Minikanren Visualizer

by

Vladimir Turov

Bachelor Thesis in Computer Science

Submission: May 15, 2023 Supervisor: Daniil Berezun

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Turov, Vladimir
Matriculation number	30006614
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

....

Date, Signature

Abstract

Logic programming has gained much more interest in the past decade and miniKanren has been a big part of it. However, despite the fact that miniKanren as a concept is easier to understand than traditional logic languages such as prolog and mercury, miniKanren and logic programming itself still remain challenging for beginners.As a result, it still does not have as many adherents as it potentially could. MiniKanren visualizer is created to make a difference and provide a tool which would help new people in relational programming community to easy understand miniKanren concepts and engage with logic programming. Moreover, this utility can help knowledgeable people to achieve better results in less time. This thesis presents the development and evaluation of a miniKanren visualizer, an interactive tool designed to aid users in better comprehending the processes and constructs of miniKanren programs.

TODO: Looks like I am missing something. Dig inside my head. (target size: 15-20 lines)

Contents

1 Introduction

The main purpose of this thesis is to introduce a useful tool for beginners in the field of logic programming. The primary issue with miniKanren is that it is difficult to debug and even harder to track all the relations and search strategies. That is why developing a visualizer will be highly beneficial. However, this tool is not supposed to be only for newcomers; it is also designed to be useful for experienced users in the field. In a more advanced capacity, it can help experienced users by showing the debugging process and providing a clear overview of program execution, which for sure will save time and effort.

One of the main problems addressed in this thesis is how to create a visualizer that is both fully functional and easy to understand for users with different levels of experience. Additionally, we want to explore if a visualizer has already been implemented in this field. Although it is highly unlikely that someone has developed one specifically for miniKanren, visualizers for languages miniKanren is built upon could potentially be useful. The usefulness of such visualizers comes from the fact that they can provide insights into the design and implementation of a miniKanren-specific visualizer.

Despite the abundance of material on miniKanren, there is limited information about visualizers in this field, which makes this project highly unique and potentially challenging. This is the second problem to address. Lastly, since miniKanren is a domain-specific language, we need to determine which concrete language to focus on and the rationale behind this choice. This is the final problem to research. As we can see, there are several aspects that need to be addressed during the research process.

This thesis is divided into several parts.

In the first section, titled Background, we will provide an overview of the field to make this thesis accessible to readers who may not be familiar with miniKanren or logic programming. We will offer a concise definition of the miniKanren language and compare it to traditional relational programming languages, such as Prolog, in the context of differences in approaches to creating visualizers. Furthermore, we will present a detailed explanation of the miniKanren concept, discuss relevant work in this field, and explore how it can inform the visualizer's implementation. Additionally, we will examine the languages miniKanren was built upon, determine which one is best suited for our purposes, and explain our choice.

The second section, Evaluation, will be the most engaging part of the thesis. This part will detail our proposed approach to implementing the visualizer, including several subtasks that will be discussed individually but are directly related to the main task. We will also provide a brief description of the technologies used in the visualizer's development. In this section, we will discuss instances where our initial approach may have been incorrect or suboptimal. While these instances may not be directly relevant to the final outcome, they could provide valuable insights for others working on similar projects in related areas.

The final section, Results, will showcase the culmination of our work. As one might expect, this part will not only present the actual visualizer but also summarize the entire research process, providing a comprehensive overview of the work.

I hope you enjoy reading this thesis and gain valuable insights from it. Have a great time exploring my findings.

2 Background

2.1 Welcome to the world of miniKanren

As much as I want to believe that miniKanren is enough popular or intuitive language that I do not need to explain it I realize that without proper insight I will not be able to talk freely in the next steps. I am not going to make a full examination of miniKanren language, it can be find in "The Reasoned Schemer". however, we are going to have a closer look into this.

miniKanren is an embedded Domain Specific Language (DSL) implemented in various host languages like Racket, Clojure, Haskell, and OCaml among others. The language revolves around several key concepts for the first we will primitive relations.

2.1.1 Unify

The key relation in miniKanren is unify. It serves as the most primitive relation, and almost certainly, any other sophisticated relation will employ this relation under the hood. As the name suggests, the unify relation unifies two expressions. In simpler terms, it attempts to assert that the two expressions are identical.

Examples:

```
(run* (q)
      (== 'cat 'cat))
```

In this scenario, the program returns () (as we never really did anything with q), but this program completes successfully because miniKanren performs unification of the two equal constants.

```
(run* (q)
      (== q 'cat))
```

The program returns 'cat because q, which was not associated with any values before, is now unified with the string constant 'cat using the (==) relation.

```
(run* (q)
    (fresh (a b c)
        (== `(a b c) `(1 2 3))
        (== q b)))
```

In this example, we have two successful unify relations. The first one associates a list of fresh variables (we will explain this construction later) a, b, and c with the list of 1, 2, and 3, respectively. As we can infer, this program returns the substitution q = 2. Similarly, we could unify q with a and c to get q = 1 and q = 3.

```
(run* (q)
    (== 'cat 'dog))
```

In this scenario, the program fails because, as we know, 'cat is not equal to 'dog.

```
(run* (q)
    (fresh (r)
        (== 'cat r)
        (== r 13)))
```

In this scenario, the program also fails because, as we know, r, which represents a cat, is not equal to 13.

