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, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Here is going to be an introduction.

2 Modes

Any relation comes with a multitude of *modes*, or directions. The mode, among other things, specifies which arguments of the relation are input, and which values are to be computed. In some systems, such as Mercury, modes also specify the determinism of a relation: whether or not it is supposed to compute one or many values. There are many other different types of determinism, one can read more about this here.

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

Listing 1. Addition relation

A relation with a specified mode can be viewed as a function which maps its bound variables into its free variables. Consider a relation add^o `x y z` in Listing 1 with the mode `(in, in, out) is det`. This mode means that the arguments `x` and `y` are known (are input), the value of `z` is computed, and there must be exactly one value of `z` for every distinct pair of values of `x` and `y`. This mode corresponds to the addition. The mode `(out, out, in) is nondet` means that the value of `z` is known, the values of `x` and `y` are computed. We use `nondet` here, since there may be a multitude of possible pairs of `x` and `y` which sum up to a given value `z`. Notice that at least one such pair exist for any value of `z`, but we do not explicitly specify it.

Modes cannot be computed exactly, only over-approximated find a citation for some Mercury paper. Mercury allows to only specify modes for top-level predicates and functions, and the uses abstract interpretation to inference other modes. It is not unusual to have several modes for a single predicate. A predicate with a given mode may directly or indirectly use the same predicate with different modes. In this case, several functions are generated for each mode of the predicate.

3 Translation by Examples

In this section we present a scheme for translation of a miniKanren relation with a given mode into pure functions. We start by exploring examples which are translated straightforwardly, and then consider aspects of relational programming which complicate translation considerably. We will then describe how translation scheme must be adjusted to incorporate these complicated features.

3.1 The Simplest Case

For a relation add^o `x y z` with the mode `(in, in, out) is det` we can easily construct a pure function which has the same semantics: see Listing 2. We construct this function using the following thinking. The relation add^o is comprised of a single disjunction (**conde**). Each disjunct involves a unification of a known (**in**) variable x. These unifications of x are disjunct: there may only be one succeeding disjunct in the **conde** when computed with a particular ground value of x. This naturally translates into a pattern matching on x with two possible branches: either x is **zero** or a **succ** of some other value. The bodies of both branches are then generated using the remaining conjuncts in the conde branches. The unifications of z rule how the result can be constructed when other variables are known. The body of the first branch of the pattern match is thus just y. The second branch of the disjunction involves a recursive call to the add^o relation and a unification of z. A recursive call to add^o relation is done in the mode (**in**, **in**, out) since y is known and x' is a unification with a known variable. This means that here we can do a recursive call to the function add_x_y. The only thing left is constructing a resulting value which corresponds to z by applying **succ** to the result of the recursive call.

3.2 Nondeterminism

The simplest translation scheme works, but in a very small number of potential modes. First of all, the mode must be deterministic for that translation scheme to work. If there are multiple answers, then we have to express nondeterminism somehow. One natural way is to use the simplest nondeterminism monad: a list, to represent the resulting value.

Consider the same relation add^o x y z, but with the mode (out, out, **in**) is nondet. This case is about finding all pairs of values which sum up to the given z. Querying this relation with the value of z = **succ zero** must compute two answers for x and y: (**zero**, **succ zero**) and (**succ zero**, **zero**). The first answer comes from the first branch of the **conde**, when z is unified with y, and x is zero. The second answer is computed in the second branch of **conde**, after a recursive call to add^o is done with the argument equal to **zero**.

Notice that the value of z do not discriminate the two branches of **conde**. We know that z cannot be **zero** in the second branch, but the first branch do not restrict the value of z at all. This means that when z is zero, answers should be generated from both branches of **conde**. One way of implementing this is shown in Listing 3. Here we concatenate the results which are generated by the first and the second branches of **conde** via a list concatenation operator @. The first branch universally provides a single answer (0, z), while the other branch does a recursive call to the add_z function and then applies **succ** constructor to x.

3.3 Infinitely Many Answers

Lists serves well as an abstraction to capture nondeterminism. However, there are infinitely many answers sometimes which may pose a problem in case of eager host languages such as OCaml. Consider add^o x y z with the mode (out, **in**, out) is nondet. Here we compute all values x and z that x + y = z when y is given. Although the implementation of this function is straightforward (see Listing 4), it contains infinite recursion which generates infinite number of answers. When the host language is lazy, one can force only the first n answers, while in an eager host language additional care must be taken.

Move to a subsection about miniKanren streams. The problem of the infinite number of answers gets more complicated when the user expects only a single answer, but computing it involves intermediate infinite list. In this case there is no way to just force only the first answers, they all have to be considered. Example.

