Fail Fast and Profile On: Towards a miniKanren Profiler

SLOAN CHOCHINOV*, University of Toronto Mississauga, Canada
DAKSH MALHOTRA*, University of Toronto Mississauga, Canada
GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada

We present a prototype of a miniKanren profiling tool. The profiler tracks, for each goal in a miniKanren program, the number of times that it fails, succeeds, and is encountered. Moreover, the profiler tracks and displays samples of failing states. We demonstrate with a few examples that these statistics are useful for reordering relations in conjuncts using a "fail fast" strategy to improve performance.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: miniKanren, relational programming, logic programming, profiling, debugging

1 INTRODUCTION

Debugging and profiling miniKanren programs is challenging for even the most accomplished users of the language. In particular, the order of the conjuncts and disjuncts in a miniKanren relation definition significantly affects whether queries can run in a reasonable time. Some general guidelines exist, like placing primitive goals before goals involving recursive user-defined relations. Still, it is generally difficult to predict which orders of conjuncts and disjuncts is optimal.

To that end, we present a prototype of a miniKanren profiler that helps miniKanren developers find more optimal orderings of conjuncts and disjuncts. The profiler tracks and displays, for each goal in a miniKanren program (which we call a *program point*), statistics related to its execution. Specifically, the profiler interface displays the number of encounters, successes, and failures at each program point. Our prototype is available at https://github.com/DakshChan/mk_debugger.

These statistics are helpful for debugging logic programs: goals that are more constraining (high failure) eliminate a larger portion of the search space and should typically be placed earlier in a conjunction. Goals that produce more branches (high success) maintain or increase the size of the search space, and should typically be placed later in a conjunction.

*Both authors contributed equally to the paper

Authors' addresses: Sloan Chochinov, University of Toronto Mississauga, Mississauga, ON, Canada, sloan.chochinov@mail.utoronto.ca; Daksh Malhotra, University of Toronto Mississauga, Mississauga, ON, Canada, d.malhotra@utoronto.ca; Gregory Rosenblatt, University of Alabama at Birmingham, Birmingham, AL, USA, gregr@uab.edu; Matthew Might, University of Alabama at Birmingham, Birmingham, AL, USA, might@uab.edu; Lisa Zhang, University of Toronto Mississauga, Mississauga, ON, Canada, lczhang@cs.toronto.edu.

miniKanren.org/workshop/2021/8-ART1

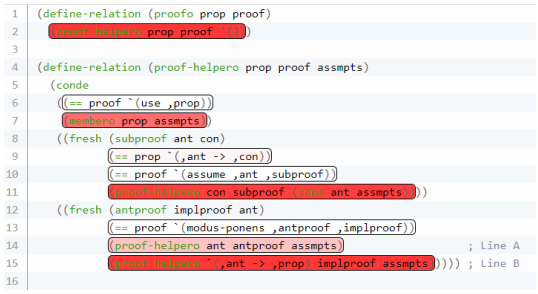

Fig. 1. Slow version of proofo, with the saturation of highlighting showing failure ratio. Notice that `Line B` has a higher failure ratio than `Line A`, potentially indicating that `Line B` should be moved earlier. See Appendix A.1 for a textual version of this figure.

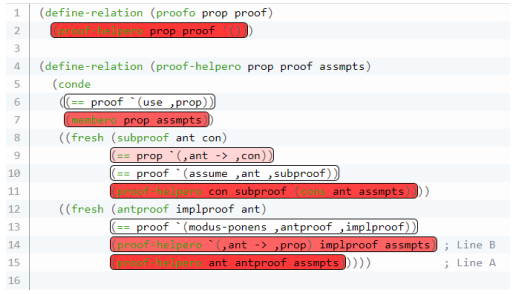

Fig. 2. Faster version of proofo, with the saturation of highlighting showing failure ratio. Notice that the difference in failure ratio between `Line A` and `Line B` is closer than in Figure 1. See Appendix A.2 for a textual version of this figure.

1.1 Example: Profiling a Relational Theorem Prover

We begin with an example that illustrates how the profiler could be used to help resolve performance issues. Suppose that we would like to write a relational proof checker and theorem prover for the purely implicational minimal logic, similar to the one presented in Byrd et al. [3]. Such a logic allows for two types of propositions: constants and implications. A proof in this logic uses a combination of these three rules: `use` which makes use of a proposition available in an assumption, `assume` which makes a new proposition available as an assumption, and `modus-ponens` which uses the proof of an implication's hypothesis to prove the implication's conclusion. Our relation is called `proofo`, and we would like to use `proofo` to synthesize propositions and proofs using queries like:

```
> (run 1 (pf) (proofo '(A -> A) pf))
'(assume A (use A))
> (run 1 (prop) (proofo prop '(assume (C -> D)
                                      (assume C
                                              (modus-ponens (use C)
                                                            (use (C -> D)))))))
'((C -> D) -> (C -> D))
> (run 1 (pf) (proofo '(((A -> (B -> C)) -> C) -> ((B -> (A -> C)) -> C)) pf))
?
```

Following the rules of this logic, we write the relation `proofo` and `proof-helpero` as shown in Figure 1. For simple queries like the first two above, miniKanren returns results quickly. However the final query shown above does not terminate in a reasonable time.