To summarize, we can attempt to unify any two expressions, such as constants, variables, and compound terms. The unification process can either complete successfully or fail if there are any conflicts.

(Rem. 1) Changing the order of the unify arguments does not make any difference.

2.1.2 Conjunction

We have already touched upon conjunction in previous examples, but it may not have been entirely clear. In the Scheme syntax, conjunction is implicit. Essentially, when you write two or more goals in a row, you are using conjunction. While this is not the standard for every implementation of miniKanren, we will use this syntax until we begin discussing unicanren.

```
(run* (q)
    (fresh (a b c)
    (== a 'cat)
    (== b a)
    (== b c)
    (== q c)))
```

All of the goals *q* in the program *run(q) ((fresh (...) g g g g g ...))* must succeed. This is a true representation of conjunction.

2.1.3 Disjunction

For disjunction, we have a specific construction called **Conde**.

```
(run* (q)
    (fresh (r)
        (conde
            ((== r 1))
            ((== r 2))
            ((== r 3))))
    (== q r))
```

The program returns (1 2 3). Conde attempts to evaluate its first line and succeeds, so it proceeds to the unify relation and successfully evaluates it. The process is repeated two more times with the other Conde goals.

```
(run* (q)
    (fresh (r v)
        (conde
            ((== r 1) (== v 'one))
            ((== r 2) (== v 'two))
            ((== r 3) (== v 'three)
                ))
    (== q `(,r ,v))))
```

The program returns ((1 one) (2 two) (3 three)). As demonstrated, a single Conde line can contain multiple subgoals. In fact, there can be any number of subgoals within a Conde line. A Conde line succeeds if all of its subgoals succeed.

(Rem. 2) Conde succeeds if any of its lines succeed.

```
(run* (q)
    (fresh (r)
        (conde
            ((== 2 1))
            ((== 3 2))
            ((== 1 3))))
    (== q 3))
```

The program returns (), as none of the Conde lines succeeded.

If we want at least one Conde line to succeed, we can use the primitive goal *success*.

```
(run* (q)
    (fresh (r v)
        (== r 4)
        (conde
            ((== r 1) (== v 'one))
            ((== r 2) (== v 'two))
            ((== r 3) (== v 'three)
               ))
            (succeed (== v 'four))
        (== q '(,r ,v))))
```

In this scenario, all lines except the last one will fail due to the fact that r was previously associated with the number 4. As a result, the output will be (4 four).

We can set a value after the run primitive to specify the search depth. Previously, we did not set the depth, and the program searched for all possible solutions.

```
(run 2 (q)
    (fresh (r v)
        (conde
            ((== r 1) (== v 'one))
            ((== r 2) (== v 'two))
            ((== r 3) (== v 'three)
               ))
    (== q '(,r ,v))))
```

The program returns ((1 one) (2 two)). If we set the value to 1, we will obtain the result ((1 one)).

(Rem. 3) In terms of imperative programming, *conde* can be likened to an **if-else** expression. In a sense, it really is, because we can treat the left side goals as conditions and the right side goals as expressions. *succeed* in this case would be the else expression. The true and false values in the context of conde are the results of the goals *fail* or *succeed*.

2.1.4 Fresh

Now, let's examine the *fresh* primitive. As its name suggests, it introduces new or "fresh" variables. Any fresh variable can be associated with some expression or variable.

```
(run* (q)
    (fresh (r v)
    (== q '(,r ,v))))
```

The program returns ((_0 _1)), as the program does not associate *r* and *v* with any value.

In this case, variables *r* and *v* are introduced but not associated with any value, so they remain fresh.

The *fresh* primitive, as the name suggests, introduces new or "fresh" variables. Any fresh variable can be associated with some expression or variable.

```
(run* (q)
    (fresh (r v)
    (== 'cat v)
    (== q '(,r ,v))))
```

The program returns ((_0 cat)), as the program associates the variable *v* with "cat".

```
(run* (q)
    (fresh (r v)
    (== 'cat r)
    (== q `(,r ,v))))
```

The program returns ((cat _0)), as the program associates the variable *r* with "cat".

As we can see, the number of fresh variables does not depend on the order of introduction in the fresh primitive.

