Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2018. On a Direction-Driven Functional Conversion. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

It is well-known that some programs are easier to implement as inversions of other, simpler programs [2]. One of the notable cases is *verifiers* vs. *solvers* [7]: it is rather easy to implement a verification procedure which tests if a given candidate is indeed a solution of a certain problem, and the inversion of this procedure delivers a solver. There are a few aproaches to program inversion, for example, universal resolving algorithm [3] and logic and relational programming. In latter case, inversion comes with a lot of overhead which may be eliminated.

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 [11] 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 study a conversion of miniKanren programs into a host functional programming language in a sequence of examples. Examples start from the simplest conversions and evolve to introduce different features of miniKanren which influence conversion. Currently the conversion is not automated: everything is done manually. We believe the conversion can be semi-automated, leaving some decisions up to a programmer. Although this project is at the early state, the 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 a 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 [6].

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



*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

goals which is just a syntactic sugar. Unifications between two terms are denoted by a not associative binary relation ≡. 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.

Consider an addition relation `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 `add`$^o$ `(S 0) 0 z` finds the sum of the first two arguments and maps `z` to their sum `S 0`. We can also provide only the last argument: `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 `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 `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 a 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 `add`$^o$ `x y z`:

- three directions with one input: `add`$^o$ `in out out`, `add`$^o$ `out in out`, and `add`$^o$ `out out in`;
- three directions with two inputs: `add`$^o$ `in in out`, `add`$^o$ `in out in`, `add`$^o$ `out in in`;
- one direction which does not have any input arguments: `add`$^o$ `out out out`;
- and one direction in which all arguments are input: `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.

## 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. For brevity, we will use Haskell as a target language in this paper. In practice, any programming language in which miniKanren is implemented may be used as a target language.

```
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

```
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

### 3.1 Basic Conversion

Consider `add`$^o$ `in in out`. This direction can be expressed as a function presented in Listing 2. The relation `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 `add`$^o$ `x' y z'` is done in the direction `add`$^o$ `in in out` and can be converted 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'`.

```
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

### 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.

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 `0` it 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 makes `z'` ground, and the recursive call to the relation is done in the direction $\text{add}^o$ `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 converted **conde** branches are then combined by using `` `mplus` ``: the same combinator which is used in miniKanren for disjunctions. We use do-notation when converting 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.

### 3.3 Free Variables in Answers

In some directions, there are infinitely many answers, such as in $\text{add}^o$ `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 the expected stream of answers is `(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.

```
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

```
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

Listing 5 shows the functional implementation of the direction $\text{add}^o$ `in out out`. This direction is very similar to the $\text{add}^o$ `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`, where `Mature` is a stream constructor. 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 $\text{add}^o$ `in in in` and its functional implementation in Listing 6. In this case there is no

```
let rec multᵒ 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

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 converting 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 `0`.

Functional implementations of other principal directions of the $\texttt{add}^o$ `x y z` relation which did not make into this section, can be found in Appendix A.

### 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 $\texttt{mult}^o$ `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`.

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

One of them is to first call $\texttt{add}^o$ in the direction $\texttt{add}^o$ `in out out` since `y` is ground, while `r` and `r'` are to be computed. After this, all arguments in the call to $\texttt{mult}^o$ are known, and it can be used as a predicate $\texttt{mult}^o$ `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 $\texttt{add}^o$ `in out out` generates an infinite streams of answers, only one which succeeds in multiplication predicate, but considering them all even to find the first (and only) answer to `multXY'` takes too much time.

Better and more efficient implementation of $\texttt{mult}^o$ `in in out` is shown in Listing 9. Here, we first execute the recursive call of the direction $\texttt{mult}^o$ `in in out`, and then use $\texttt{add}^o$ `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. Note also that in this case there is no need to generate any additional functions for directions which are different from the one being converted.

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

```
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

```
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

```
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 (x nb ne)
        (lookupᵒ numbering b nb ∧
         lookupᵒ numbering e ne ∧
         lessᵒ nb ne x ∧
         conde [
          (x ≡ false ∧ r ≡ false);
          (fresh (y)
            (x ≡ true ∧
             lessᵒ ne n y ∧
             conde [
              (y ≡ false ∧ r ≡ false);
              (y ≡ 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

## 4 Evaluation

To evaluate our proposed conversion 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 conversions.

### 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. The first vertex of a pair is the beginning of the

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

edge, and the second vertex is the end of the edge. A vertex is a distinct Peano number in the range $[0..\ \mathtt{n}{-}1]$ 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 vertex is the second, the first vertex is the first, and the last vertex is the zeroth in the ordering.

The relational program is generated from a functional verifier as proposed in [7]. The functional interpreter takes a graph and a numbering and checks if the vertices 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 end of the edge is not greater than the number of vertices in graph.

This simple predicate along with the other functions it uses is converted 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 converted in a direction `topsortᵒ in out`. This creates a function which constructs a numbering which topologically sorts vertices in a given graph. Most of the conversion follows the principles outlined in the previous section, but there are several notable details about this conversion.