Reordering the conjuncts may change the performance characteristics of this relation, and we would like to find these performance enhancements in a reasoned way using our profiler. To do so, we load the program into the profiler and run the query for 100,000 steps; the query runs for approximately 600ms. The resulting *failure ratio* at each program point is shown in Figure 1 by red highlighting[1]. Notice that the ratio of states rejected by *Line A* is much smaller than *Line B*, as indicated by the strength of the red highlighting. This implies that many states survive past the first recursive relation call, only to be rejected by the second.

To improve the performance of our relation, we place the conjunct with the higher failure ratio first: we reverse *Line A* and *Line B* to produce the relation in Figure 2. When we run the same query

[1]For accessibility purposes, textual representations and descriptions of figures are included in Appendix A.

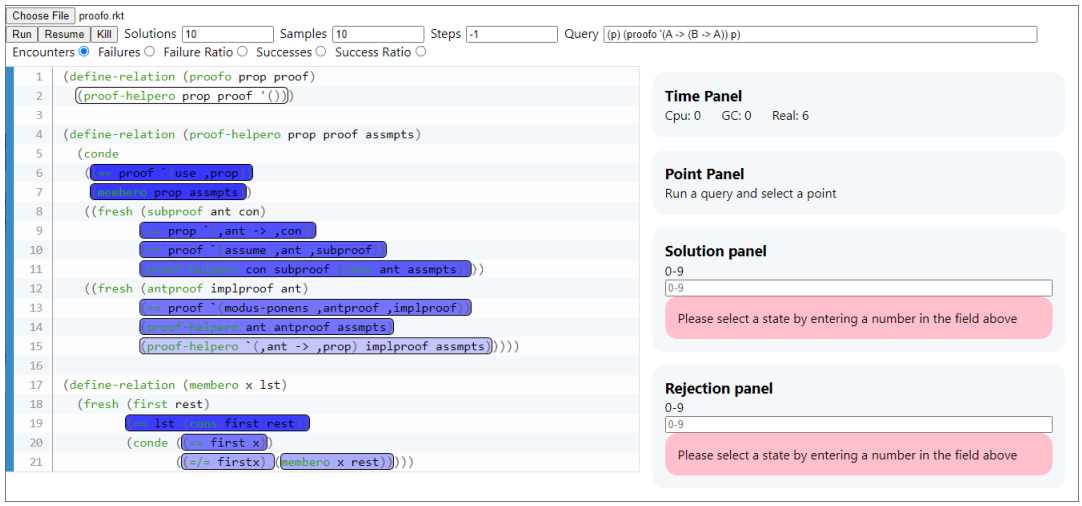

Fig. 3. The profiler interface with a file loaded, showing available parameters and controls. Here, the query `(p) (proofo '(A -> (B -> A)) p)` is run with 10 solutions, 10 failed samples, and unbounded steps. See Appendix A.3 for a textual description of this figure.

for 100,000 steps, the difference in failure ratio between *Line A* and *Line B* is much smaller. Indeed, attempting to query for a solution with the rewritten relation yields results almost instantly.

The rest of this paper is structured as follows: Section 2 describes the interface of our profiler, and defines terms that we use throughout the paper. Section 3 describes the profiler implementation. Section 4 contains several examples to show how various features of the profiler can be used to optimize or debug miniKanren programs. Section 5 outlines related work, and Section 6 concludes the paper with suggested future work.

2 PROFILER USAGE

We provide a web interface for users to interact with the profiler, shown in Figure 3. Users press the **Choose File** button to load a file containing relation definitions. The file contents will then appear in the code window.

Users run queries by entering the query body in the **Query** field, and then clicking the **Run** button. An example of a query body is "`(x) (== x 1)`". When running a query, we provide the option to constrain the number of solutions found or the number of steps taken (defined below). We also provide the option to uniformly randomly sample failures as the query runs. Users can limit the number of solutions found, number of failed states sampled, and number of steps taken by modifying the **Solutions**, **Samples**, and **Steps** fields, respectively. The default value for these fields, -1, causes the corresponding parameter to be treated as unbounded.

If a query execution reaches the step limit, its stream continuation is saved. Users can resume computation of the query by clicking the **Resume** button.

Recall that a **solution** is a state whose constraints succeed under the specifications of the query. In particular, whenever execution of a query yields a mature stream containing a state `st` and a continuation of the stream, we say that `st` is a solution. Conversely, whenever miniKanren fails to evaluate a constraint on some state `st` during execution of a query, we say that `st` is a **failed state**. Finally, a **step** is one call to the `step` procedure as described in Rosenblatt et al. [7].

2.1 Failure and Success Statistics

Our profiler tracks and displays statistics associated with each **program point** – a primitive goal (==, =/=, type, not-type constraints) or a call to a user-defined relation – encountered during a query. We do not track statistics for uses of `conde` and `fresh`, because those can be reconstructed from the statistics of these atomic goals.

Figure 3 shows the profiler interface when a query is run. All program points encountered during execution of the query are highlighted. By default, the saturation of the highlighting represents the number of times a program point was encountered during the query. However, by selecting a radio button option above the code panel, users can view program point highlighting representing any of five metrics:

- **Encounters**: A program point is encountered whenever it is evaluated by miniKanren.
- **Failures**: A failure is attributed to a primitive goal when its evaluation does not return a state. A failure is attributed to a relation goal when any of its sub-goals have failed.
- **Successes**: A success is attributed to a primitive goal when its evaluation returns a state. A success is attributed to a relation goal only when all of its sub-goals have succeeded.
- **Failure Ratio**: $= \frac{\text{Failures}}{\text{Successes} + \text{Failures}}$
- **Success Ratio**: $= \frac{\text{Successes}}{\text{Successes} + \text{Failures}}$

In our experience, a combination of these metrics is useful for debugging: raw failure count works as a default starting point; failure ratio works well to identify goals that are never succeeding; successes and encounters are useful for distinguishing two goals that have similar failure counts.