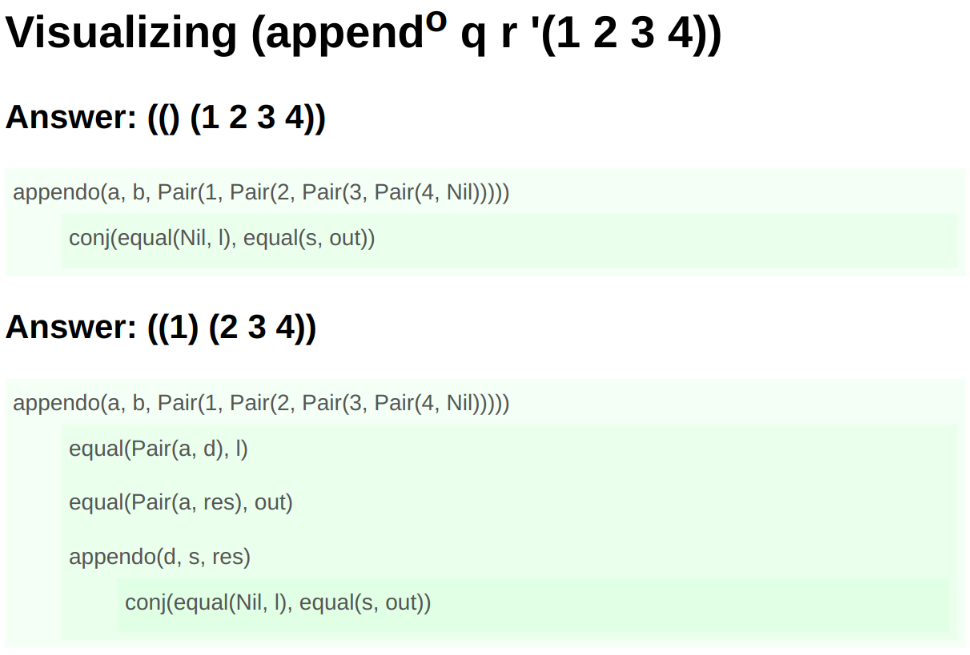
That concludes all the necessary parts of miniKanren needed for now. During the research, there will be new introductions. For now, we should focus on the analogs.

2.2 Analogs and work in the field

There are actually just a few analogs.

2.2.1 microScopeKanren

MicroScopeKanren is a variant of the miniKanren relational programming language that incorporates a microscope-based approach to debugging and visualization. It is much similar to what we are trying to aproach. The main idea behind MicroScopeKanren is to provide a tool that allows users to observe and analyze the internal state of a miniKanren program as it executes. This is achieved by capturing and displaying various intermediate states of the program, including the search tree, substitutions, constraints, and goals.



Visualizing (append^o q r '(1 2 3 4))

Answer: (() (1 2 3 4))

```
appendo(a, b, Pair(1, Pair(2, Pair(3, Pair(4, Nil)))))
    conj(equal(Nil, l), equal(s, out))
```

Answer: ((1) (2 3 4))

```
appendo(a, b, Pair(1, Pair(2, Pair(3, Pair(4, Nil)))))
    equal(Pair(a, d), l)
    equal(Pair(a, res), out)
    appendo(d, s, res)
        conj(equal(Nil, l), equal(s, out))
```

Figure 1: microScopeKanren interface

One of the features is it provides wild options observing the states of the program. It is implemented on JavaScript and it visual part is defined by HTML page so it works in a browser. It is providing step-by-step visualization which is very useful.

MicroScopeKanren can be a valuable tool for developers working with miniKanren programs, as it provides a powerful and intuitive way to observe and understand the program's execution. It allows for more efficient debugging and exploration of complex relational logic, leading to improved program understanding and development productivity.

But it it does not really what we want from miniKanren visualizer and this realization has many disadvantages:

1. It is written in JavaScript and look very different from the original Scheme implementation.
2. It does not have any appropriate documentation.
3. It works in a browser and it is not very comfortable
4. It does not have any code window and you cannot debug you own programs

These disadvantages make this solution not applicable for our purposes. But it can be used as a reference for the graphical design of our program and can bring some necessary ideas about our approach.

2.2.2 First-order miniKanren representation:

A paper exists that explores the stepper for the original miniKanren implementation and investigates its potential as a debugger and search tool. While this paper presents intriguing ideas, it may not be directly applicable to our case as it does not focus on visualization and is based on the original implementation of miniKanren.

The paper provides insights into the stepping mechanism and offers a deeper understanding of the execution process in miniKanren. It explores techniques for tracking and analyzing program execution, which can be valuable for debugging and optimization purposes. However, the lack of visualization capabilities limits its practical use as a visualizer.

In our case, the goal is to develop a visualizer that provides a graphical representation of the program execution, allowing users to observe and analyze the search process in a more intuitive and interactive manner. This requires a different approach that specifically focuses on visualizing the execution tree, substitutions, and other relevant aspects of miniKanren programs.

2.3 Unicanren

2.3.1 The reasons to choose Unicanren

In order to implement a visualizer, we must choose a specific implementation of miniKanren. As previously mentioned, we need to select an exact language implementation of miniKanren. The implementation should be easily modified to be applicable to a visualizer, be designed as close to the original Scheme implementation as possible, and be able to integrate with existing GUI frameworks.

Initially, we must determine which languages already have miniKanren implementations. MiniKanren has numerous implementations across many languages. It's important to choose a functional paradigm language to support and manipulate these implementations easily. Functional language implementations of miniKanren have multiple advantages. Firstly, they are closer to the original Scheme implementation, making them easier to inspect due to the ample information available on the original implementation. Secondly, miniKanren implementations in imperative languages tend to differ from the original implementation. Lastly, functional languages' pattern-matching capabilities are highly useful for manipulating miniKanren's complex structures.

This leaves us with two language choices: Haskell and OCaml. Although Haskell offers more miniKanren implementations, we should consider using OCaml as the base language for several reasons. Firstly, some work has already been done in this field using OCaml. Secondly, OCaml is more user-friendly. Lastly, the choice depends on the specific miniKanren implementation.