```
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
          x ← lessXY nb ne
          guard x
          y ← lessXY ne n
          guard y
          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
                 O → return h
                 S key' → lookupXsKey tl key'
```

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

First of all, we replaced all Peano numbers with `Int`s and all miniKanren boolean values with `Bool`s. This can be done because of the groundness of variables in this direction.

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 converting 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 number of the generated 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. Conversions 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.

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

**Listing 15.** Term data type

### 4.2 Logic Formulas Generation

In this example we convert 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 `and`$^o$, `or`$^o$, and `not`$^o$ for boolean operations.

Conversion of `eval`$^o$ relation in the direction `eval`$^o$ `out out in` is presented in Listing 17. As in the previous example, the 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 `and`$^o$ `in in out`, `or`$^o$ `in in out`, `not`$^o$ `in out`, and `elem`$^o$ `in in out` are in Listing 19.

```
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

### 4.3 Execution Time Comparison

In order to assess the usefulness of the proposed transformation scheme we compared execution times of miniKanren relations `topsort`$^o$ and `eval`$^o$ with their functional conversions. All functional conversions are done by hand, having a specific direction in mind. All implementations are written in OCaml language and can be found in the 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 conversions 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 and figure 2. "OCanren" column 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

```
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 b = return (Lit b, [])
```

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

| 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 |

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

| 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 |

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

```
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 b = Lit b
```