2.2 Additional Panels

Figure 3 shows that to the right of the code panel, four additional panels are displayed.

The **time panel** displays information about timing of the state. In this implementation, we get timing information from the Racket procedure `time-apply`.

The **point panel** can display information about any highlighted program point that is clicked in the code panel. This information includes encounters, failures, successes, the location of the program point in the code by line and column, and the contents of the program point.

The **solution panel** can display information about any solution yielded by the query. Each solution is assigned a number starting from 0, which can be specified in the number entry field in the solution panel. The information shown for each solution includes the bindings of the query parameters, any additional constraints present in the state, and the state's path and stack. We define a state's **path** as the sequence of program points that were applied to an empty state in order to arrive at the current state. We define a state's **stack** as the current evaluation stack of a state, particularly with respect to relations.

The **rejection panel** is similar to the solution panel, instead displaying information about any of the failed states that were randomly sampled during execution of the query. The information shown for each failed state includes the bindings of the query parameters, any additional constraints present in the state, a reification of the primitive goal that caused the state to fail, all constraints relevant to the reification of that primitive goal, and the state's path and stack.

3 IMPLEMENTATION

For this implementation, we build upon the first-order miniKanren implementation from Rosenblatt et al. [7]. In first-order miniKanren, goals are first-order structures (rather than procedures) which can be examined during the evaluation of a query. Using first-order miniKanren allows us to decouple the representation of goals from their behaviour during evaluation, which makes it easier

to alter the behaviour of the profiler. To be clear, our current approach can be implemented in a typical higher-order miniKanren representation. However, we use a first-order representation of goals to leave room for extending debugging features to those that require introspection of goals, or for allowing the debugger to automate live program manipulation.

3.1 Capturing Syntax

Our implementation exploits the ability of first-order miniKanren to augment goals with additional information. In particular, we augment the primitive goal representation with an additional `stx` field to capture the source code of primitive goals. We are able to do this by defining a primitive goal constructor as a macro. This macro captures the source code of the goal in a `syntax` object and records it in the primitive goal's representation.

In our miniKanren implementation, a user-defined relation is defined via the `define-relation` macro, which captures its source code. We factor the result of expanding this macro into a procedure definition to encompass the body of a relation in a goal, and another macro definition for calling the relation. This factoring avoids infinite macro expansion.

3.2 Tracking Statistics

To track information about successes and failures at each program point, we use a global hash map whose keys are `syntax` objects and whose values are hash maps assigning an integer to each of the symbols `count`, `successes`, and `fails`.

To track information about failed states, we use a global list of `state` structures.

When miniKanren evaluates a program point on a state, we increment the `count` of the primitive goal's syntax in the global hash map. Then, if the program point is a primitive goal, we consider two cases: (1) If applying a primitive goal's constraint on the state fails, we increment the `fails` of the primitive goal's syntax in the hash map, as well as the `fails` of user-defined relation calls that are currently being evaluated. The list of such relation calls is available in the state's `stack`, which we describe in Section 3.3. We also randomly determine whether to add the current state to the global list of failed states, such that any failed state encountered during a query has equal probability to appear in the list. (2) If applying a primitive goal's constraint succeeds, we increment the `successes` of the primitive goal's syntax in the hash map.

A call to a user-defined relation never fails directly, only by being on a state's `stack` when that state fails. A call to a user-defined relation succeeds via `pop` goals described in Section 3.3.

3.3 Tracing Execution

We augment a state with two additional fields, `stack` and `path`, each of which is an ordered list of `syntax` objects representing program points. The `stack` field supports the attribution of successes and failures to user-defined relation calls. The `path` field tracks the entire program point encounter history of the state, and may be useful for future work. Both fields are displayed in the "Solution" and "Rejection" panels described in Section 2.2.

When miniKanren evaluates a program point on a state, we extend the `path` of the state with the `syntax` object of the program point.

When miniKanren evaluates a relation goal on a state, we extend the `stack` of the state with the `syntax` object of the relation goal, and we conjunct the goals produced by the relation's `thunk` with a new goal that we call `pop`. This ensures that once the goals produced by the relation's `thunk` have been evaluated on the state, the next goal evaluated on the state is `pop`.

We use the `pop` goal to manage the stack. When miniKanren evaluates `pop` on a state, we pop the top element of the state's `stack` and increment its `successes` as described in Section 3.2. In this sense, `pop` is a marker for the completion of the relation goal at the top of the stack.

```
(define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
    (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
        ; Shift the mantissa of the SMALLER exponent
        (mantissa-shifto mant1 expo-diff shifted-mant1)

        ; oleg number addition
        (pluso shifted-mant1 mant2 mant-sum)

        ; Rounding by chopping
        (drop-leastsig-bito mant-sum pre-rmant bit)

        ; Add 1 to the exponent as necessary
        (conde ((== bit '()) (== pre-rexpo expo2))
               ((=/= bit '()) (pluso '(1) expo2 pre-rexpo)))

        ; Check for overflow
        (fp-overflowo pre-rexpo pre-rmant rexpo rmant)
))
```

Fig. 4. An initial implementation of floating-point addition. The highlighting shows number of failures. Notice how the call to `fp-overflowo` is highly saturated indicating a large number of failures at this program point. We move this call earlier in Figure 5. See Appendix A.4 for a textual version of this figure.