Unicanren is a miniKanren implementation designed specifically for the OCaml programming language. It builds upon the core principles and features of the original miniKanren implementation in Scheme while taking advantage of OCaml's expressive capabilities and performance characteristics.

Similar to the Scheme implementation, Unicanren provides a powerful relational programming paradigm, enabling the exploration of multiple possible solutions and facilitating the development of sophisticated search strategies.

Similarity to the original miniKanren implementation is crucial for us due to the existing documentation on the Scheme implementation.

Unicanren benefits from OCaml's ecosystem, which includes a wide range of libraries and tools for building robust applications. Developers can leverage these resources to enhance their miniKanren programs with additional functionality and seamlessly integrate them into OCaml projects.

Unicanren retains miniKanren's core syntax and principles, while its implementation in OCaml provides a powerful and efficient environment for logic programming. By combining miniKanren's relational programming paradigm with OCaml's performance and type system, Unicanren empowers us to tackle complex computational problems and reason about relationships and constraints effectively.

2.3.2 Inspecting Unicanren

Unicanren works similarly to the original miniKanren. However, we never really discussed how miniKanren works under the hood. As previously mentioned, the Unify (==) is the core relation by which we can implement all other relations. (==) is the basic goal that should be inspected for its mechanism of work.

Unify relies on associations. An association is the basic relation that, like any binary relation, can be explained as a pair. In the original miniKanren, we use (z . 'b) to show an association. The first element of the pair must be a variable, and we call it the left-hand side (lhs); the second element is the right-hand side(rhs) and can be any value.

`rhs (z . 'b)`	The value is b.
`rhs (z . w)`	The value is the variable w.

(Rem. 1) As previously mentioned, the rhs can be any value. For example:

`rhs (z . (1 w 3))`	The value is the list (1 w 3).
`lhs (z . ’b)`	The value is *z*. But this one function is unnecessary.
`(z . z)`	Even though every association is a pair, we do not allow the rhs to be the same as the lhs. Such a pair cannot be treated as an association.

The list of associations is called a substitution. For example:

`((z . ’a)) (x . w) (y . z))`	This is exactly a substitution.

As we can see, substitution is essentially a map. One might consider a function that traverses the list and finds the rhs based on the defined lhs. However, we will actually require something more powerful.

```
(define walk
  (lambda (v s)
    (cond
      ((var? v)
       (cond
         ((assq v s) ⇒
          (lambda (a)
                (walk (rhs a) s)))
          (else v)))
       (else v))))
```

This is the definition of the function walk. What exactly does this function do? In short, this function traverses the list and attempts to find *v* in the lhs of the associations. When it locates *v*, it checks whether its rhs is a variable or not. If it is not a variable, it returns this rhs; otherwise, it continues the recursion with rhs as the new *v*.

Essentially, it traverses the list multiple times before finding anything other than a variable. Although it follows a depth-first search (DFS) approach with no ideal asymptotic behavior, it is quite simple to write and explain.

```
(walk z ((z . 'a)) (x . w) (y . z))
    )
```

The result is "a".

```
(walk y ((z . 'a)) (x . w) (y . z))
    )
```

The result is "a". The function walk encounters the association (y . z) and then performs another walk with *z* as the input, eventually finding the value "a".

Substitutions can also be circular. For example:

```
(walk x ((x . y)) (z . x) (y . z)))
```

There is no value. The walk function will enter an infinite loop.

However, if we construct the substitution correctly, this situation can be avoided.

There is another problem:

```
(walk u ((x . 'b)) (w . (x 'e x)) (
     u . w )))
```

The result will be ‘(x ’e x)‘. However, we have the association ‘(x . ’b)‘, so we may expect the result to be ‘(’b ’e ’b)‘. But the ‘walk‘ function finds the first rhs that is not a variable. Therefore, we most definitely need another implementation.

Here is an alternative implementation:

```
(define walk*
  (lambda (v s)
    (let ((v (walk v s)))
    (cond
      ((var? v) v)
      ((pair? v)
            (cons
                (walk* (car v) s)
                (walk* (cdr v) s)))
       (else v))))
```

In this case, ‘car‘ is a function that extracts the head of a list, and ‘cdr‘ extracts the tail. Within this function, there is a check on the list. If the list is a pair (which is essentially any non-empty list), there are two branches where we recursively continue the operation on both sides of the pair.

As the result:

```
(walk u ((x .'b)) (w . (x 'e x)) (u
    . w)))
```

The result will be ‘(’b ’e ’b)‘, as expected.

In the original Scheme implementation, there are additional functions like ‘ext-s‘ and ‘ext-s’‘ that prevent the addition of associations that would cause loops. However, we will not delve into them as they are unnecessary for our purposes and implemented differently in Unicanren.

Now, let’s examine how ‘walk*‘ is implemented in Unicanren:

```
let rec walk subst : t -> t = function
    | Var v ->
      (match Subst.find v subst with
       | exception Not_found -> Var v
       | t2 -> walk subst t2)
    | Symbol s -> Symbol s
    | Cons (l, r) -> cons (walk subst l) (walk subst r)
    | Nil -> Nil
  ;;
```

Listing 1: Walk implementation in Unicanren

As we can see, it is even easier to implement ‘walk*‘ in OCaml due to the power of pattern matching. This partial function is typed to take a value of type ‘t‘ and return a value of the same type. In our case, our type is ‘Subst‘ (substitution). As you can observe, ‘Subst‘ is implemented as a map, which simplifies things. The implementation of ‘walk*‘ closely resembles the logic we discussed earlier. If ‘v‘ is a variable, we recurse; if it is a list, we also recurse; otherwise, we simply return the input.

