# 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, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Here is going to be an introduction.

## 2 Examples of Translation

A relation add$^o$ `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 add$^o$ `(S O) O z` finds the sum of the first two arguments and maps `z` to the sum `S O`. We can also provide only the last argument: add$^o$ `x y (S O)`. This computes all pairs of Peano numbers `(x, y)` which

*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/XXXXXXX.XXXXXXX

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

**Listing 1.** Addition relation

sum up to the given value `z = S O` which are `(O, S O)` and `(S O, O)`. 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 O` 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 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 with no input arguments: add$^o$ `out out out`;
- one direction when all arguments are input: add$^o$ **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 add$^o$ `x y z` relation which does not make into this section, may be found in Appendix.

### 2.1 Basic Translation

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 **zero** and the

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

**Listing 2.** 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 ≡ 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 add$^o$ x' y z' is done in the direction 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'.

## 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. One example in which there are multiple answers is 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 3.

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.

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 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

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

**Listing 3.** Function for addo out out **in** direction

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 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 O, 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 (O, S O), (S O, S (S O)), … . 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 4 shows the 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 **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 4, 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 add$^o$ **in in in** and its functional implementation in Listing 5. 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.

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

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

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

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

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

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 Different Directions within One Function

Mult example

### 2.6 Order within Conjunctions

Example where order matters

## 3 Related Work

Previous work [1]

## 4 Future Work

## 5 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.

```
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 6.** 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 7.** 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 8.** 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 9.** Function for addo out **in in** direction

## A Principal Directions of the Addition Relation