4 EXAMPLES

In this section we present four examples of usage of the profiler. Section 4.1 shows how the **failure and success** metrics are used to debug an implementation of the floating point number system, remedying a query from non-termination. Section 4.2 shows how the **rejection panel** is used to debug an implementation of selection sort, solving a dilemma of performance tradeoff between two types of queries. Section 4.3 shows how the **solution panel** is used to debug `membero`, fixing extraneous solutions. Section 4.4 shows an attempt to optimize a regular expression matching relation, highlighting the limitations of profiling on a single query.

4.1 Profiling Floating-Point Addition

In this section, we profile a variation of the floating-point addition relation presented in Sandre et al. [8]. In particular, we profile and recreate the relation `fp-samesignaddero`, which performs floating-point addition given that the addends have the same sign, and that the first addend is larger than the second. In Sandre et al. [8], core logic for `fp-pluso` is located in this `fp-samesignaddero` relation. We aim to show how to profile long-running queries calling `fp-samesignaddero`.

We start with an ordering of the conjuncts, shown in Figure 4, consistent with how a human would describe floating-point addition. First, the mantissa of the smaller exponent is shifted by the difference in the exponents, resulting in `shifted-mant1`. Then we add the newly shifted mantissa to the larger mantissa, resulting in `mant-sum`. We apply rounding by chopping, and add 1 to shift the resulting exponent if necessary. Finally, we check for overflow on the exponent.

Unfortunately, when running the query `(run 1 (x) (fp-pluso one x pi))` to compute $\pi - 1$, miniKanren does not return any result within a reasonable time frame, suggesting that we should explore other conjunct orderings.[2]

Figure 4 shows the failure counts at each program point when we run the above query for 1,000,000 steps (approximately 10 seconds). Notice that only the call to `fp-overflowo` is highlighted in red, indicating that the call to `fp-overflowo` has significantly more failures than the other goals.

As an attempt to improve performance, we move the call to `fp-overflowo` to be the first conjunct. We see the results of this change in Figure 5. Moreover, Figure 5 shows that the calls to `pluso`

[2]We choose this query following an anecdotal recounting of performance issues in developing floating-point arithmetic in miniKanren.

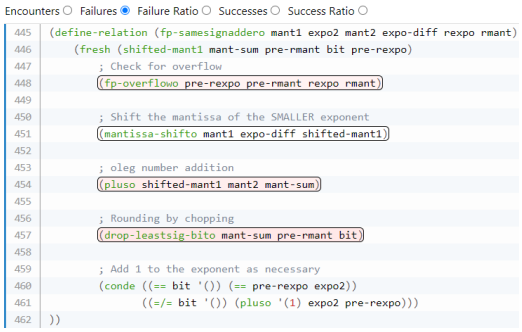

Fig. 5. Floating-point addition with `fp-overflowo` as the first conjunct, with highlight saturation showing failure count. Both `drop-leastsig-bito` and `pluso` have high failure counts. See Appendix A.5 for a textual version of this figure.

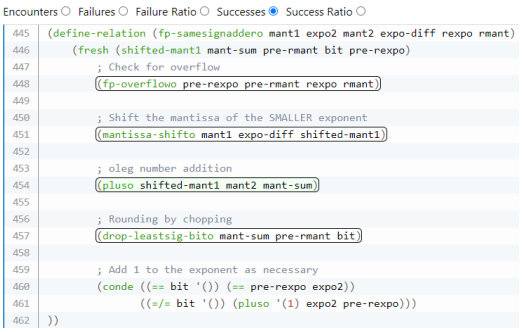

Fig. 6. Floating-point addition with `fp-overflowo` as the first conjunct, with highlight saturation showing success count. Only `pluso` has a large number of successes. See Appendix A.6 for a textual version of this figure.

```
445 (define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
446     (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
447         ; Check for overflow
448         (fp-overflowo pre-rexpo pre-rmant rexpo rmant)
449
450         ; Shift the mantissa of the SMALLER exponent
451         (mantissa-shifto mant1 expo-diff shifted-mant1)
452
453         ; Rounding by chopping
454         (drop-leastsig-bito mant-sum pre-rmant bit)
455
456         ; Add 1 to the exponent as necessary
457         (conde ((== bit '()) (== pre-rexpo expo2))
458               ((=/= bit '()) (pluso '(1) expo2 pre-rexpo)))
459
460         ; oleg number addition
461         (pluso shifted-mant1 mant2 mant-sum)
462 ))
```

Fig. 7. The final version of `fp-samesignaddero`. See Appendix A.7 for a textual version of this figure.

and `drop-leastsig-bito` now have the largest number of failures. However, Figure 6 shows that `pluso` actually succeeds frequently, whereas `drop-leastsig-bito` does not.

To further improve performance, we move the call to `pluso` to be the last conjunct. The result is shown in Figure 7. At this point, running our initial query `(run 1 (x) (fp-pluso one x pi))` with this code returns the expected answer within 1 second.