Now, let’s explore the ‘unify‘ function. In this case, inspecting the original implementation may not be as useful as it differs significantly from the implementation in Unicanren and can be more challenging to comprehend. Here is the original code from Unicanren:

```
let rec unify acc x y =
  match Value.walk acc x, Value.walk acc y with
   | Value.Var n, Value.Var m when n = m -> Some acc
   | Var n, (Var _m as rhs) -> Some (Subst.add n rhs acc)
   | Symbol m, Symbol n when n = m -> Some acc
   | Symbol _, Symbol _ -> None
   | Nil, Nil -> Some acc
   | rhs, Var n | Var n, rhs -> Some (Subst.add n rhs acc)
   | Cons (l1, r1), Cons (l2, r2) ->
     let open Base.Option in
     unify acc l1 l2 >>= fun acc -> unify acc r1 r2
   | Symbol _, Cons (_, _)
   | Cons (_, _), Symbol _
   | Nil, Cons (_, _)
   | Cons (_, _), Nil
   | Symbol _, Nil
   | Nil, Symbol _ -> None
;;
```

Listing 2: Unify implementation in Unicanren

In this scenario, ‘acc‘ represents the substitution. The ‘unify‘ function covers several cases, explaining all the possible variants. When the constraints are not applicable by definition, it returns ‘None‘. If both ‘x‘ and ‘y‘ are symbols, it returns the same substitution if they are equal, or ‘None‘ otherwise. The same applies to variables; if they are equal, there is no need to make any changes. Otherwise, a new association is added to the substitution. If there are two lists, they are split into their heads and tails, and the unification is performed recursively on each corresponding pair.

Lastly, there is the ‘eval‘ function.

```
let eval ?(trace_svars = false) ?(trace_uni = false) ?(trace_calls = false) =
  let open State in
  let open StateMonad in
  let open StateMonad.Syntax in
  let rec eval root : (st, subst Stream.t) StateMonad.t =
    match root with
    | TraceSVars xs ->
      ...
    | Unify (l, r) ->
      ....
    | Conde [] -> assert false
    | Conde (x :: xs) ->
      ...
    | Conj [] -> assert false
    | Conj [ x ] -> eval x
    | Conj (x :: xs) ->
      ...
    | Fresh (name, rhs) ->
      ...
    | Call (fname, args) ->
      ...
  and eval_term = function
    | Nil -> return Value.Nil
    | Symbol s -> return (Value.symbol s)
    | Cons (l, r) -> return Value.cons <*> eval_term l <*> eval_term r
    | Var s ->
      let* next = lookup_var_syntax s in
      (match next with
       | None -> fail (‘UnboundSyntaxVariable s)
       | Some t2 -> return t2)
  in
  eval
;;
```

Listing 3: Eval implementation in miniKanren

```
// Simple JavaScript code example
function greet(name) {
    console.log("Hello, " + name + "!");
}
```

Listing 4: JavaScript Example

This is a part of the evaluator, which is the main component of Unicanren. The function is recursive, and some of the relations, such as Call, Unify, or TraceSVars, terminate. Others depend on the evaluation of their sub-goals. The eval function takes the root of the program and returns a stream of substitutions. These substitutions become the solutions in the end. For now, it is important to note that by adding a logger within these cases, we can gather all the necessary information to build a tree.

Now, let's examine how we can approach solving this problem.

3 Approach

There are different ways to integrate GUI with OCaml. The easiest option is to use Js_of_ocaml to compile OCaml code into JavaScript and import it into QML to build the GUI. The second option is to use **lablqml** and use OCaml as the main language for deploying the main application. The first option is preferred because there is existing information available online, and Js_of_ocaml is easier to use compared to **lablqml**, which has limited documentation.

Js_of_ocaml is straightforward to use as it allows calling external JavaScript functions from OCaml code. This is achieved by translating OCaml code to JavaScript. Thus, we can write a logger in JavaScript and easily integrate it into QML.

3.1 GUI

Since I have chosen to use Js_of_ocaml, it is necessary to select a compatible framework for working with JavaScript. In this case, Qt/QML is the most suitable framework for our purposes. Now, let's discuss the steps that need to be taken to design the visualizer for miniKanren in the context of using Js_of_ocaml and Qt/QML.

3.1.1 Design Steps

To begin designing the visualizer, we need to consider the following steps:

1. Define the basic design: Start by determining the overall design of the visualizer. This involves deciding how the program execution will be graphically represented. Since miniKanren programs resemble trees, it is logical to represent the execution as a tree structure. However, there are various ways to visualize a tree. It can be represented as a graph with nodes and arrows, or it can be presented in a hierarchical manner similar to how file systems are displayed. As well as visualization of execution it should be definitely a window for the code that is going to be executed.
2. Identify applicable tools: Determine the tools and techniques that can be used to implement the desired visual representation. This may involve exploring different libraries or frameworks that provide tree visualization capabilities. Consider the features, documentation, and community support offered by these tools to ensure their compatibility with Js_of_ocaml and Qt/QML.
3. Design the JavaScript interface: As we are using Js_of_ocaml to compile OCaml code into JavaScript, it is essential to design the JavaScript interface that will facilitate the communication between the OCaml code and the Qt/QML GUI. This interface should enable the exchange of data and function calls between the two components.