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

```
andR :: Bool → Stream (Bool, Bool)
andR True = return (True, True)
andR False = return (True, False) `mplus`
             return (False, True) `mplus`
             return (False, False)

orR :: Bool → Stream (Bool, Bool)
orR True = return (True, True) `mplus`
           return (True, False) `mplus`
           return (False, True)
orR False = return (False, False)

notR :: Bool → Stream Bool
notR True = return False
notR False = 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

4 vertices has the following edges: `[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]`, 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), (0, 2), (0, 1)]`. 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 and figure 1. Columns "Functional" and "Functional (r)" contain execution times of functional implementations

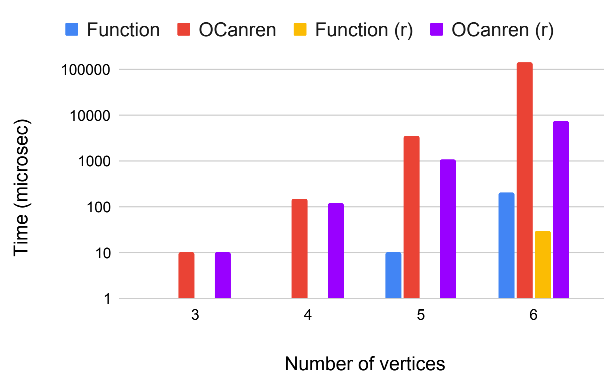


**Figure 1.** Comparison of exection time of topologic sort (logarithmic scale, time measured in microseconds)

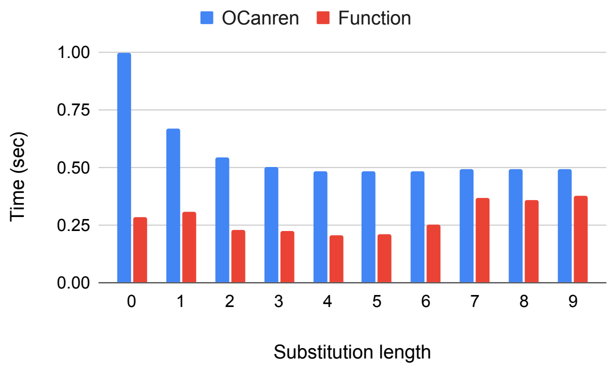


**Figure 2.** Comparison of exection time of formulas generator (time measured in seconds)

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 is faster than the miniKanren program. Topologically sorting a reversed graph takes significantly 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 conversion 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.

## 5 Related Work

There are several research area relevant to our conversion. Semantic modifiers [1] and universal resolving algorithm [3] may be used for inverse interpretation of first order functional programs. They do not guarantee termination in general, which is reasonable, given that the problem is undecidable.

Logic and relational programming languages inherently support inverse computations, but they often come with significant overhead. Reducing such overhead may be done with such techniques as partial evaluation, or partial deduction. Applying these techniques to miniKanren has not yet done successfully, although some speed ups were achieved [11].

Functional logic programming languages such as Curry and Mercury translate their logic subsets into a general programming language. Mercury uses a sophisticated system of modes along with mode analysis [9] which we plan to adapt to miniKanren as part of future work. The search strategy in Mercury is not complete which limits its use for our application.

Curry has several compilers including the one whose target language is Haskell [4]. Although, Curry provides some flexibility in choosing the search strategy [5], it uses `choice` to implement nondeterminism instead of unifications.

An earlier attempt at conversion of miniKanren into a functional program has been made. It involved binding-time analysis to determine `in`/`out` annotations of variables [10]. It only works for directions which return finite answer streams which severely limits its applications.

There exist an automatic conversion from a subset of OCaml into OCanren [8]. Coupling it with conversion from miniKanren back into OCaml can be used to efficiently inverse computations.

## 6 Future Work

Since this project is in active phase of development, there are many directions for future work.

First of all, we need to research how to best order calls within a conjunction. Since the order of calls greatly influences the efficiency of the converted function, this research direction is of upmost importance. Annotations of variables with `in` and `out` are also affected by the order of calls and thus we need to adapt the mode analysis to take it into account.

Second of all, we plan to formalize the conversion and prove its correction.

Third of all, the conversion should be implemented either as a standalone tool or integrated into some of the major miniKanren implementations.

Finally, after all these building blocks are done, we would like to integrate the conversion into a relational interpreters framework. This would made a fullstack solution for the program inversion problem.

## 7 Conclusion

In this paper we described a new conversion from a miniKanren relation with a fixed execution direction into a functional programming language. We manually converted several miniKanren relations and compared execution time of the converted functions with their relational sources. The evaluation showed that the conversion is able to speed up computations significantly. We also mentioned some complicated steps within conversion and outlined directions for future research.

## References

[1] Sergei Abramov and Robert Glück. 2001. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science* 12, 02 (2001), 171–211.

[2] Sergei Abramov and Robert Glück. 2002. *Principles of Inverse Computation and the Universal Resolving Algorithm*. Springer Berlin Heidelberg, Berlin, Heidelberg, 269–295. https://doi.org/10.1007/3-540-36377-7_13

[3] Sergei Abramov and Robert Glück. 2002. Principles of inverse computation and the universal resolving algorithm. In *The essence of computation*. Springer, 269–295.

[4] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. 2011. KiCS2: A new compiler from Curry to Haskell. In *International Workshop on Functional and Constraint Logic Programming*. Springer, 1–18.

[5] Michael Hanus, Björn Peemöller, and Fabian Reck. 2012. Search strategies for functional logic programming. *Software Engineering 2012. Workshopband* (2012).

[6] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices* 40, 9 (2005), 192–203.

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

[8] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed relational conversion. In *International Symposium on Trends in Functional Programming*. Springer, 39–58.

[9] David Overton, Zoltan Somogyi, and Peter J Stuckey. 2002. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 109–120.

[10] Ekaterina Verbitskaia, Irina Artemeva, and Daniil Berezun. 2020. Binding-Time Analysis for miniKanren. https://www.coalg.org/tease-lp/files/2020/05/verbitskaia.pdf. Workshop on Trends, Extensions, Applications and Semantics of Logic Programming.

[11] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2021. An Empirical Study of Partial Deduction for miniKanren. In Proceedings of the 9th International Workshop on *Verification and Program Transformation,* Luxembourg, Luxembourg, 27th and 28th of March 2021 *(Electronic Proceedings in Theoretical Computer Science)*, Alexei Lisitsa and Andrei P. Nemytykh (Eds.), Vol. 341. Open Publishing Association, 73–94. https://doi.org/10.4204/EPTCS.341.5


# A Principal Directions of the Addition Relation

```
add :: Stream (Nat, Nat, Nat)
add =
    disj1 `mplus` disj2
  where
    disj1 = do
      z ← genNat
      return (O, z, z)
    disj2 = do
      (x', y, z') ← add
      return (S x', y, S z')
```

**Listing 20.** Function for addo `out out out` direction

```
addY :: Nat → Stream (Nat, Nat)
addY y =
  return (O, y) `mplus`
  do
    (x', z') ← addY y
    return (S x', S z')
```

**Listing 21.** Function for addo `out in out` direction

```
addXZ :: Nat → Nat → Stream Nat
addXZ x z =
  case x of
    O → return z
    S x' →
      case z of
        O → Empty
        S z' →
          addXZ x' z'
```

**Listing 22.** Function for addo `in out in` direction

```
addYZ :: Nat → Nat → Stream Nat
addYZ y z =
  if y == z
  then return O
  else
    case z of
      S z' → do
        x ← addYZ y z'
        return (S x)
      O → Empty
```

**Listing 23.** Function for addo `out in in` direction