4.2 Improving Selection Sort

In this section, we debug of a sorting relation for Peano numbers, with zero represented as the empty list, and the successor to a number `n` as `(cons 's n)`. We begin with an implementation corresponding to what a selection sort *function* might look like:

```
(define-relation (selection-sorto lst out)
  (conde ((== lst '()) (== out '()))
         ((fresh (out-fst out-rst sublst)
           (== out (cons out-fst out-rst))
           (remove-firsto lst out-fst sublst)
           (<=firsto out-fst out-rst)
           (selection-sorto sublst out-rst)))))
```

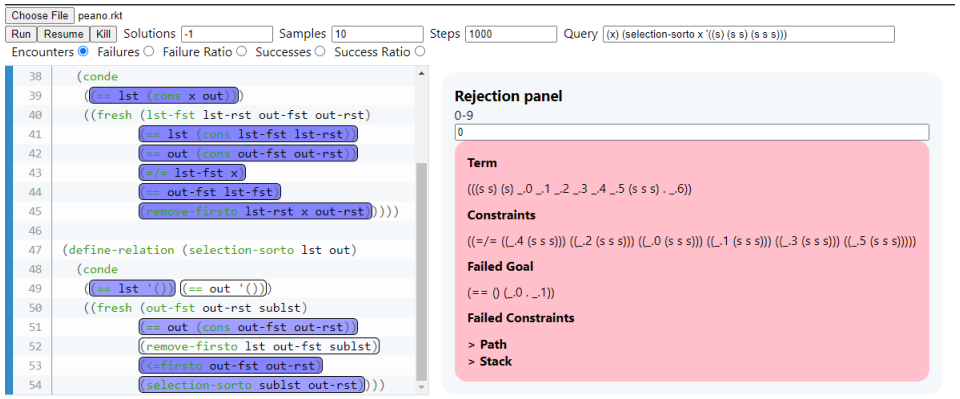

Fig. 8. The rejection panel displaying a sample state from the query `(x) (selection-sorto x '((s) (s s) (s s s)))`, run for 1000 steps. Notice that the query variable shown under **Term** is bound to a list longer than expected. See Appendix A.8 for a textual version of this figure, along with the helper relation definitions.

The relation performs well when running "forwards" to sort a list:

```
> (run* (x) (selection-sorto '((s s) (s) (s s s)) x))
'(((s) (s s) (s s s)))
```

The relation also performs well when running "backwards" to permute a sorted list:

```
> (run 6 (x) (selection-sorto x '((s) (s s) (s s s))))
'((((s) (s s) (s s s)))
  (((s) (s s s) (s s)))
  (((s s) (s) (s s s)))
  (((s s) (s s s) (s)))
  (((s s s) (s) (s s)))
  (((s s s) (s s) (s))))
```

However, if we ask for more solutions than exist, the query fails to terminate:

```
> (run 7 (x) (selection-sorto x '((s) (s s) (s s s))))
... ; does not terminate after 24 hours
```

Figure 8 shows a sample rejected state from running this query for 1,000 steps (approx 100ms). Here, the query variable is bound to a list of length >8, even though all answers must be length 3.

This discrepancy indicates that we may be able to prune failing states more quickly by adding a constraint to our relation, namely that `lst` and `out` must be the same length. In `selection-sorto`, we add a call to the relation `same-lengtho`, which succeeds if both lists are the same length:

```
(define-relation (selection-sorto lst out)
  (conde ((== lst '()) (== out '()))
         ((fresh (out-fst out-rst sublst)
           (== out (cons out-fst out-rst))
           (same-lengtho lst out)
           (remove-firsto lst out-fst sublst)
           (<=firsto out-fst out-rst)
           (selection-sorto sublst out-rst)))))
```

After making this change, the program terminates on both forwards and backwards queries.

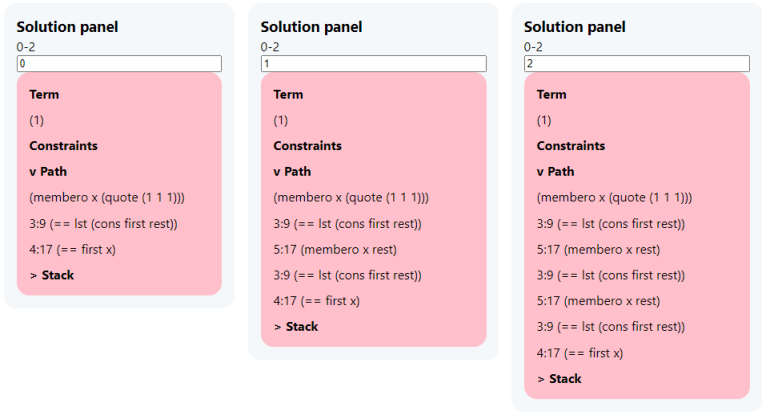

Fig. 9. The path of each of the solutions returned by the query `(x) (membero x '(1 1 1))`, using the implementation of membero presented at the start of Section 4.3. Selecting a solution in the text box shows the path taken to reach that solution. Each line of the path shows the `row:column` of the corresponding program point in the code. See Appendix A.9 for a textual version of this figure.

4.3 Debugging Membero

In this section we begin with the following, suboptimal definition of the relation `membero`:

```
#;1 (define-relation (membero x lst)
#;2   (fresh (first rest)
#;3          (== lst (cons first rest))
#;4          (conde ((== first x))
#;5                 ((membero x rest)))))
```

This definition correctly identifies the elements in a list:

```
> (run* (x) (membero x '(1 2 3)))
'((1) (2) (3))
```

However, we obtain the same answer multiple times when a list contains duplicate items:

```
> (run* (x) (membero x '(1 1 1)))
'((1) (1) (1))
```

To understand why these duplicate solutions occur, we visualize, in Figure 9, the path miniKanren takes to arrive at these three solutions. The figure shows that the path taken to arrive at each solution consists of zero to two repetitions of lines 3 and 5, followed by lines 3 and 4. Looking more closely, line 4 `(== first x)` corresponds to the base case of the `conde`, and line 5 `(membero x rest)` corresponds to the recursive case. To avoid repeated solutions, in the recursive call, we should *not* continue to look for `first`, as it is already considered in the base case. In other words, in order for `(membero x rest)` to be called, we would want `(=/= first x)` to be true.