3.1.2 Basic Design

miniKanren programs can be very deep and tree in the traditional form of graph is not very scalable. In opposite, if we represent tree by the TreeView model it can be easy scalable as we can hide and reveal any nodes we want. In QML there is the basic entity TreeView it need only model which can be implemented in Qt. There are not many requirements for the code window. Basically, it only needs text field and numeration. Maybe syntax highlight.

As well we need to to decide how our program will look like. Basically there are few concepts that we may use.

Our program may look like IDE. In this scenario we often need such components:

1. Main code window. Maybe with tabs on the top.
2. File system bar on the left.
3. Tool bar at the top.
4. Debug tools at the bottom.

But at this case there are a lot of elements that are not actually needed at this point. On the other side we need a lot of space for our tree representation of an execution. So it becomes almost obvious that we do not need file system sidebar because it will be an issue to navigate between them. So we basically need only tree representation side and code editor and they must have as much space as possible. That leaves as with the following prototype:

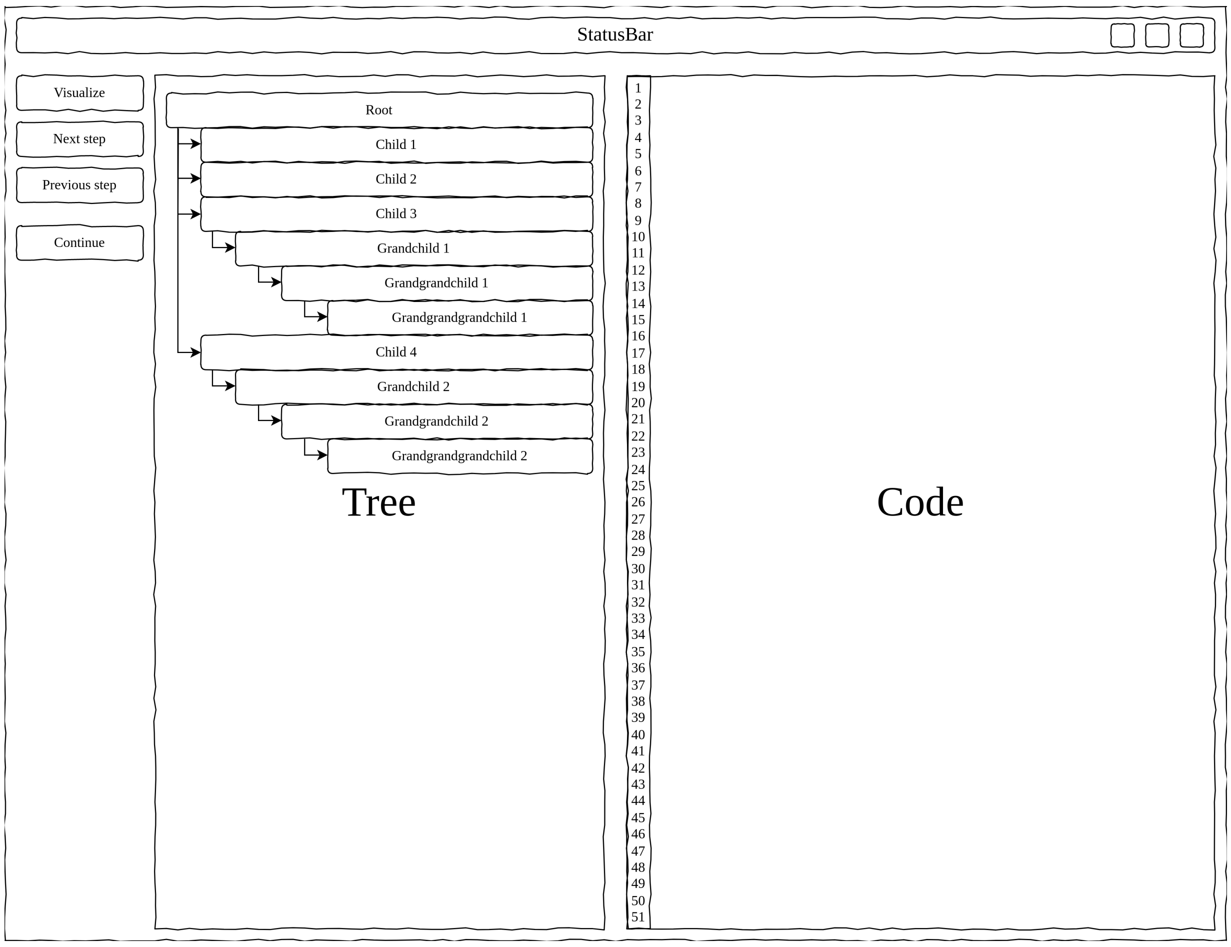


Figure 2: Visual Design Prototype

As we can see it has all necessary tools and buttons. But there is a problem. Where will the result show? We never actually defined output window. Actually, if this is a side tool it is not really necessary. But if we should lets add.

