An Introduction to the miniKanren Programming Language

Benjamin W. Amato
Department of Computer Science
University of Dayton
Dayton, Ohio
amatob1@udayon.edu

ABSTRACT

We provide an introduction to miniKanren—a family of relational programming languages.

KEYWORDS

miniKanren, logic programming, relational programming.

ACM Reference Format:
Benjamin W. Amato. 2019. An Introduction to the miniKanren Programming Language. In *Proceedings of CPS 452: Emerging Programming Languages (CPS 452)*, Saverio Perugini (Ed.). ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The langauge miniKanren is a family of relational programming languages first defined in 2005. The core of miniKanren is simple, with just three operators needed for the most simplified language definition. This simplicity allows miniKanren to be embedded into other languages, with library implementations of miniKanren available for over forty languages[1]. This core language can be extended to offer to simplify development and add new features. Programmers are able to define the relationship between inputs and outputs, enabling the program to generate the outputs which follow from the given relationships. A unique feature of miniKanren is the ability to run 'backward', generating the inputs which cause a specified output.

2 LOGIC AND RELATIONAL PROGRAMMING

The *logic* and *relational* paradigms are related programming paradigms which exist as a subset of the declarative paradigm. In *declarative* programming, a developer specifies what should be computed without specifying how that computation takes place[3]. Logic programming allows a user to take predicate logic and represent it in a way a computer can understand. Once the logic is defined, the computer will search for the answer, without the actual steps in the search being specified by the user. `Prolog`, a popular logic language, takes the predicates defined by the programmer, and attempts the find an answer efficiently by making some assumptions and skipping some possibilities. Byrd argues that this makes logic languages like `Prolog` impure, since these operations can cause `Prolog` to never reach a valid result, and many of these features are non declarative[2]. Byrd considers relational programming languages as the subset of logic programming languages which ensures that a program will always fail if there is no answer, always succeed if there is an answer, and only contains declarative functionality[4].

CPS 452, Fall 2019, University of Dayton, Dayton, Ohio 45469—2160

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Summary of primitive operators.

Operator	Function
==	Unifies two values
fresh	Introduces new variables
conde	Introduces multiple execution paths

3 CORE MINIKANREN DEFINITION

The core definition of miniKanren includes just three operators: `==`, `fresh`, and `conde`. Most relationships can be defined with just these three operators, but there are extensions to the core of miniKanren which enable additional functionality. Along with these operators, there is the `run` command which allows the host language to evaluate the miniKanren expression. Because the original miniKanren interpreter was written in Scheme, most miniKanren implementations use Lisp-like syntax. In this paper, we demonstrate miniKanren using a variant for Racket[1].

3.1 Unification

Unification, accomplished by the == operator, 'unifies' two terms, ensuring they have the same value for the duration of a run. Unification can unify variables, lists, and atoms together. This example shows the unification of the output variable `q` to the number `2`:

```
(run 1 (q) (== q 2))
```

The output is the list `'(2)` since the only possible value `q` can have is `2`. Note that unification is commutative, so the example below is equivalent to the previous example:

```
(run 1 (q) (== 2 q))
```

If there are multiple conflicting unifications, such as in the example below, the run will fail and and output the empty list `'()`.

```
(run 1 (q) (== q 2)(== q 3))
```

Likewise, a variable cannot be a part of both parameters to unification. The example below attempts to unify q to the list `'(,q . 1` and fails with output `'()`.

```
(run 1 (q) (== q `(,q . 1)))
```

[1] Available at https://github.com/miniKanren/Racket-miniKanren.

3.2 Fresh

Additional variables can be introduced using the operator `fresh`. The new variables can be unified to the output value, lists, and atoms. This example demonstrates a chaining of unifications using `fresh` variables:

```
(run 1 (q) (fresh (x y)
                  (== x 4)
                  (== y x)
                  (== q y)))
```

The output to this expression is the list `'(4)` since `q` is unified to `y` which is unified to `x` which is unified to `4`.