We fix `membero` by adding the conjunct `(=/= first x)` to the recursive branch:

```
#;1 (define-relation (membero x lst)
#;2   (fresh (first rest)
#;3          (== lst (cons first rest))
#;4          (conde ((== first x))
#;5                 ((=/= first x) (membero x rest)))))
```

4.4 Attempting to Improve a Regular Expression Matcher

In this section we profile Byrd's implementation [2] of relational regular expression matcher with derivatives [5]. In doing so, we showcase a limitation of the profiler: optimizing the order of conjuncts for the performance of one query can hurt the performance of other queries.

This implementation of the relation `regex-matcho` determines whether the list of symbols `data` matches the regular expression pattern `pattern`, and equates `out` to either `#t` or `#f` [2]. To do so, the relation computes the derivative of `pattern` with respect to each successive symbol in `data` by calling `regex-derivativeo`.

```
#;137 (define-relation (regex-matcho pattern data out)
#;138   (conde
#;139     [(== '() data) (deltao pattern out)]
#;140     [(fresh (a d res)
#;141        (== `(,a . ,d) data)
#;142        (regex-derivativeo pattern a res)
#;143        (regex-matcho res d out))]))
```

To explore the performance characteristics of this relation, consider this query that generates regexes and corresponding matching non-empty data:

```
(run num-results
     (q)
     (fresh (regex data) (=/= '() data)
                         (regex-matcho regex data #t)
                         (== `(,regex ,data) q)))
```

The time the query takes to finish increases exponentially with the number of solutions, taking 2 seconds for 6 solutions, 4 seconds for 7 solutions, and 18 seconds for 8 solutions.

Inspecting the program points, we see 13 million failures attributed to the recursive call to `regex-matcho`. This indicates that the call to `regex-derivativeo` produces a large number of branches that will ultimately fail. We attempt to reduce the this branching by swapping the conjuncts on lines 142 and 143, breaking the general guideline that "recursive calls come last". After the swap, querying for 7 solutions returns instantly, and the number of failures of `regex-matcho` is reduced to 287. Even running for 100 solutions takes a tenth of a second.

Unfortunately, the swap impacts other queries. Consider a query that generates strings matching a given pattern:

```
(run num-results (q) (regex-matcho '(seq foo (rep bar)) q regex-BLANK))
```

Before swapping conjuncts, this query yielded 3 solutions in under a second. However, after the swap, searching for 3 solutions does not complete within 10 minutes.

This example highlights the dangers of over-optimizing for a single query. If our regex-generating query is the main focus, then swapping the conjuncts results in an improvement. However if performance across a variety of queries is required, then we need profiling data that takes those queries into account.

5 RELATED WORK

Nonlinear execution makes miniKanren programs difficult to debug and profile, and source language profilers tend to attribute cost to the miniKanren implementation code rather than the program code. There is little published work on miniKanren debugging. Rosenblatt et al. [7] describes a prototype stepper, built on a first-order implementation of miniKanren, that allows users to "drive"

a miniKanren search. Authors of miniKanren programs [3, 8] sometimes use manual tracking of program point metrics to improve relation performance.

Although there is little published work on miniKanren debugging, there are some publications on debugging logic programs in Prolog. Pereira [6] attempts to solve the challenge of debugging a program with nonlinear execution by implementing a program stepper driven by an algorithm that prompts the user about intended program behaviour in order to locate buggy clauses. We see a similar approach in Vidal [9], which implements a reversible stepper to allow the user to manually step forwards and backwards through a computation. Lee and Dershowitz [4] approaches debugging in logic programming with a different method, attempting to locate and correct bugs automatically, given some specifications about the intended behaviour of a program. We take inspiration from Zhang et al. [10], which describes similar interfacing strategies to our own, such as displaying performance statistics and random samples of states to the user.

6 CONCLUSION AND FUTURE WORK

In this work we present a prototype for a miniKanren profiling tool. The profiler records information about solutions, failures and their relevant constraints, and displays a visual representation of the quantity of successes and failures attributed to program points. Our strategy of allowing the execution of a query to be paused and resumed at any point allows profiling of a non-terminating program. Our hope is that such a tool could help beginning miniKanren programmers write better relations using the "fail fast" strategy.

We believe that a more fully-featured profiler and debugger could help such programmers. In particular, future work could involve:

- Making accessible the state at any point during program execution for transparency.
- Reconstructing statistics for `conde` and `fresh` to help reorder larger blocks.
- Tracking which primitive goal produces each constraint to infer provenance of failures.
- Allowing tracking of multiple queries at once to summarize overall performance.

Finally, it may be possible to use the statistics tracked by the profiler to perform dynamic goal-reordering and other optimizations automatically or programmatically [1]. Such optimization would allow different goal orders for different queries.

ACKNOWLEDGMENTS

We thank William E. Byrd for stumping us with the relational regular expression matcher, and Daniel Zingaro for feedback on accessibility.

REFERENCES

[1] William J Bowman, Swaha Miller, Vincent St-Amour, and R Kent Dybvig. 2015. Profile-guided meta-programming. *ACM SIGPLAN Notices* 50, 6 (2015), 403–412.