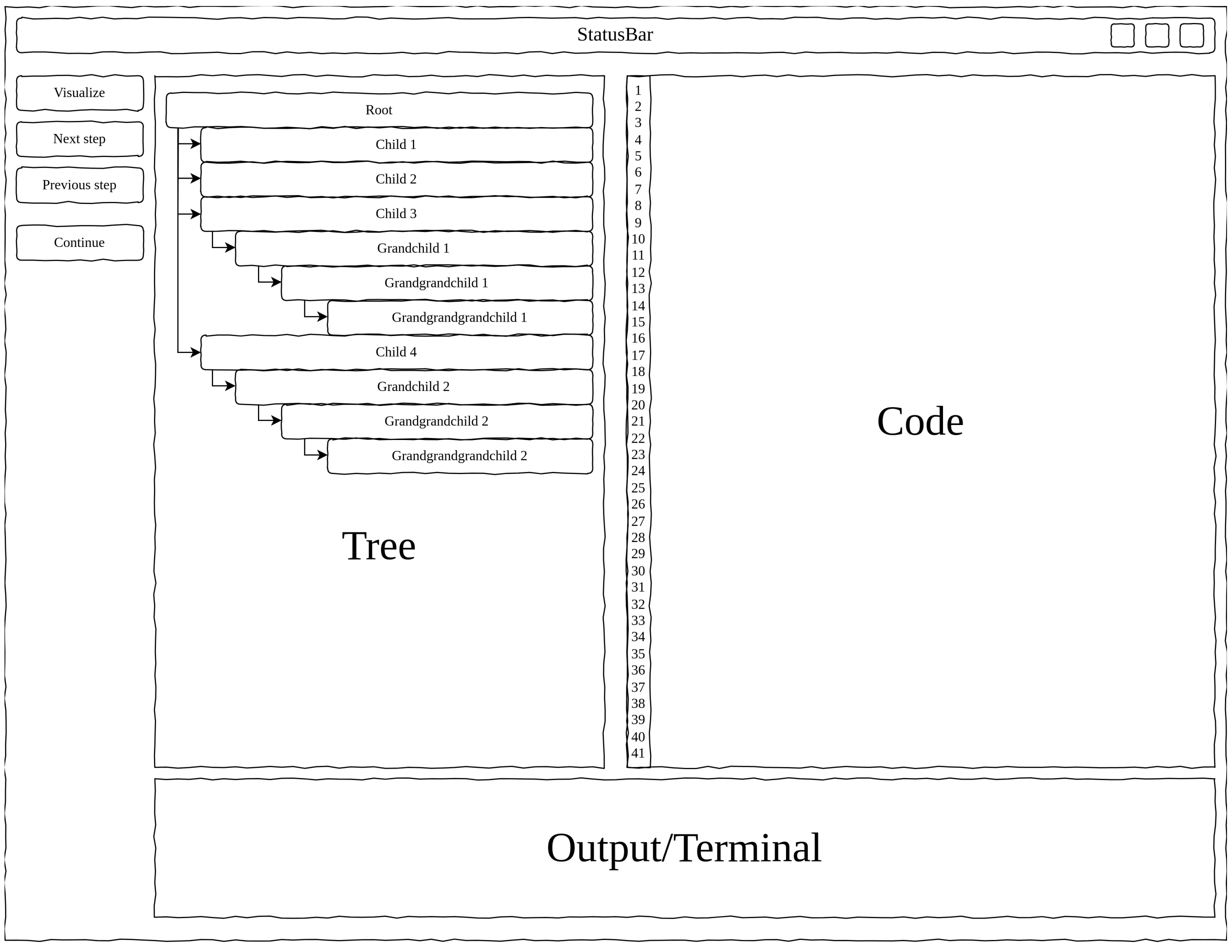


Figure 3: Visual Design Prototype 2

This one looks fine. We should use this as a goal.

3.1.3 GUI implementation

We already defined basic interface of interacting with OCaml. There is left to implement the GUI. Lets start from implementing the tree representation. As well we need to choose the QML entity and implement a model for interacting with the tree. For this purpose we should choose TreeViev QML entity. This one is supposed to be used in file hierarchy entities. But there is a problem that it does not have standard Qt models that would be applicable for drawing anything beside file hierarchy. So that is the problem we should solve.

We should provide a model that would be nice for us to use. TreeView could use any Qt model that are derived from QAbstractItemModel. QAbstractItemModel is obciously an abstract class with following interface:

```
bool hasIndex(int row, int column, const QModelIndex &parent = QModelIndex())

virtual QModelIndex index(int row, int column,
    const QModelIndex &parent = QModelIndex())

virtual QModelIndex parent(const QModelIndex &child)
virtual QModelIndex sibling(int row, int column,
    const QModelIndex &idx)

virtual int rowCount(const QModelIndex &parent = QModelIndex())

virtual int columnCount(const QModelIndex &parent = QModelIndex())

virtual bool hasChildren(const QModelIndex &parent = QModelIndex()) const

virtual QVariant data(const QModelIndex &index, int role = Qt::DisplayRole)

virtual bool setData(const QModelIndex &index, const QVariant &value, int role =
    Qt::EditRole)

virtual QVariant headerData(int section, Qt::Orientation orientation, int role =
    Qt::DisplayRole)

virtual bool setHeaderData(int section, Qt::Orientation orientation, const QVariant
    &value, int role = Qt::EditRole)

virtual QMap<int, QVariant> itemData(const QModelIndex &index)

virtual bool setItemData(const QModelIndex &index, const QMap<int, QVariant> &roles)
```