3.3 Conde

The final core operator, `conde`, allows the result to be one of multiple possibilities. Each clause in a `conde` is evaluated independently. Even if the first clause results in a valid output, the other results are still evaluated, allowing multiple outputs to be generated. The example below illustrates this:

```
(run 2 (q) (conde
            ((== q 1))
            ((== q 2))))
```

The result is the list '(1 2) since both `((== q 1))` and `((== q 2))` are evaluated seperatly, resulting in two answers.

3.3.1 How to use `run`. To understand `conde` it is helpful to explore the `run` command. Three arguments must be given to `run`: a number of runs, a variable, and the miniKanren code on which to execute. With `conde`, there can be multiple outputs when executing a segment of code, therefore `run` must be given the maximum number of outputs to obtain. If there are fewer possible results than the number given, then the program will exit with all possible results. The command `run*` indicates to the engine to continue looking for more results until all possibilities are exhausted. MiniKanren does not guarantee the order results will be found, therefore if fewer runs are requested than possible results, then the results could be different between runs or on different systems. If the previous example was run with the command `(run 1 ...)`, the output would have been either `'(1)` or `'(2)` since either the first or the second clause could return. If the previous example was run with the command `(run 3 ...)` the result would still be `'(1 2)` since all possible results are exhausted before the third run occurs.

3.4 Common Language Extensions

There are extensions to miniKanren which include additional operators to make programming in miniKanrnen easier and more powerful. Three of the most common are: `=/=`, `symbolo`, and `numbero`. These three operators ensure the operator is not a particular set of values. Dis-equality, `=/=`, ensures that a given variable is not a particular value. Since miniKanren is effectively typeless, the core language does not have any operators which dictate the type of a varaible. The common extensions `symbolo` and `numbero` give miniKanren user some ability to declare datatypes. The operator `symbolo` indicates that a variable must be a symbol, like 'foo, and not an number or list. The operator `numbero` indicates that a variable must be a number, not a symbol or list. These additional operators allow for more simple programs.

4 AN ILLUSTRATIVE EXAMPLE

An analysis of the program `rembero` demonstrates how the three core operators can be used in conjunction. `Rembero` is a variation on the classic Scheme program `rember`, which removes the first instance of an atom from a list. By convention, miniKanren functions end in `o` to differentiate them from functions written in the host language. The following code is adapted from the Byrd and Freeman's talk "miniKanren Philospohy"[5] and "The Reasoned Schemer".

```
(define rembero
  (lambda (x l out)
    (conde
      ((== '() l)(== '() out))
      ((fresh (d)
              (== `(,x . ,d) l)
              (== d out)))
      ((fresh (a d res)
             (== `(,a . ,d) l)
             (=/= a x)
             (== `(,a . ,res) out)
             (rembero x d res))))))
```

`Rembero` takes in three arguments: the atom to search for, the list to search through, and the output. After declaration of the function, there is a `conde` operator, indicating that any of the next three clauses could be valid.

The first clause attempts to unify the empty list to the input list. If this unification is successful, then there is no answer, and the output is bound to the empty list. The second clause introduces an additional variable `d` with `fresh`. It then attempts to unify the list `l` with `x` and `d`. This is possible if the head of the list is the same as the input `x`. The tail of the list is unified to the variable `d`. If that unification is successful, then the output is bound to `d`, returning the list without the matching variable. The third clause uses recursion to search though the remainder of the list. Three new variables are introduced: `a`, `d`, and `res`. The variables `a` and `d` are unified to the head and tail of the list respectively. The variable `a` is dis-equaled with `x` to ensure that the recursion stops if the second clause finds a value. Skipping over the third line, the fourth line recursively calls `rembero` on the tail of the list, the output of the recursive call being stored in `res`. The third line builds a list with the head being `a` and the tail being the result of the recursive call, and unifies this list with `out`.

4.1 A run of `rembero`

`Rembero` can be run to get a list without the first occurrence of a value.

```
(run* (q) (rembero 'b '(a b c b) q))
```

The output is `'((a c b))` since this only possible list that can be formed by removing `b` from the list `'(a b c b)`.