[2] William E. Byrd. 2013. Relational parsing with derivatives. https://github.com/webyrd/relational-parsing-with-derivatives/blob/2b5fea523ed2b35fb1687954449ee75dc67e94dd/miniKanren-version/mk-rd.scm.

[3] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–26.

[4] Yuh-jeng Lee and Nachum Dershowitz. 1993. Logical Debugging. In *Automatic Programming 15 (5–6): 745–773, May/June*. Citeseer.

[5] Matthew Might. [n. d.]. A regular expression matcher in Scheme using derivatives. https://matt.might.net/articles/implementation-of-regular-expression-matching-in-scheme-with-derivatives/.

[6] Luis Moniz Pereira. 1986. Rational debugging in logic programming. In *International Conference on Logic Programming*. Springer, 203–210.

[7] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-order miniKanren representation: Great for tooling and search. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop*.

[8] Lucas Sandre, Malaika Zaidi, and Lisa Zhang. 2021. Relational floating-point arithmetic. In *Proceedings of the 2021 miniKanren and Relational Programming Workshop*.

[9] Germán Vidal. 2020. Reversible Debugging in Logic Programming. *arXiv preprint arXiv:2007.16171* (2020).

[10] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.

A TEXTUAL REPRESENTATIONS OF THE FIGURES

In this section, we include textual representations and descriptions of figures for accessibility purposes. Where applicable, the relevant program point statistic is shown as an s-expression comment at the beginning of each line of code.

A.1 Textual Representation of Figure 1

Slow version of proofo, with commented number before each line indicating failure ratio. Notice that the last line, `Line B`, has a higher failure ratio than the second to last line, line `Line A`, potentially indicating that `Line B` should be moved earlier.

```
       (define-relation (proofo prop proof)
#;1      (proof-helpero prop proof '()))

       (define-relation (proof-helpero prop proof assmpts)
         (conde
#;0        ((== proof `(use ,prop))
#;0.72     (membero prop assmpts))
           ((fresh (subproof ant con)
#;0.04             (== prop `(,ant -> ,con))
#;0                (== proof `(assume ,ant ,subproof))
#;0.96             (proof-helpero con subproof (cons ant assmpts))))
           ((fresh (antproof implproof ant)
#;0                (== proof `(modus-ponens ,antproof ,implproof))
#;0.30            (proof-helpero ant antproof assmpts)                  ; Line A
#;0.99           (proof-helpero `(,ant -> ,prop) implproof assmpts))))) ; Line B
```

A.2 Textual Representation of Figure 2

Faster version of proofo, with commented number before each line indicating failure ratio. Notice that the difference in failure ratio between the last line, `Line A`, and the second to last line, `Line B`, is closer than in Figure 1.

```
       (define-relation (proofo prop proof)
#;0.99   (proof-helpero prop proof '()))

       (define-relation (proof-helpero prop proof assmpts)
         (conde
#;0        ((== proof `(use ,prop))
#;0.81     (membero prop assmpts))
           ((fresh (subproof ant con)
#;0.21             (== prop `(,ant -> ,con))
#;0                (== proof `(assume ,ant ,subproof))
#;0.95             (proof-helpero con subproof (cons ant assmpts))))
           ((fresh (antproof implproof ant)
#;0                (== proof `(modus-ponens ,antproof ,implproof))
#;0.79             (proof-helpero `(,ant -> ,prop) implproof assmpts) ; Line B
#;0.99             (proof-helpero ant antproof assmpts)))))           ; Line A
```

A.3 Textual Representation of Figure 3

This image shows the profiler interface with the file from Figure 2 loaded and profiled.

The top part of the profiler is the control panel. At the very top, there is a button that allow a user to “Choose File” to select a file to run. Below that, there are three buttons “Run”, “Resume”, and “Kill” that controls the interface, along with a textbox for a user to select the max number of “Solutions”, number of states or “Samples”, and the max number of “Steps” to take. In the figure, the max “Solutions” and “Samples” are both set to 10, and the number of “Steps” set to -1 to indicate the lack of a bound. Finally, there is a “Query” textbox that accepts the arguments to a typical `run` or `run*` query. In the figure, the query that was run is `(p) (proofo '(A -> (B -> A)) p)`.

There is a radio box for selecting the metric to visualize. In the figure, “Encounters” is selected, rather than “Failures”, “Failure Ratio”, “Successes”, or “Success Ratio”.

Below the control panel, the code panel is on the left and takes up the remaining vertical space. The code shown is the file containing the Figure 2 code.

To the right of the code, there are 4 additional panels: a “Time Panel” showing that the query finished within 5 ms, a “Point Panel” that is empty, a “Solution” panel that does not yet show a solution since none is selected, and a “Rejection” panel that does not yet show a rejected state since none is selected.