Listing 5: QAbstractItemModel Interface

We must define all of it for our purposes. There are two interesting methods: addNode and stepUp. Others just support Qt model.

```
void TreeModel::addNode(const QString &str) {
    beginResetModel();
    QList<QVariant> columnData;
    columnData << newCustomType(str, 0);
    auto childItem = new TreeItem(columnData, lastNode);
    lastNode->appendChild(childItem);
    lastNode = childItem;
    endResetModel();
}
```

Listing 6: addNode Implementation

```
void TreeModel::stepUp() {
    beginResetModel();
    lastNode = lastNode->parentItem();
    endResetModel();
}
```

Listing 7: stepUp Implementation

```
    addNode(title) {
       this.canvas.model.addNode(title);
    }

    end() {
        this.canvas.model.stepUp();
    }
```

Listing 8: JS Part

There will be no explanation of other functions because there are just defining model logic and have no connection to the main theme.

As well we should implement code editor. For our purposes there will be enough just QML canvas. The rest of logic might be implemented through JavaScript and redirected straight to the OCaml interpreter.

```
module UserInterface = struct
  open Language
  open Interpreter

  let eval_program (prog : string) =
    let open StateMonad in
    let res = match interpretor prog with
        | Interpret.Ok g ->
        (match StateMonad.run (eval true true true g) State.empty with
            | Result.Ok r -> Stream.take ~n:(-1) r
            | Result.Error e -> failwiths "Error: %a" pp_error e)
        | Interpret.Error e -> failwiths "Error: %a" pp_error e
    in List.iter (fun st -> Js.Unsafe.global##.resultf (Js.asprintf "%a\n%!"
        (Subst.pp Value.pp) st) res
    ;;
```

Listing 9: Interpreter prototype

Result will pass via JS interface connection to QML output window. Js.Unsafe.global##.resultf calles the external function from our JS interface.

(target size: 5-10 pages)

4 Results

1. Code Window: The code window component was successfully implemented, providing a user-friendly interface for inputting and editing MiniKanren programs. Users can conveniently enter their code, navigate through it, and make modifications. The prototype parser and lexer allow for basic error detection during the build process. Although syntax highlighting was not included in the current implementation, it can be considered as a future enhancement.

2. Tree Representation: The tree representation of the MiniKanren search tree was implemented. As the program executes, the visualizer displays the search tree, representing different states and choices made during the computation. This visualization allows us to understand program flow, observe backtracking, and explore alternative solutions. The tree provides valuable insights into the execution process.

3. OCaml-GUI Interaction: The interface for seamless interaction between OCaml and was implemented successfully. This integration enables real-time updates and synchronization between the MiniKanren engine implemented in OCaml and the graphical user interface designed using QML. As the program executes or user interactions occur, the visualizer dynamically reflects these changes, providing an interactive and responsive experience.

4. Parser and Lexer: A prototype of the parser and lexer for MiniKanren programs was developed. These components allow for parsing the input code and detecting basic syntax

errors during the build process. What is more important it allows to run programs directly from the window.

5 Perspectives

The implementation of the MiniKanren visualizer prototype opens up several exciting avenues for future development and enhancement. Here are some potential directions for further improvement:

1. Syntax Highlighting: Implementing syntax highlighting in the code window can significantly improve the user experience. Syntax highlighting can visually distinguish different elements of the MiniKanren language, making the code more readable and aiding in error detection.
2. Advanced Error Reporting: Enhancing the parser and lexer to provide more detailed and informative error messages can assist users in identifying and resolving syntax errors more efficiently. Clear and descriptive error messages can guide users towards rectifying issues and improving code quality.
3. Constraint Visualization: If working with constraint logic programming variants of MiniKanren, incorporating visualizations for constraints and their propagation can aid in understanding and debugging constraint satisfaction problems. Visualizing constraints and their interactions can help users identify inconsistencies and refine their constraint specifications.
4. Improving current application Interface in order to make it look intuitive and fully functional. As well it would be great to implement user-friendly controls so it will be easy to manipulate the execution and writing the program.
5. Integration with IDEs: Integrating the MiniKanren visualizer as a plugin or extension for popular integrated development environments (IDEs) can streamline the development workflow. Seamless integration with IDEs can provide features such as code completion, documentation lookup, and integrated debugging, enhancing productivity and ease of use.

6 Conclusion

In summary, the implementation of the MiniKanren visualizer demonstrates the concept of programming in miniKanren in suitable way with the possibility to inspect code and run various programs with the visualization of all processes, which has a big step to simplification of the programming on miniKanren. While syntax highlighting was not included in this iteration, the prototype showcases the potential for further enhancements and future development of the visualizer, including additional features and error detection capabilities in the parser and lexer.