3.4 Generators

Some relations in some modes do not restrict certain variables at all. Consider the relation lookup^o xs i v in which the i-th element of the list xs is v (see Listing 5). When run with the mode (out, out, **in**) is nondet this relation computes all list-index pairs, such that one element of the list is fixed and given. No restrictions exist for any other elements of the list, neither the length of the list is known. In miniKanren this can be expressed as a reified value (i, _.0 :: ... :: _.(i − 1) :: v :: _.i), where v is the input ground value. This is not a ground term, and to provide a set of answers for the end user we need a way to generate the fresh elements of the list which come before the given value, as well as the tail of the list which comes after it.

One extra caveat is that we need to take care of what values we generate. In this example, the index must be a number, while the elements of the list have the same type as the given value v, assuming the host language is strictly typed such as Haskell or OCaml. Once we determined the types of the values to be generated, then we may use some automatic system for generating values by types, such as used in property based testing, for example. However, if there is no type restriction on a fresh value, or the host languages is not typed, such as Scheme, the only way to choose a proper generator is to as the user to provide it. For now, we decided to use user-provided generators always, but we will try to lift this restriction whenever possible in future work.

3.5 Interleaving Answers

It is easy to deal with an infinite number of answers of a top-level relation: laziness and forcing only a finite number of answers does the trick. If some intermediate computation constructs an infinite number of answers, then matters become more complicated. Consider the eval^o fm st result relation on Listing 6. It evaluates a propositional formula fm with a variable mapping st to get the result. Formula may be either a boolean literal lit b, a variable var i, indexed by a number, a negation of another formula neg n, a conjunction or disjunction of two formulas: conj x y and disj x y correspondingly. A variable mapping st is just

```
let rec add_x_y x y =
  match x with
  | zero → y
  | succ x' → succ (add_x_y x' y)
```

Listing 2. Functional representation of `addo x y z` with the mode `(in, in, out) is det`

```
let rec add_z z =
  [(0, z)] @
  match z with
  | succ z' → List.map (λ x y → (succ x, y)) (add_z (z − 1))
  | _ → []
```

Listing 3. Functional representation of `addo x y z` with the mode `(out, out, in) is nondet`

```
let rec add_y y =
  [(0, y)] @ List.map (λ x z → (succ x, succ z)) (add_y y)
```

Listing 4. Functional representation of `addo x y z` with the mode `(out, in, out) is nondet`

```
let rec lookup xs i v =
  fresh (h tl) (xs ≡
h :: tl) ∧ conde [
    (i ≡ zero ∧ h ≡ v) ;
    (fresh (k) (i ≡
succ k ∧ lookup tl k v))
  ]
```

Listing 5. Lookup relation

a list, `i`-th element of which is a boolean number assigned to a formula `var i`. The mode `(out, in, in) is nondet` corresponds to generation of formulas which evaluate to the given `result` with the given variable mapping `st`. In case of conjunctions and disjunctions, we first need to generate its two subformulas and then apply the corresponding constructor. There are infinitely many formulas which evaluate to the given result which entails infinite number of subformulas. In order to start generating the second subconjunction, one needs to finish generating all possible first subconjunctions which is impossible when using lists as an abstraction for several answers.

An answer to this conundrum already exists in miniKanren. The answers computed by different intermediate computations can be organized as Streams and then interleaved, so that new answers could be constructed. This change does not really change anything major in the implementation, but allows for better dealing with infinite streams of answers

4 Related Work

Previous work [?]

5 Future Work

6 Conclusion

Acknowledgments

Here is where acknowledgments come

References

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

```
let rec evalᵒ fm st result = conde [
  (fm ≡ lit b  ∧  result ≡ b);
  (fm ≡ var i  ∧  lookupᵒ st i result);
  (fm ≡ neg x  ∧  notᵒ v result  ∧ evalᵒ x st v);
  (fm ≡ conj x y  ∧  andᵒ v w result  ∧ evalᵒ x st v  ∧ evalᵒ y xs w);
  (fm ≡ disj x y  ∧  orᵒ v w result  ∧ evalᵒ x st v  ∧ evalᵒ y xs w)
  ]
```

Listing 6. A simple evaluator relation

```
let rec appendᵒ xs ys zs = conde [
  (x ≡ []  ∧  y ≡ z);
  (fresh (h t z')
    (x ≡ h :: t  ∧
     z ≡ h :: z'  ∧
     appendᵒ t y z') ) ]
```

Listing 7. List concatenation relation