A.4 Textual Representation of Figure 4

An initial implementation of floating-point addition. The commented number before each line shows the number of failures. Notice how the call to `fp-overflowo` in the last line has a high number of failures. We move this call earlier in Figure 5.

```
      (define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
             (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
                 ; Shift the mantissa of the SMALLER exponent
#;1474           (mantissa-shifto mant1 expo-diff shifted-mant1)

                 ; Oleg number addition
#;15445          (pluso shifted-mant1 mant2 mant-sum)

                 ; Rounding by chopping
#;16215          (drop-leastsig-bito mant-sum pre-rmant bit)

                 ; Add 1 to the exponent as necessary
#;85             (conde ((== bit '())
#;0                     (== pre-rexpo expo2))
#;890                  ((=/= bit '())
#;16069                 (pluso '(1) expo2 pre-rexpo)))

                 ; Check for overflow
#;290784         (fp-overflowo pre-rexpo pre-rmant rexpo rmant)))
```

A.5 Textual Representation of Figure 5

Floating-point addition with `fp-overflowo` as the first conjunct, with commented number before each line showing failure count. Both `drop-leastsig-bito` and `pluso` have high failure counts.

```
      (define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
             (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
                 ; Check for overflow
#;71482          (fp-overflowo pre-rexpo pre-rmant rexpo rmant)

                 ; Shift the mantissa of the SMALLER exponent
#;712            (mantissa-shifto mant1 expo-diff shifted-mant1)

                 ; Oleg number addition
#;94222          (pluso shifted-mant1 mant2 mant-sum)

                 ; Rounding by chopping
#;130438         (drop-leastsig-bito mant-sum pre-rmant bit)

                 ; Add 1 to the exponent as necessary
                 (conde ((== bit '())
                         (== pre-rexpo expo2))
                        ((=/= bit '())
                         (pluso '(1) expo2 pre-rexpo)))))
```

A.6 Textual Representation of Figure 6

Floating-point addition with `fp-overflowo` as the first conjunct, with commented number showing success count. Only the call to `pluso` has a large number of successes. We move this call later in Figure 7.

```
     (define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
           (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
               ; Check for overflow
#;27           (fp-overflowo pre-rexpo pre-rmant rexpo rmant)

               ; Shift the mantissa of the SMALLER exponent
#;27           (mantissa-shifto mant1 expo-diff shifted-mant1)

               ; Oleg number addition
#;7124         (pluso shifted-mant1 mant2 mant-sum)

               ; Rounding by chopping
#;0            (drop-leastsig-bito mant-sum pre-rmant bit)

               ; Add 1 to the exponent as necessary
               (conde ((== bit '())
                       (== pre-rexpo expo2))
                      ((=/= bit '())
```

```
                    (pluso '(1) expo2 pre-rexpo)))))
```

A.7 Textual Representation of Figure 7

The final version of `fp-samesign-addero`.

```
(define-relation (fp-samesignaddero mant1 expo2 mant2 expo-diff rexpo rmant)
    (fresh (shifted-mant1 mant-sum pre-rmant bit pre-rexpo)
        ; Check for overflow
        (fp-overflowo pre-rexpo pre-rmant rexpo rmant)

        ; Shift the mantissa of the SMALLER exponent
        (mantissa-shifto mant1 expo-diff shifted-mant1)

        ; Rounding by chopping
        (drop-leastsig-bito mant-sum pre-rmant bit)

        ; Add 1 to the exponent as necessary
        (conde ((== bit '())
                (== pre-rexpo expo2))
               ((=/= bit '())
                (pluso '(1) expo2 pre-rexpo)))

        ; Oleg number addition
        (pluso shifted-mant1 mant2 mant-sum)
))
```

A.8 Textual Representation of Figure 8

The rejection panel displaying a sample state from the query `(x) (selection-sorto x '((s) (s s) (s s s)))`, run for 1000 steps. Notice that the query variable shown under **Term** is bound to a list longer than expected.

Rejection Panel:

```
Term
(((s s) (s) _.0 _.1 _.2 _.3 _.4 _.5 (s s s) _.6))

Constraints
((=/= ((_.4 (s s s))) ((_.2 (s s s))) ((_.0 (s s s)))
      ((_.1 (s s s))) ((_.3 (s s s))) ((_.5 (s s s)))))

Failed Goal
(== () (_.0 . _.1))

Failed Constraints

> Path
> Stack
```

The definitions of the `selection-sorto` helper relations are as follows:

```
(define-relation (same-lengtho lst1 lst2)
  (conde
   ((== lst1 '()) (== lst2 '()))
   ((fresh (fst1 rst1 fst2 rst2)
           (== lst1 (cons fst1 rst1))
           (== lst2 (cons fst2 rst2))
           (same-lengtho rst1 rst2)))))

(define-relation (remove-firsto lst x out)
  (conde
   ((== lst (cons x out)))
   ((fresh (lst-fst lst-rst out-fst out-rst)
           (== lst (cons lst-fst lst-rst))
           (== out (cons out-fst out-rst))
           (=/= lst-fst x)
           (== out-fst lst-fst)
           (remove-firsto lst-rst x out-rst)))))

(define-relation (<=firsto x lst)
  (conde
   ((== lst '()))
   ((fresh (lst-fst lst-rst)
           (== lst (cons lst-fst lst-rst))
           (<=o x lst-fst)))))
```

A.9 Textual Representation of Figure 9

The path of each of the solutions returned by the query `(x) (membero x '(1 1 1))`, using the implementation of `membero` presented at the start of Section 4.3. Each line of the path shows the `row:column` of the corresponding program point in the code.

Solution 0 Path:

```
(membero x (quote (1 1 1)))
3:9 (== lst (cons first rest))
4:17 (== first x)
```

Solution 1 Path:

```
(membero x (quote (1 1 1)))
3:9 (== lst (cons first rest))
5:17 (membero x rest)
3:9 (== lst (cons first rest))
4:17 (== first x)
```

Solution 2 Path:

```
(membero x (quote (1 1 1)))
3:9 (== lst (cons first rest))
5:17 (membero x rest)
3:9 (== lst (cons first rest))
5:17 (membero x rest)
3:9 (== lst (cons first rest))
4:17 (== first x)
```