4.1.1 Running it 'backward'. Since all relations in a miniKarnen program are commutative, an interesting feature of miniKanren is the ability to run the function 'backwards' to get the inputs that result in a specific output, rather than the output from a particular input. The run below demonstrates this:

```
(run* (q) (rembero 'b q '(a c)))
```

The unknown variable which miniKanren must generate replaces the output rather than the input. Running this results in the list:

```
'((b a c) (a b c) (a c) (a c b))
```

This is a list of all the possible inputs which would result in the output `'(a c)`. This demonstrates the property of true relational programs, the inputs and outputs are treated the same by the system, allowing for more powerful functions.

5 CODE GENERATION USING MINIKANREN

The utility of this 'backward' running is demonstrated "miniKanren, Live and Untagged"[6]. By writing a Scheme interpreter in miniKanren, Byrd et al. are able to both evaluate and generate Scheme code. Setting the `run` variable as the output of the evaluation, Scheme expressions could be evaluated. By replacing the Scheme expression with the `run` variable and inputting a possible output, and miniKanren generated a theoretically infinite number of programs which would result in that output. Lastly, tquines — programs which produce themselves as outputs, could be generated by setting both the input and the output to the `run` variable.

6 CONCLUSIONS

Though its relational programming constructs and simple yet powerful grammar, miniKanren allows for a unique form of logic programming which can be directly embedded inside other languages. Its powerful relational constructs give it the unique ability to run 'backwards', which can be used for generation of valid inputs and programs

7 EXERCISES

7.1 Absento

Many extensions to miniKanren include the absento operator. This operator accepts two inputs: an element and a list. If the element is in the list then then absento fails. If the elements in not in the list, absento succeeds. Implement absento in Racket-miniKanren. Examples:

```
(run 1 (q) (absento 7 '(1 2 3)))
>> '(_.0)
(run 1 (q) (absento 2 '(1 2 3)))
>> '()
(run 1 (q) (absento q '(1 2 3))(== q 5))
>> '(5)
(run 1 (q) (absento q '(1 2 3))(== q 3))
>> '()
(run 1 (q) (absento q '('foo 'bar)))
>> '((_.0 (=/= ((_.0 quote bar)) ((_.0 quote foo)))))
(run 4 (q) (absento 3 q))
>> '(() ((_.0) (=/= ((_.0 . 3)))) ((_.0 _.1) (=/= ((_.0 .
      3)) ((_.1 . 3)))) ((_.0 _.1 _.2) (=/= ((_.0 . 3))
     ((_.1 . 3)) ((_.2 . 3)))))
```

7.2 Boolean Expression

Relational languages like miniKanren can be used for describing first order predicate logic. Define a miniKanren function which encodes the binary expression $(x \wedge y \wedge z) \vee (x \wedge y) \vee (x \wedge z))$ where $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT. The function should accept 3 variables, x, y, z, and should fail if given an invalid input and should succeed if given a valid input. Examples:

```
(run 1 (q) (binExp 1 1 1))
>> '(_.0)
(run 1 (q) (binExp 1 0 0))
>> '()
(run 1 (q) (binExp q 0 0))
>> '(0)
```

REFERENCES

[1] [n. d.]. miniKanren. http://minikanren.org/. Online; accessed: 15 October-2019.

[2] 2014. William Byrd on Logic and Relational Programming, miniKanren. https://www.infoq.com/interviews/byrd-relational-programming-minikanren/. Online; accessed: 15 October-2019.

[3] K. R. Apt. 1996. Logic Programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)* (1996), 493–574.

[4] W. Byrd. 2009. Relational Programming in miniKanren: Techniques, Applications, and Implementations. (09 2009).

[5] W. Byrd and D. Friedman. 2013. miniKanren Philosophy - William Byrd & Daniel Friedman. https://www.youtube.com/watch?v=fHK-uS-Iedc&t=1533s. Online; accessed: 15-October-2019.

[6] W.E. Byrd, E. Holk, and D.P. Friedman. 2012. miniKanren, live and untagged. In *Proceedings of Scheme Workshop*, Vol. 12.